Politecnico di Torino

Politecnico di Torino

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

A.y. 2025/2026

Graduation Session March 2026

Privacy Dashboard

Integrating the Smartotum Home Automation System, Frontend Architecture, and CI/CD Pipeline

Supervisor:
Luca Ardito

Candidate:
Alberto Hugonin

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, with a focus on the integration with Smartotum, which enables the user to retrieve their smart home context, to manage and withdraw consents per application and home, and automatically enforce those choices through device reconfiguration. In addition, the work includes a proof of concept integration with an application marketplace, and the design of a CI/CD pipeline that automatically deploys an online demo instance. Finally, a retrospective of our work is conducted, in order to examine the constraints of our software and how it could be further improved in the future.

Acknowledgements

I would like to thank my supervisor, Prof. Luca Ardito, for giving me the opportunity to work on a real project in the field of home automation, an area that has always interested me. This experience made this thesis more concrete and gave me the chance to work on something closely connected to real-world problems. I would also like to thank my colleagues, Giorgio and Eric, who shared this project with me, for the collaboration and for everything we built together during this experience.
I am very grateful to my family for all the support they gave me throughout my university journey. Their encouragement has always meant a lot to me. A sincere thanks also goes to my friends, who have been close to me throughout these years. A special thanks goes to my partner, Aurora, whose support during the last months has been fundamental to me. Her encouragement and closeness helped me face this final stage with greater strength and calm.

Table of Contents

List of Figures

Glossary

GDPR
: General Data Protection Regulation

IoT
: Internet of Things

ENISA
: European Union Agency for Cybersecurity

DDoS
: Distributed Denial of Service

HTTP
: Hypertext Transfer Protocol

URL
: Uniform Resource Locator

DBMS
: Database Management System

CRUD
: Create, Read, Update, Delete

ACID
: Atomicity, Consistency, Isolation, Durability

SQL
: Structured Query Language

REST

Representational State Transfer

HTML

HyperText Markup Language

PDF

Portable Document Format

API

Application Programming Interface

DPO

Data Protection Officer

CI

Continuous Integration

CD

Continuous Delivery

E2E

End-to-End Testing

JSON

JavaScript Object Notation

TLS

Transport Layer Security

WSGI

Web Server Gateway Interface

SIFIS-Home

Secure Interoperable Full-Stack Internet of Things for Smart Home

PTP

Policy Translation Point

DHT

Distributed Hash Table

NSSD

Not-So-Smart Devices

MPA

Multi-Page Application

SPA

Single Page Application

DOM

Document Object Model

PIA

Privacy Impact Assessment

JWT

JSON Web Token

JWK

JSON Web Key

JWKS

JSON Web Key Set

UUID

Universally Unique Identifier

Chapter 1

Introduction

1.1 Context

The growing adoption of IoT devices in domestic environments, such as sensors, cameras, and voice assistants, turns the Smart Home into a dense ecosystem of personal data. This is already a mainstream phenomenon: the UK data protection authority reports that "four in five Brits" own at least one smart product. [1]

Because these devices operate inside private spaces, they can collect highly sensitive information about inhabitants, including habits, routines, preferences and presence within the home. This creates concrete privacy risks, such as data exposure, third party sharing and limited user awareness about how personal data are processed.

To overcome these concerns, the SIFIS-Home project proposed a secure by design framework for privacy in smart homes, while Smartotum provides a home automation system that brings this vision into practice. This infrastructure alone is not enough. The main GDPR actors, such as Data Subjects, Data Controllers and Data Protection Officers need a tool able to manage consents, rights requests and GDPR documents in relation to the actual devices and applications deployed in the home.

In this context, the Privacy Dashboard was introduced with the goal of making privacy management concrete in smart home environments by supporting the main GDPR actors in handling their privacy related activities. The legacy prototype already moved in this direction, but it still lacked integration with a real home automation system and support for some essential GDPR functionalities. For this reason, this thesis focuses on the development of a more complete dashboard connected to a real smart home.

1.2 Project Goals

The software developed during this project was based on the previous Privacy Dashboard prototype introduced in the SIFIS-Home initiative and aimed to overcome its main limitations by evolving it into a single platform integrated with a real smart home.

Although the general architecture was already in place and some of the main functionality had already been implemented, the software was still divided into separate parts: the original Privacy Dashboard, developed in Java and Vaadin and the Policy Translation Point as a separate Java application.

What was still missing was the integration with a real home automation system, needed to make the Policy Translation Point effective in practice: the high level privacy rules defined by the user could already be translated into a formal representation, but without a smart home capable of supporting the corresponding actions they could not be enforced on real devices. From these limitations, the project developed around three main goals. The first was to redesign the Privacy Dashboard as a more modern and maintainable platform, introducing a clear separation between frontend and backend and adopting Flask and React as the new technology stack. The second was to incorporate the Policy Translation Point directly into its codebase. The third was to design and implement the integration with Smartotum, so that the platform could operate on a real smart home and enforce privacy rules on real devices. The project also aimed to keep the new platform aligned with the broader SIFIS-Home ecosystem and ready for further development.

Within this shared effort, foundational decisions such as the choice of the technology stack and the general system architecture were taken collectively, while each team member was individually responsible for a distinct area of the platform:

- One colleague focused on the core backend infrastructure, end to end testing and frontend components for GDPR document management.
- Another colleague focused on the Policy Translation Point, including ontology based translation, conflict detection, XACML generation and interaction with the DHT for high level privacy rules.
- My contribution concerned a different set of components needed to make the new Privacy Dashboard operational and deployable in a real smart home setting.

1.3 Thesis Goals

More specifically, my contribution focused on five specific objectives:

1. **Integration with Smartotum, Authentication and Synchronization**

   A first objective was to connect the Privacy Dashboard with Smartotum so that the platform could operate on a real smart home. To access the homes associated with a Data Subject, the dashboard first needed to rely on Smartotum authentication. Once a secure connection was established, the platform also needed a synchronization mechanism to retrieve homes, applications, devices and rules from Smartotum and keep the local dashboard aligned. My goal was therefore to define this flow, from authentication to data retrieval and synchronization, so that privacy management could rely on real smart home information.

2. **Automatic Enforcement of Consent Changes**

   A second objective was to make consent changes effective in the smart home. From the user perspective, this operation is not limited to updating the consent status shown in the dashboard, but also has an effect on the underlying smart home behavior for applications whose consents are associated with Smartotum actions. In practice, the dashboard must align the consent related enforcement rules so that, if a consent is currently given, any corresponding deny rule is removed, while if a consent is not given and an action mapping is available, the corresponding deny rule is ensured for the selected home. My goal was therefore to implement the logic required to connect consent changes with the corresponding deny rules in Smartotum.

3. **Marketplace Proof of Concept**

   A third objective was to extend the platform with the Marketplace proof of concept. The Marketplace is meant to be an external system that can register applications into the Privacy Dashboard for a specific Data Controller, who is then linked to the application as its owner. Once the application is registered, the Data Controller can manage the application and handle all GDPR requests and documents. My goal was therefore to define the dedicated endpoints and the authentication mechanism used by this integration.

4. **Frontend Architecture and Shared State Management**

   A fourth objective was the frontend design of the Privacy Dashboard. The main challenge was to provide a unified and easy to use interface for users with different roles, while still allowing each of them to access different information and functionalities. Another important part of this work was the definition of dedicated contexts that enable the dashboard to keep user, home and application information consistent across different pages. My goal was therefore to define a frontend architecture able to support role based navigation and shared state management.

5. **Continuous Integration, Delivery and Deployment**

   A fifth objective was to support the project with a workflow able to validate and deploy the platform consistently. In the Privacy Dashboard we focus on the creation of a comprehensive end to end testing suite to validate the most critical user workflows from start to finish. On top of this, the pipeline is organized into four stages, build, test, publish and deploy, so that each validated version of the application can be released and propagated to the demo environment. My goal was therefore to define the testing, delivery and deployment workflow needed to move the system towards a usable demo platform.

1.4 Thesis Structure

The thesis is structured as a progression from the general context of smart home privacy to the implementation of the new Privacy Dashboard, with particular attention to the parts of the platform to which I contributed directly. To make this progression more concrete, the thesis uses a recurring example based on the Smartotum camera manager application to illustrate how a privacy decision expressed in the dashboard can be translated into a concrete effect in the smart home, and revisits it from different perspectives in Chapter 3, Chapter 6 and Chapter 8.

Chapter 2. *Background and Legacy System Analysis*

This chapter provides the background needed to motivate the role of the Privacy Dashboard in smart home environments. It first introduces the concept of the smart home, starting from the key role played by IoT devices within it, and then discusses the main threats to user privacy. It then presents SIFIS-Home as a framework aimed at addressing these concerns and Smartotum as a home automation system that brings this vision into practice. On this basis, the chapter introduces the GDPR concepts supported by the Privacy Dashboard, such as the actors involved, the required documents and the rights that users must be able to exercise. Finally, it analyzes the legacy Privacy Dashboard and its limitations, highlighting the need for a more complete platform.

Chapter 3. *The New Privacy Dashboard*

This chapter presents the new Privacy Dashboard as the evolution of the legacy platform introduced in the previous chapter. It first introduces the main user roles and the different perspectives from which Data Subjects, Data Controllers, and Data Protection Officers interact with the system. It then describes the main features, from Smartotum integration and automatic enforcement of consent

changes to policy translation and GDPR document handling. Finally, it outlines the functional requirements that serve as a reference for the design and implementation choices discussed in the following chapters.

Chapter 4. *System Architecture and Technologies*

This chapter presents the main architectural choices and technologies on which the Privacy Dashboard is built. It introduces the overall client server architecture of the system together with the main technologies adopted for database, backend and frontend development. In this way, it provides the technical basis for the more detailed design choices discussed in the following chapters.

Chapter 5. *Frontend Architecture and Shared State Management*

This chapter focuses on the frontend design of the Privacy Dashboard, that is one of the main parts of the project to which I contributed directly. It first introduces the main design objectives, especially the need to support different user roles while keeping the interface clear and easy to use. It then presents the frontend architecture adopted, together with the React mechanisms used to keep information consistent across different pages of the dashboard. Finally, it describes how the Privacy Dashboard manages shared information through dedicated contexts for authentication, homes and applications.

Chapter 6. *Integrating Smartotum*

This chapter is centered on the integration of the Smartotum home automation system, which is another of my main contributions to the project. It first introduces the practical challenges raised by this integration, especially the need to support Smartotum authentication and to interact with the smart home of the user. Finally, it describes how the Privacy Dashboard retrieves smart home information, including device level rules, from Smartotum and synchronizes this data with the platform.

Chapter 7. *Marketplace Proof of Concept*

This chapter focuses on the Marketplace proof of concept, which is another of my main contributions to the project. It first introduces the Marketplace as an external integration through which applications can be registered in the Privacy Dashboard for specific Data Controllers. It then presents the API and authentication mechanism adopted for this integration. Finally, it describes the main use cases of this integration and its current limitations.

Chapter 8. *Continuous Integration, Delivery and Deployment*

This chapter focuses on the CI/CD workflow of the Privacy Dashboard, which is another of my main contributions to the project. It first introduces the concepts of continuous integration, delivery, deployment and testing, together with the testing

approach adopted. It then presents the pipeline architecture and its stages used to build, validate, publish and deploy the application. Finally, it describes the server architecture that supports this workflow and hosts the demo deployment of the platform.

Chapter 9. *Conclusion*

The final chapter summarizes the outcome of the project, discussing the degree to which the original goals have been fulfilled. It then discusses the main limitations that still affect the current system and outlines the future developments through which the platform could evolve into a more complete solution for privacy management in smart home systems.

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, actuator, alarms and many more, that are characterized by the ability of exchanging information and commands with each other and with external services connected to the Internet.

IoT systems offer many advantages, such as enabling data-driven decision making, since actions can be based on the data that devices continuously collect and share, and increasing automation, for example by allowing actuators to perform autonomous physical actions when some particular changes are detected in the environment.

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 connected IoT devices all over the world will more than double by 2030, increasing from 18.5 billion in 2024 to 39 billion in 2030 [2]. 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 them 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 allowing home owners to easily monitor and rule their Smart Home from a centralized software accessible from the Internet. Such kind of system can be implemented using a three-layer architecture (Figure 2.1): the perception layer, the network layer, and the application layer [3].

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.

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 [4]. For instance, when the homeowner leaves, the Smart Home can automatically turn off lights and enable security alerts if unusual activity is detected. It can also monitor the owner's location to detect their return and, for example, open the garage door to provide quick access to the home. 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.

This approach can raises privacy concern, because data are processed and stored by third parties that homeowners do not directly control. In fact, 52% of Smart Home users are concerned about the possibility that someone could control their smart home devices [5].

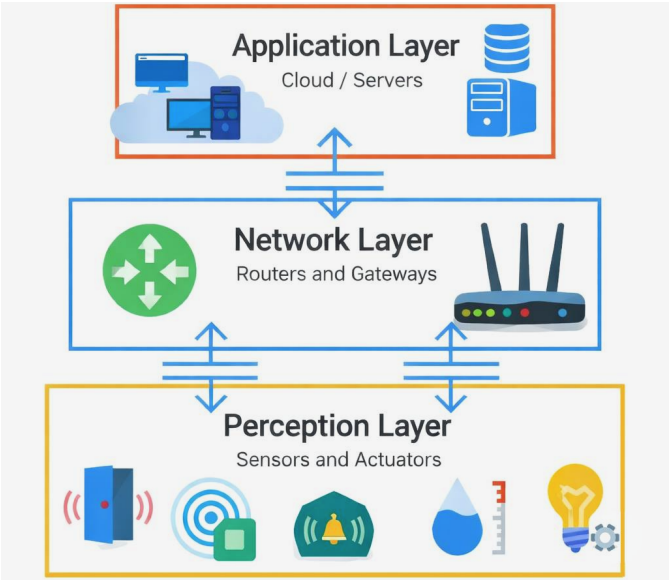


Figure 2.1: The three layer model of IoT architecture.

2.1.2 Smart Homes Privacy Threats

The increasing capabilities of IoT devices brings many advantages but unfortunately various drawbacks as well, among which the concern of user's privacy: 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 defined as follows [6]:

- **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.

Moreover, ENISA has observed that IoT devices are becoming victims of more frequent DDoS (Distributed Denial of Service) attacks because of their limited resources that often result in poor security. In this context, DDoS aims to threaten the availability of components as well as to disrupt the operation of other networks or systems but they also have the potential to threaten the safety of users.

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 [7].

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 [8].

2.1.3 SIFIS-Home and Smartotum

The SIFIS-Home Project was created to address these concerns by developing a secure-by-design and consistent software framework across all stack levels [9].

SIFIS-Home

From a design standpoint the SIFIS-Home Framework is organized into dedicated frameworks (Smart Device, NSSD, Application, Cloud, Development Tools), each comprising the set of software components executed on a specific platform. [10]

The overall architecture follows a microservices approach based on Docker containers: most of the components can expose a REST 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 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 [10].

Smartotum

Smartotum originated as a technology transfer initiative that brings the SIFIS-Home vision into a deployable home automation system.[11]

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.[11]

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 user privacy.

2.1.4 GDPR

In May 2018, 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:

> 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:

> '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, whereas its core functions are described in Art. 39 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, 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 is granted the following rights when using a product or a service that is collecting their data:

1. **Right of access** (Art. 15): 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): 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): 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): The data subject shall have the right to obtain from the controller restriction of processing [...]

5. **Right to portability** (Art. 20): 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): 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 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. These requirements are also reflected in two important compliance documents under the GDPR.

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 [12]. 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 Another important document, that is addressed by the GDPR in Art. 35 [13] and also important in the context of the Privacy Dashboard 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. 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.

These two documents play a key role in GDPR compliance, for this reason, their management is one of the main features of the Privacy Dashboard.

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 we took was to analyze its functionalities, and address its limitations, which we recap in this section.

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 [14]. 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, one of the main characteristics of the Privacy Dashboard is that both the features and the user interface are differentiated according to the role of the logged in user (Data Subjects, Data Controllers, 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.

- **Applications page**: The core of the Privacy Dashboard, as it contains a list of installed or managed applications, with different details and functionalities to each one, 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.

- **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 see 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 don't 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.

- **Rights page**: Another key page that 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.

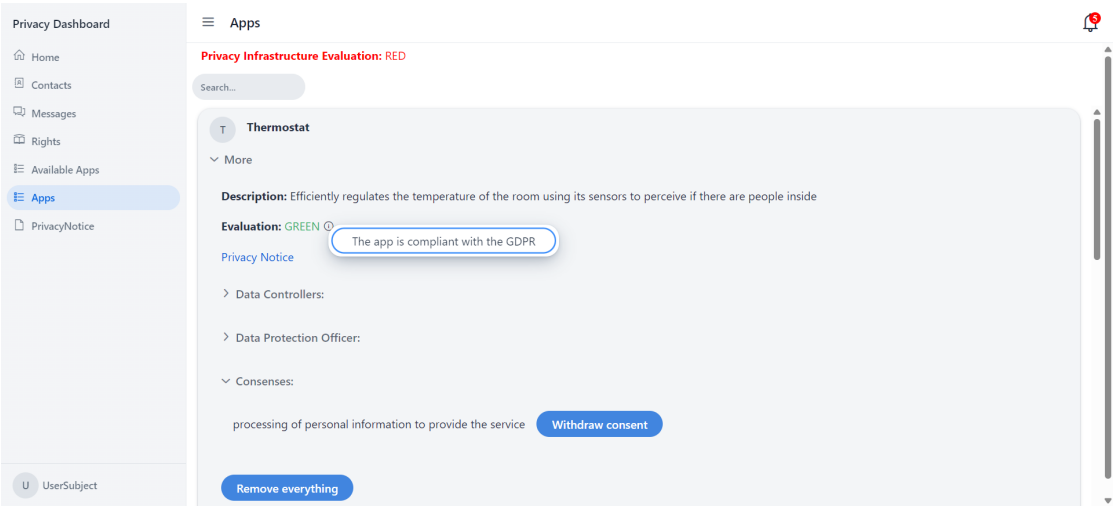


Figure 2.2: Applications Page for Data Subjects.

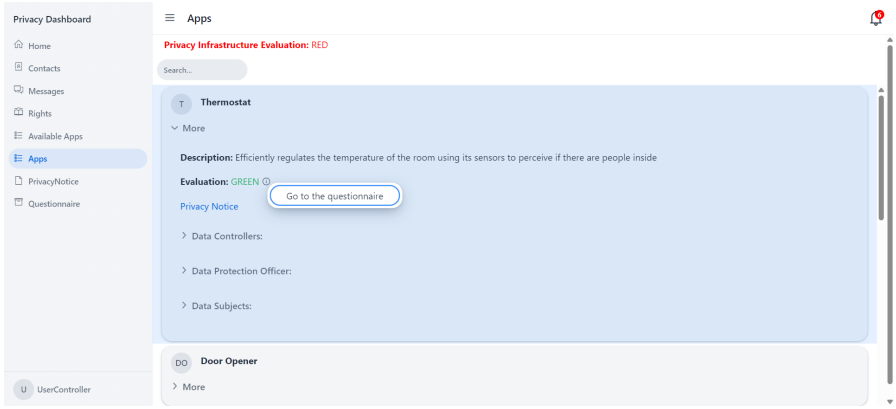


Figure 2.3: Applications Page for Data Controllers and Data Protection Officers.

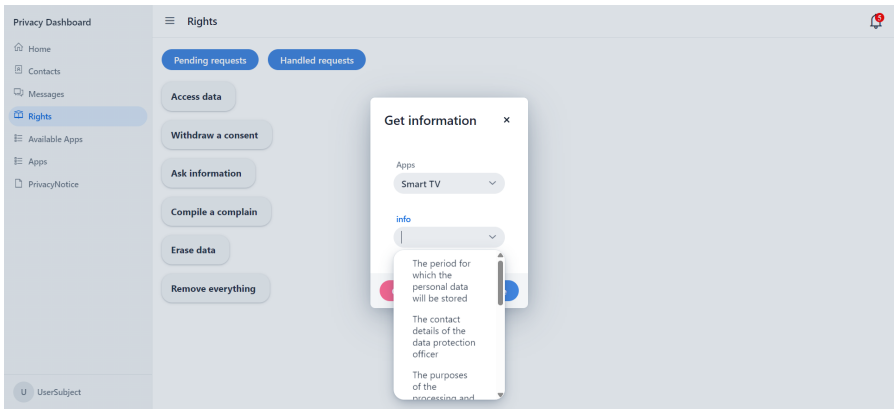


Figure 2.4: Rights Page for Data Subjects.

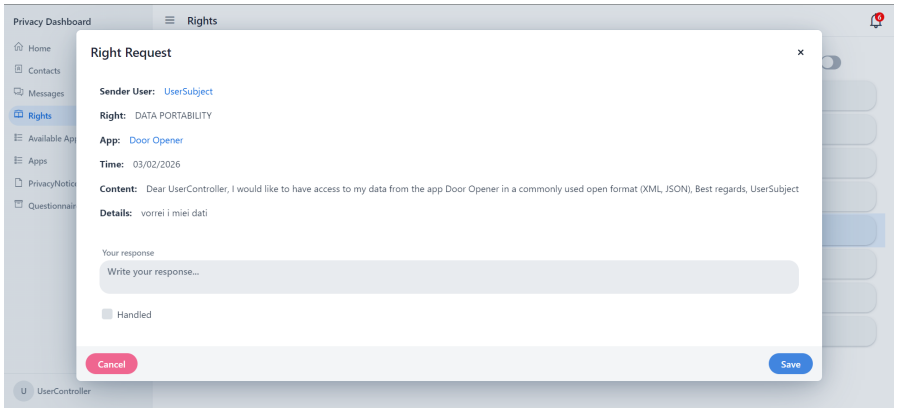


Figure 2.5: Rights Page for Data Controllers and Data Protection Officers.

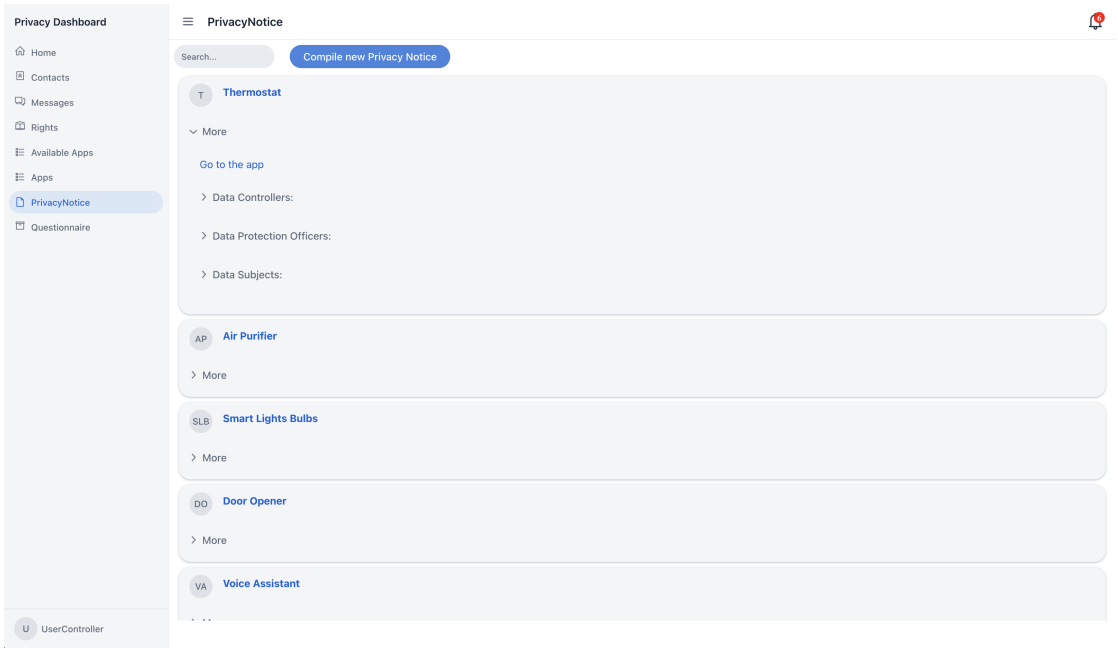


Figure 2.6: Privacy Notice Page for Data Controllers and Data Protection Officers.

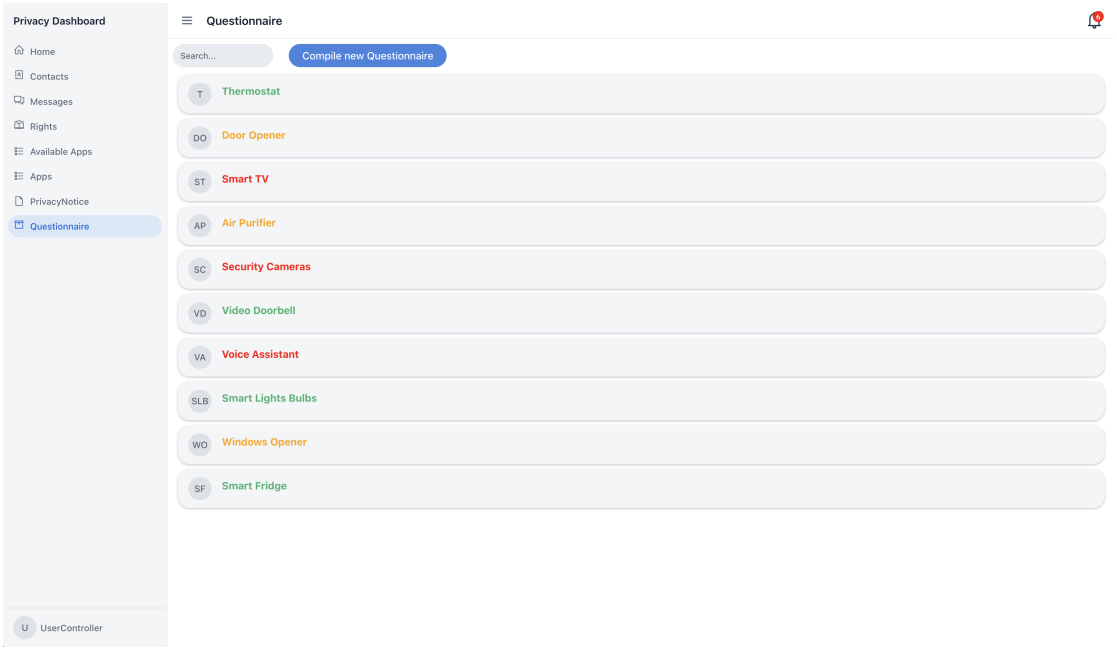


Figure 2.7: Questionnaire Page for Data Controllers and Data Protection Officers.

2.2.2 Limitations of the Existing Privacy Dashboard

One of the main limitations of the existing Privacy Dashboard is the absence of interaction with a real home automation system, which means that, for a Data Subject, the association between the applications shown in the dashboard and those installed in their personal Smart Home is missing. As a result, privacy related actions remain detached from the underlying domestic environment, in particular, consent changes and high level privacy rules cannot produce concrete effects on real devices. Support for downloading files containing Privacy Notices and stored personal data was also planned but not implemented. Finally, the creation and management of another important document related to GDPR compliance, the Privacy Impact Assessment, is missing.

These limitations were acknowledged by Filippo Peron in the conclusion of his thesis paper [14] 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 functionality 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 analysing 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 3.1.

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 3.1: Main frontend functionalities available to each role.

3.2 Main Features

3.2.1 Smartotum Integration

The main focus was the integration of the Smartotum Home Automation System, enabling 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 is retrieved 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

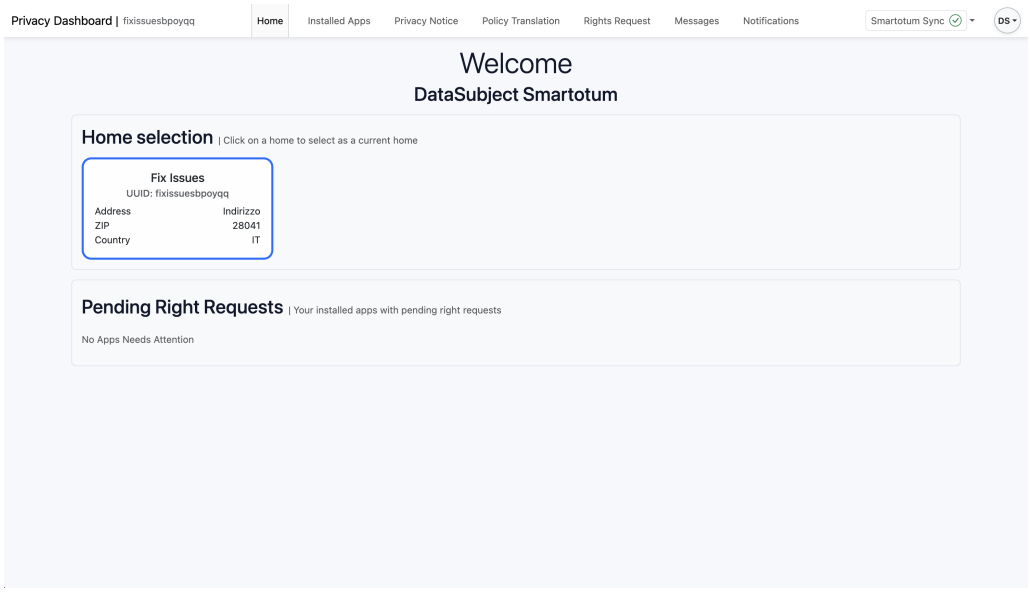


Figure 3.1: Privacy Dashboard Homepage for Data Subjects

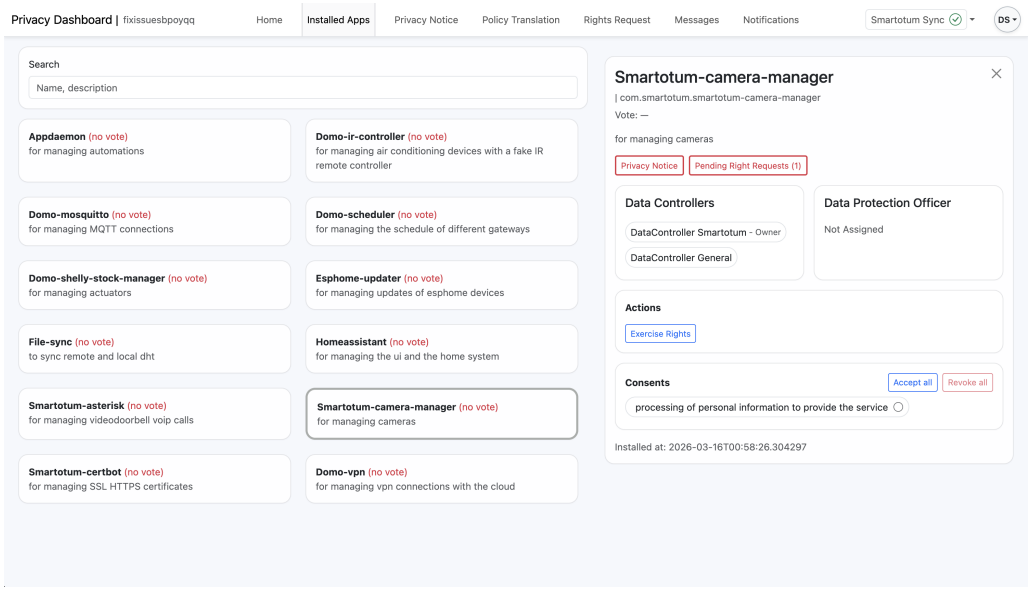


Figure 3.2: Privacy Dashboard Installed Applications page for Data Subjects

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 the camera manager application, this means 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. The technical details of this mechanism are discussed later in Chapter 6.

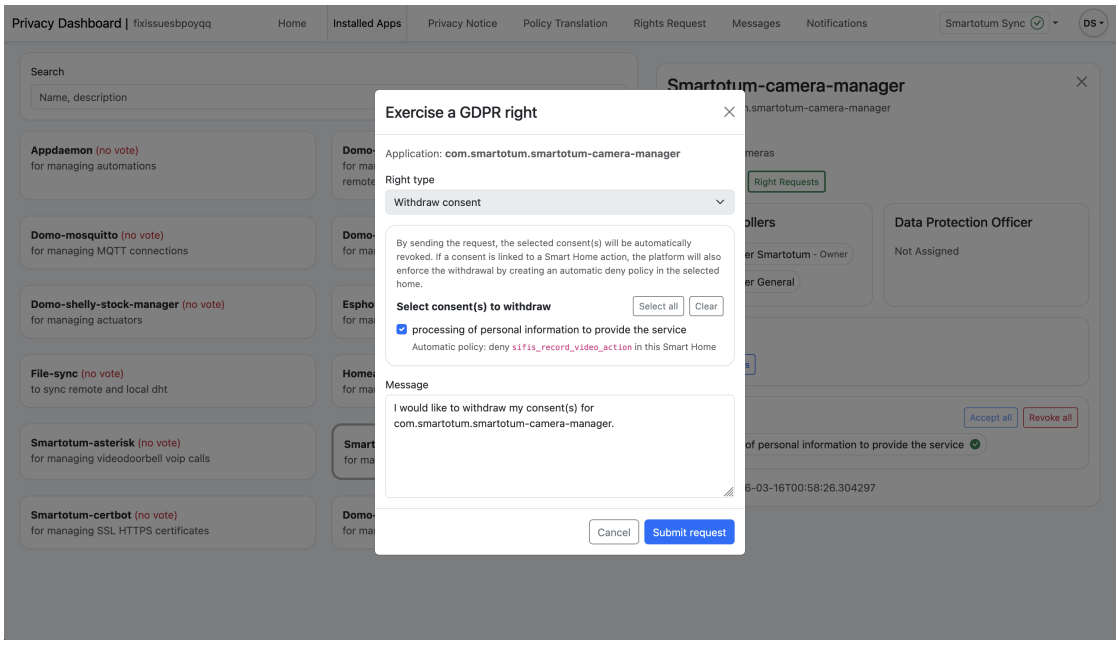


Figure 3.3: Example of a withdraw consent

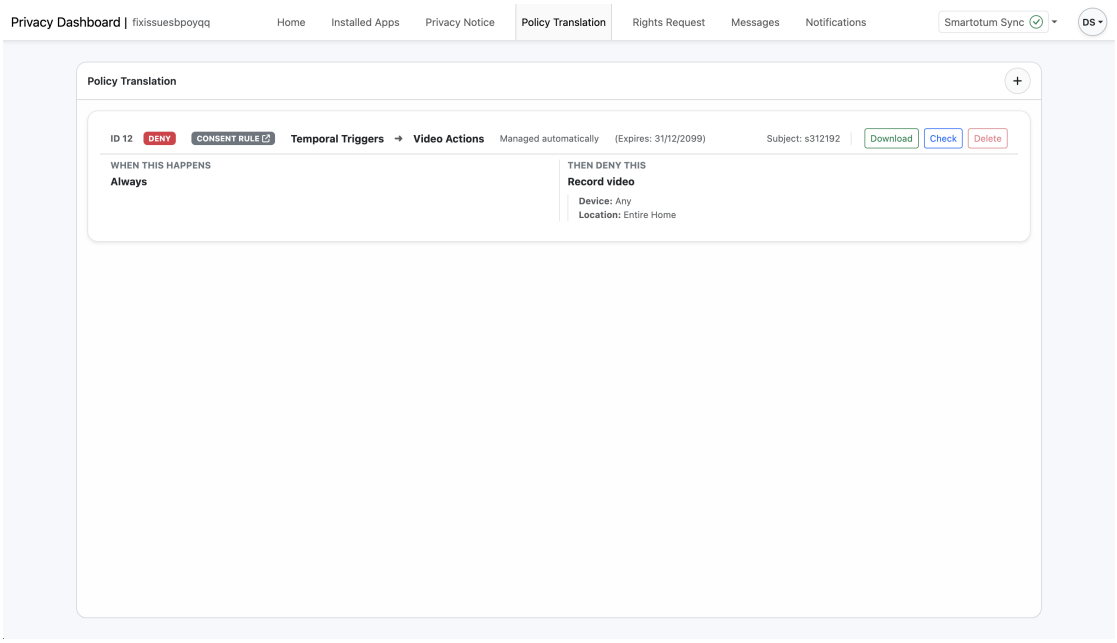


Figure 3.4: Example of a rule denying permission to record video

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.

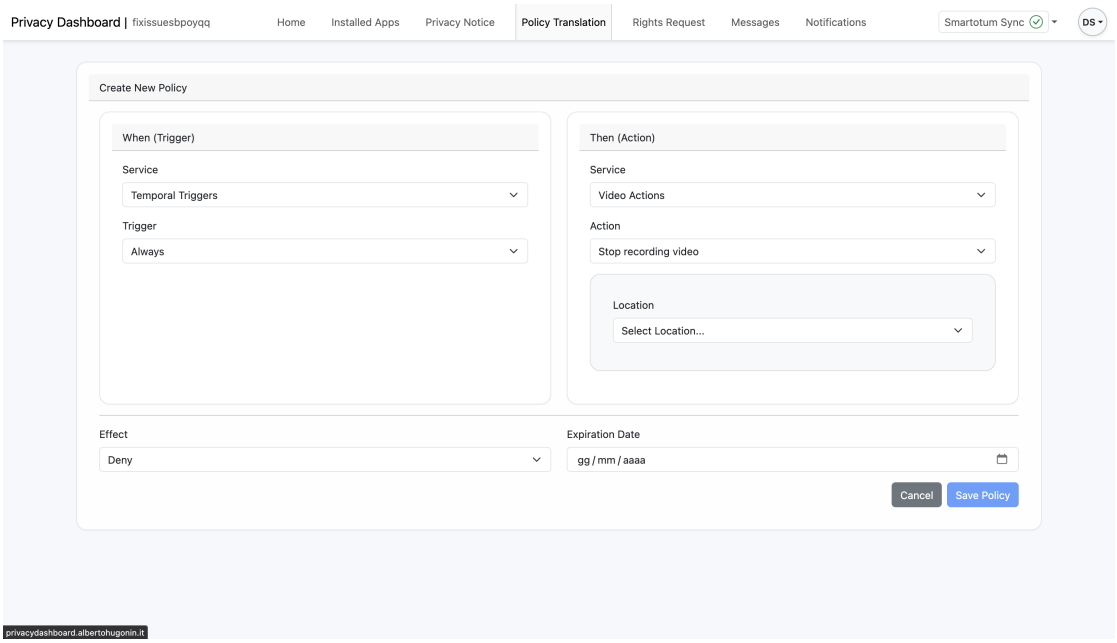


Figure 3.5: Privacy Policy Rule Creation Interface

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. 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. While not logically dangerous, highlighting redundancies helps users maintain a clean and performant 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.

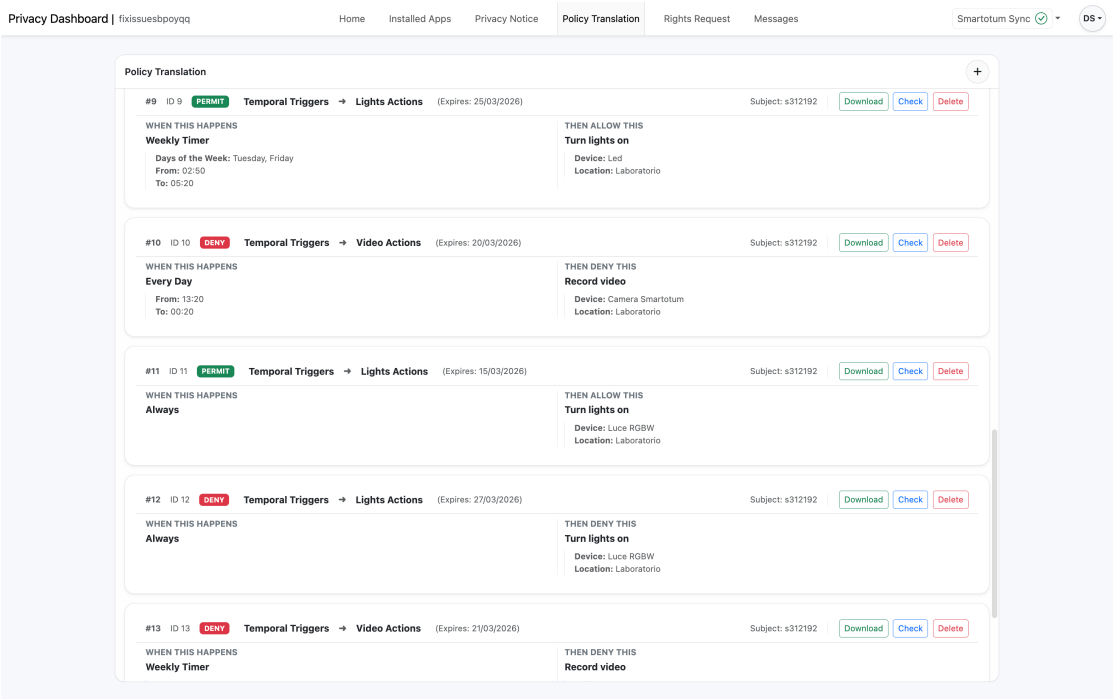


Figure 3.6: Overview of created high-level policies with action buttons.

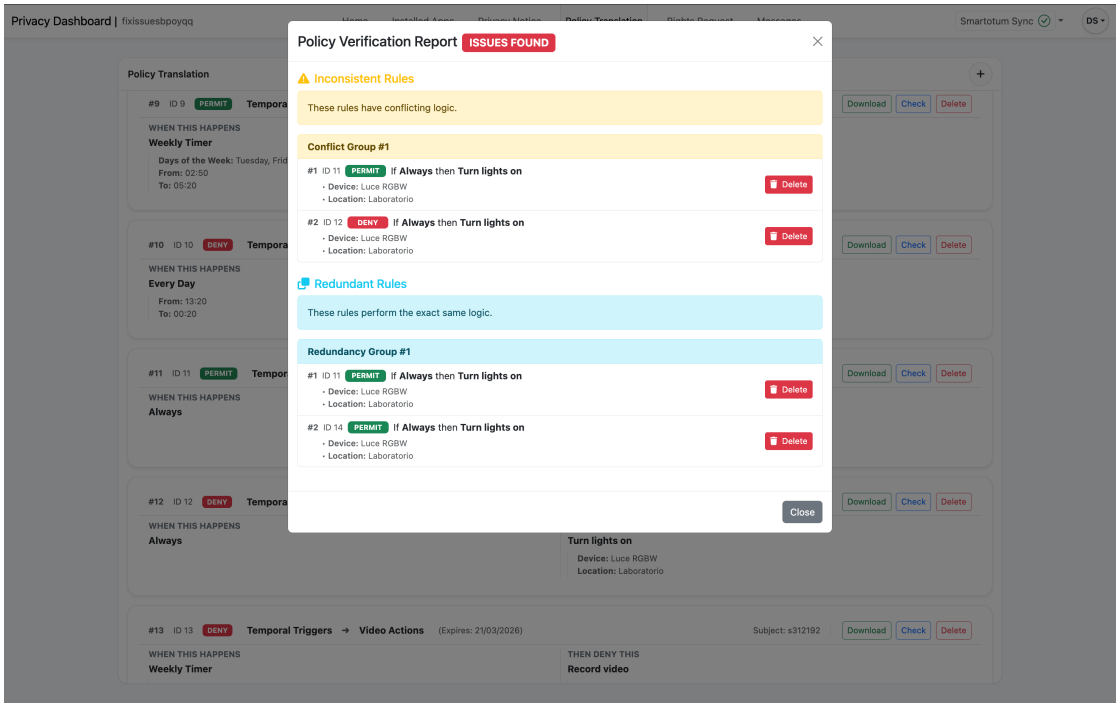


Figure 3.7: Policy verification report displaying categorized conflicts.

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 authorised 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, 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 required by the application.

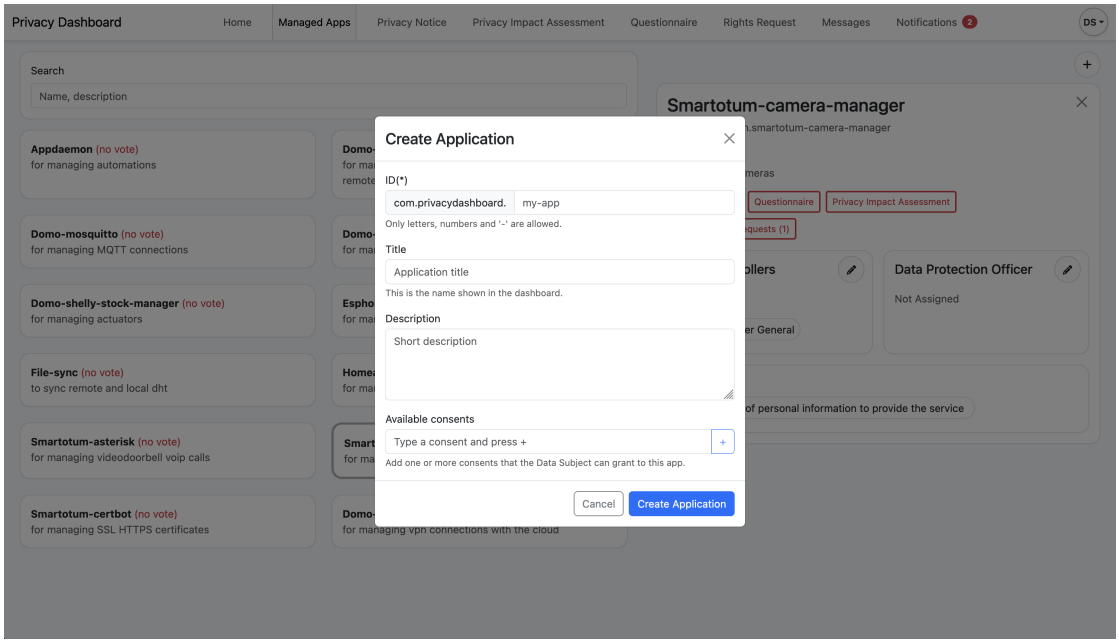


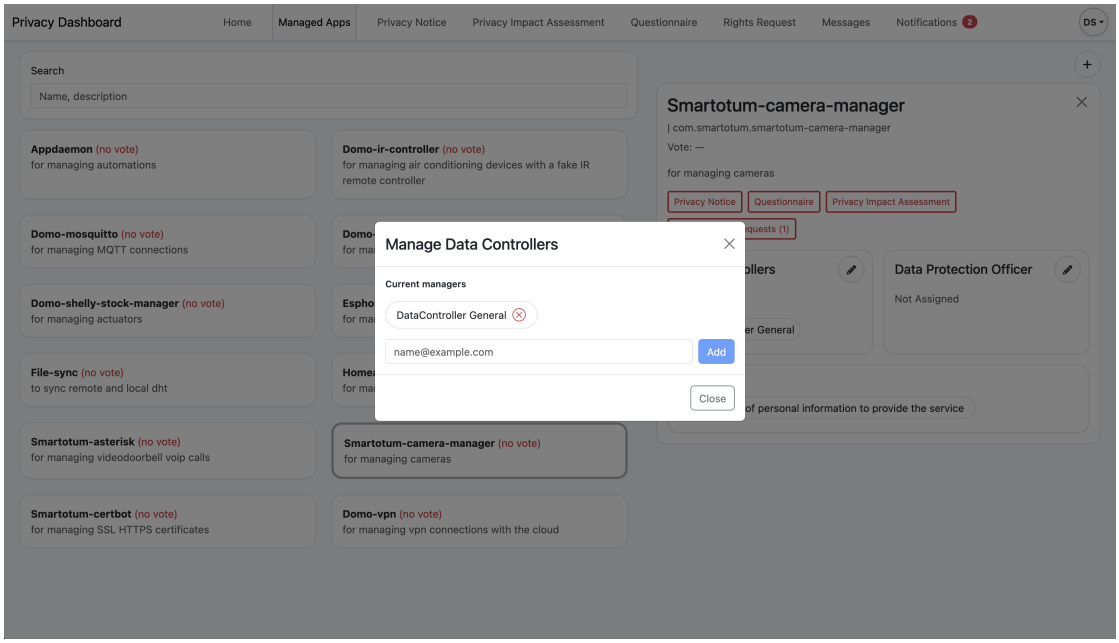
Figure 3.8: Privacy Dashboard Create Application modal for Data Controllers

3.2.6 Role Assignment

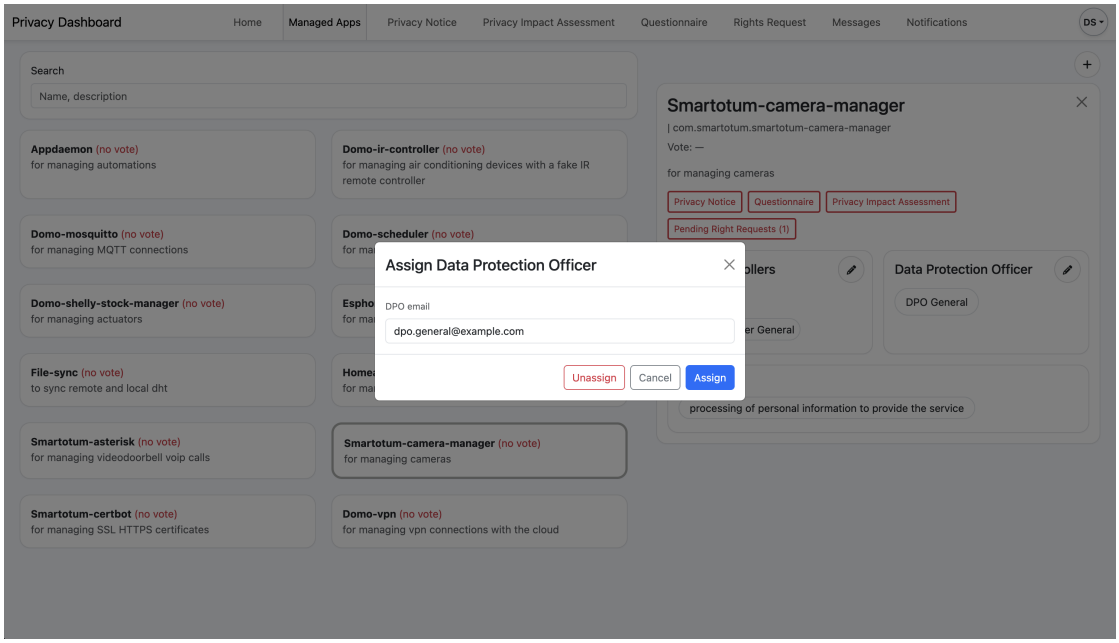
By supporting different sources of applications, a role assignment process is needed to ensure that every application is linked to the correct Data Controller and, when applicable, a Data Protection Officer, so that the Privacy Dashboard can act as an all-in-one platform for managing privacy in smart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

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 consent 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 3.9: Role Management

3.2.7 Available Consents Management

Available consents can also be added and removed by the owner of the application following its creation as shown in Figure 3.10.

Because Smartotum is the source of truth for the 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 applications and Marketplace ones.

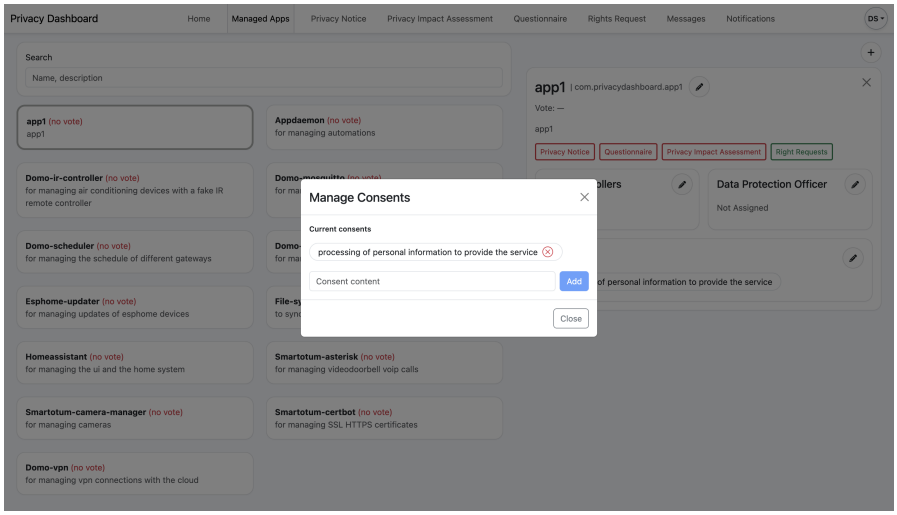


Figure 3.10: Privacy Dashboard manage available consents for Data Controllers

3.2.8 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 3.11 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.

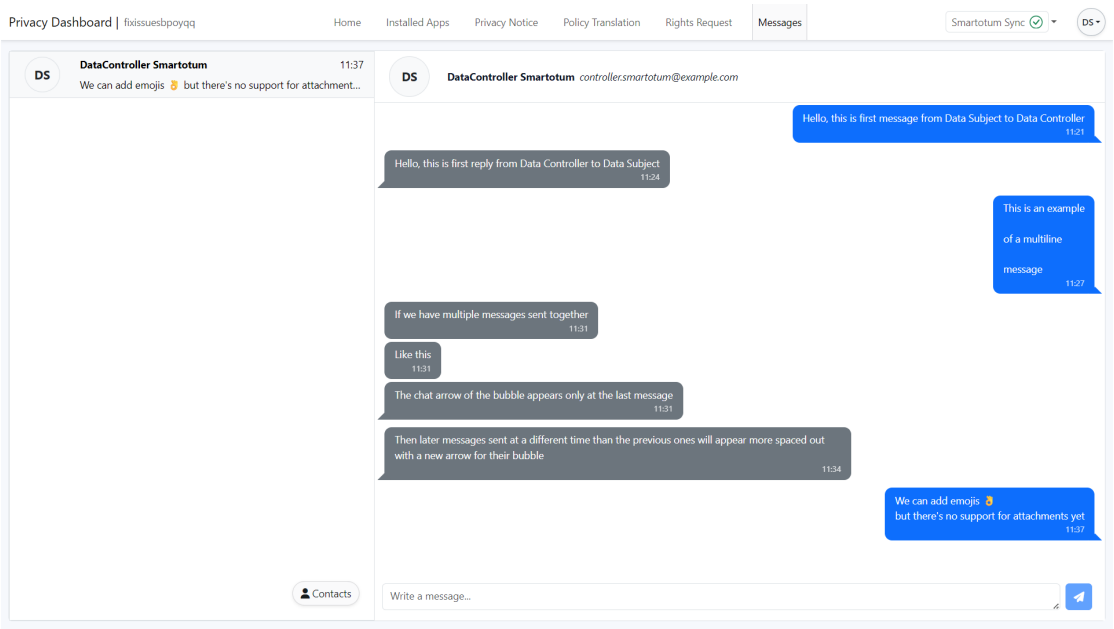


Figure 3.11: Messages and contacts page.

3.2.9 GDPR Document Management

Privacy Notice

Figure 3.14 shows the interface for managing Privacy Notices. 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.

Privacy Impact Assessment

As described in Section 2.1.5, the *Privacy Impact Assessment* is an important document for assessing risks related to personal data processing. For this reason, building on the limitations discussed in Section 2.2.2, the Privacy Dashboard provides a dedicated page for managing it. As shown in Figure 3.12, 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 instead, the document can be viewed when available and, using the toolbar, users can create, edit, delete or download it as a PDF.

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 3.15 shows our user interface to manage it. 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.

GDPR Documents Download

For all GDPR documents, a download feature is available that generates a PDF ready to be downloaded. Examples of downloaded GDPR documents are shown in Figure 3.16.

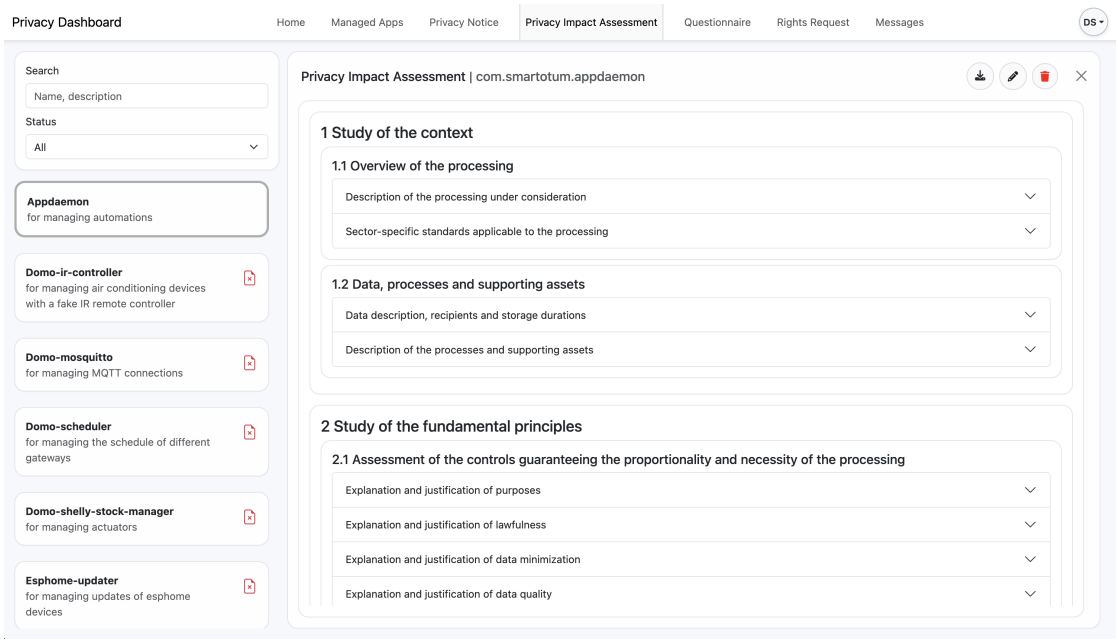


Figure 3.12: New Privacy Impact Assessment editor for Data Controllers and DPO

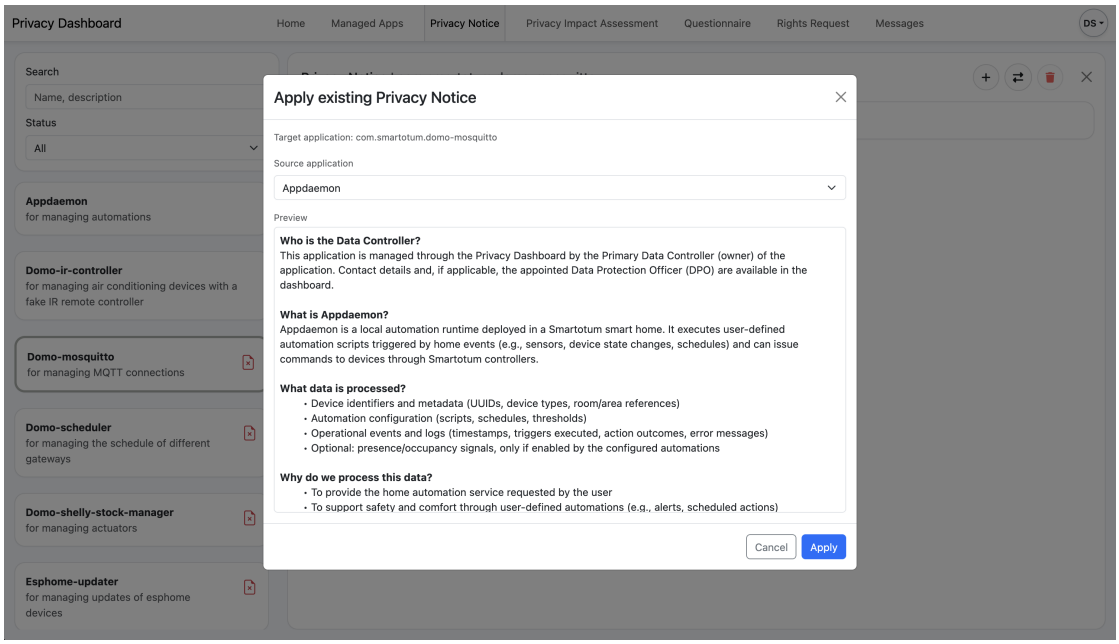


Figure 3.13: Apply Existing Privacy Notice

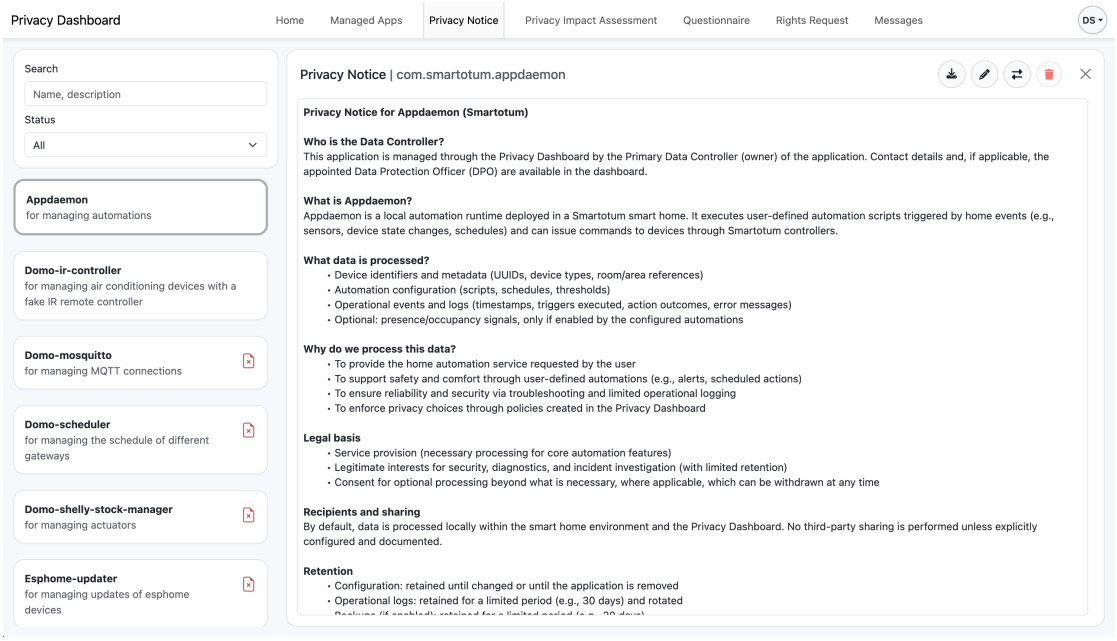


Figure 3.14: New Privacy Notice editor for Data Controllers and DPO

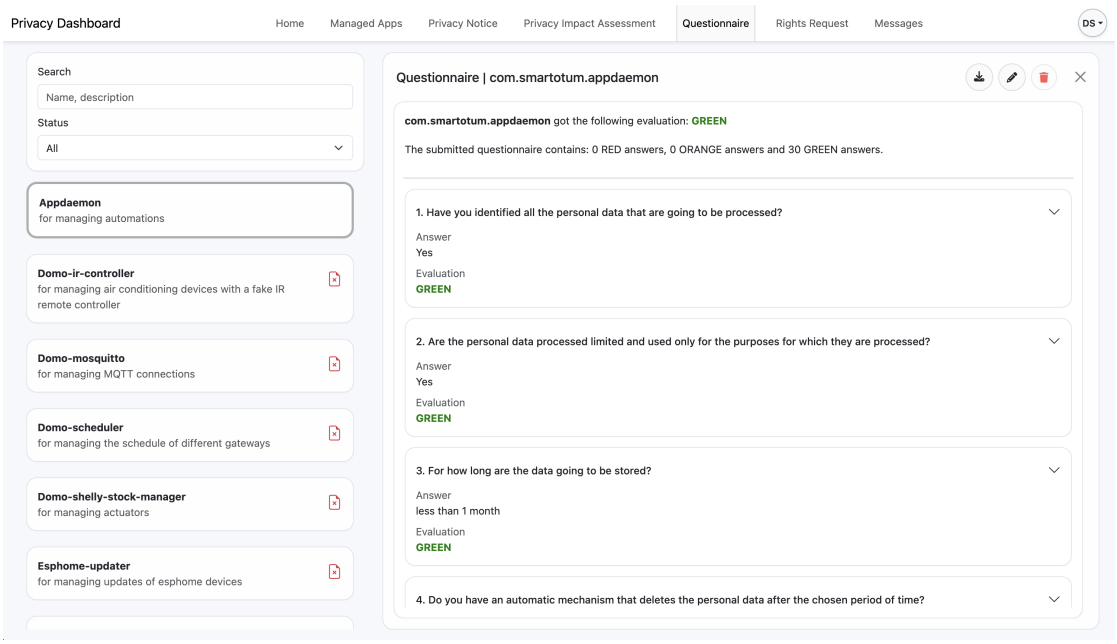


Figure 3.15: New Questionnaire editor for Data Controllers and DPO

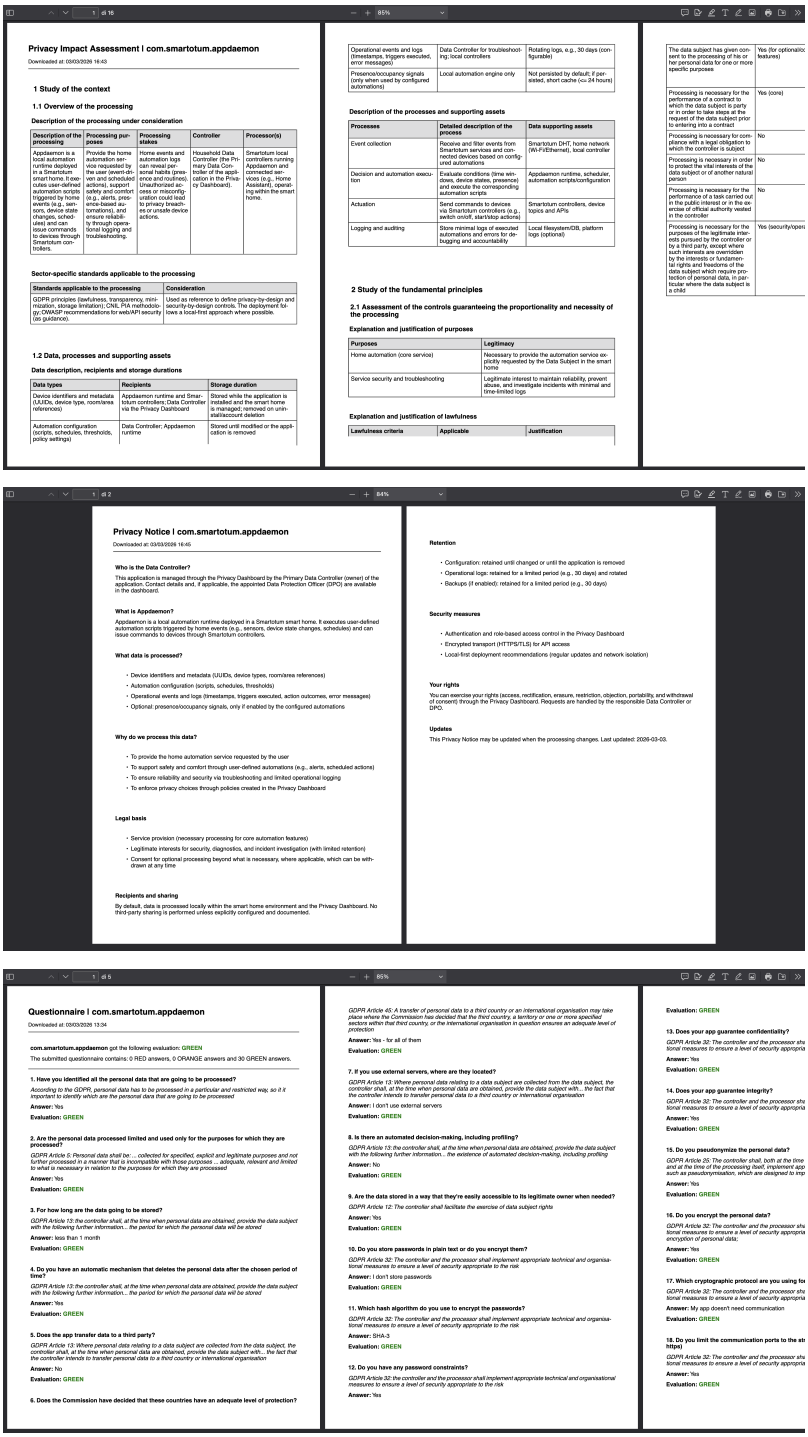


Figure 3.16: Examples of generated documents (Privacy Impact Assessment, Privacy Notice and Questionnaire).

3.2.10 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 DPO receive a notification for accounting purposes. Similarly, a notification is also received by a Data Controller and DPO 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 3.17.

Audit Preservation The main function of the Notification History page, besides its name, 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 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 DPO.

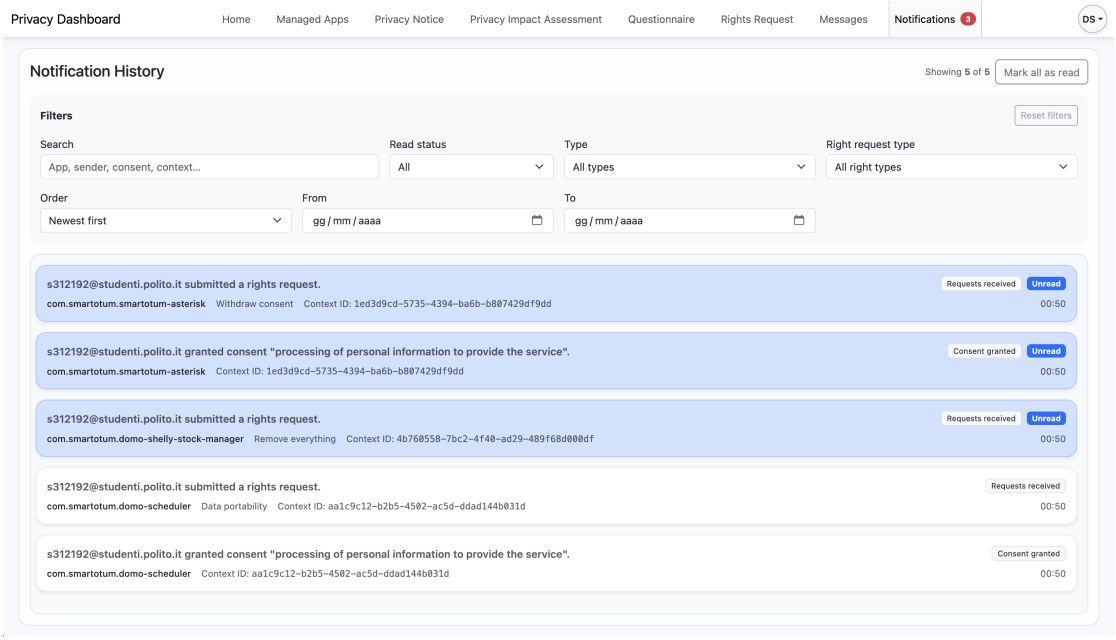


Figure 3.17: Notification History page

3.3 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.
Following the functional requirements, the architectural diagram of the Privacy Dashboard is also presented, highlighting the most important parts of the system and the external actors involved.

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**

ID	Name
FR3.1	Enable the user to send and receive messages to and from any contact with a dedicated interface.
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**

ID	Name
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.
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.

ID	Name
FR6.2	Enable Data Controller and DPO to edit an existing privacy notice associated to an application.
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.

ID	Name
FR9	**Manage Notifications**
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.

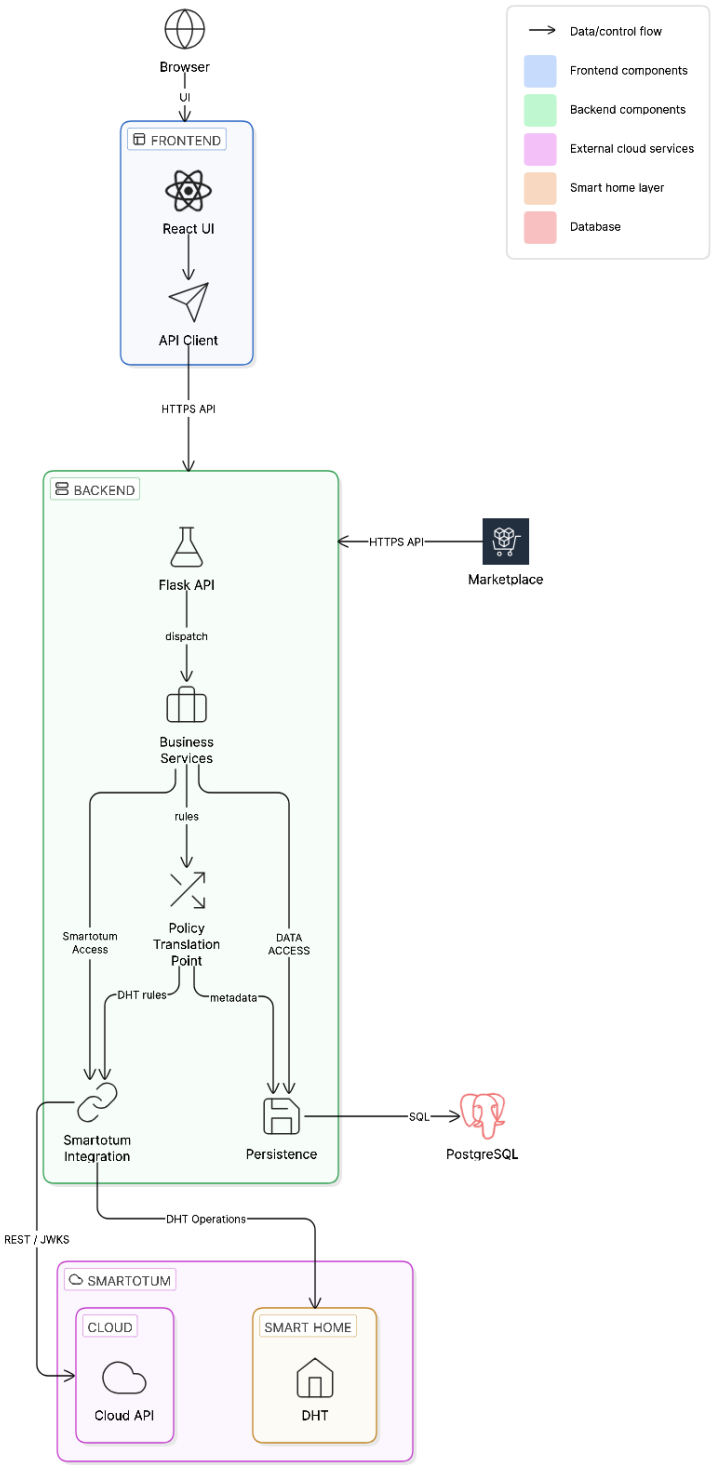


Figure 3.18: Privacy Dashboard System Architecture

Chapter 4

System Architecture and Technologies

4.1 Client and Server Architecture

The Privacy Dashboard can be considered a *network-based application*. One of the most common architectural model for *network-based applications* is the *client–server model*, which establishes a clear separation of concerns between two main components: the *client*, which acts as the service consumer, and the *server*, which acts as the service provider. The server exposes a set of services and listens for incoming requests, while the client sends requests through a connector when it needs one of those services. The server then either rejects the request or processes it and returns a response to the client [15], as shown in Figure 4.1.

In web applications, the client is often referred to as the *frontend* and, as the name suggests, represents the first point of interaction with the user, whereas the server is often called the *backend*. This separation of concerns 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 [15].

The communication between frontend and backend usually happens over the Internet and follows a predefined message exchange protocol, the HTTP (Hypertext Transfer Protocol). Fielding et al. formally defined HTTP as [16]:

> 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.

HTTP requests contain different fields, among which we find:

- **URL (Uniform Resource Locator)**: used to identify the resource in the server we want to point to
- **Method**: which describes the type of action we want the server to perform (*e.g.* read or write some desired data);
- **Body**: an optional field for requests that write or update data stored in the server.

HTTP responses also have a specific structure, among which we find:

- **Status code**: It informs the client on the outcome of its request (*e.g.* whether it has failed or completed successfully);
- **Body**: Contains the data requested by the client.

An example of an HTTP request and response is shown 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. 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 [15].

The following sections describe the architectural patterns and technologies adopted in the Privacy Dashboard, describing how data is stored in the database, processed by the server, exchanged via HTTP and consumed by the client.

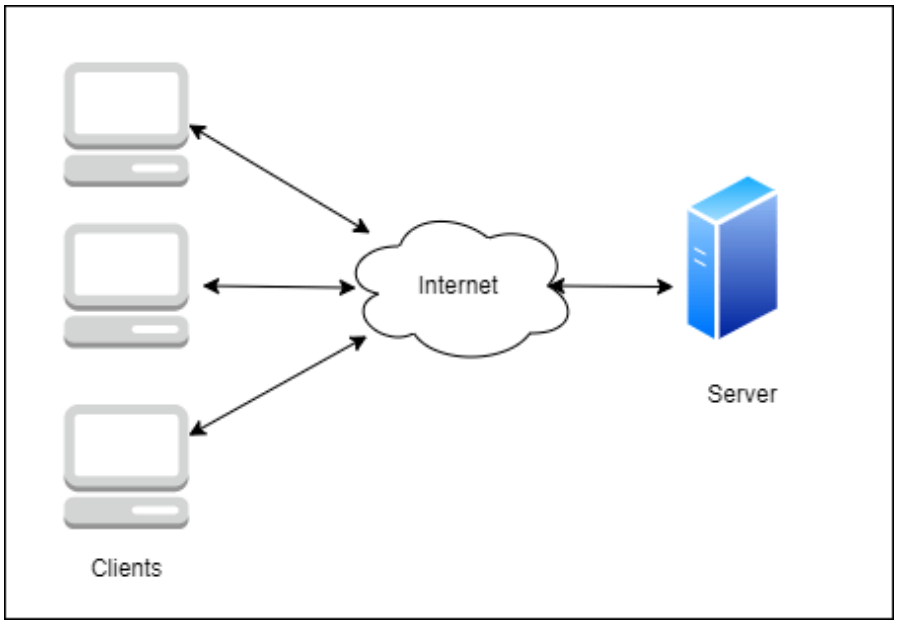


Figure 4.1: The Client-Server Architecture.

Figure 4.2: Example of an HTTP request and response.

4.2 Database Technologies

4.2.1 The Relational Data Model

Storing data that 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 [17]. 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 [17]. 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 [17]. 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 [17] 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 [18]. 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)*. When these operations involve multiple related changes, they are often grouped into transactions.

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 [19]:

- **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 a 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 fully compliant to 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 [20], PostgreSQL has now become one of the world's most used DBMS, according to Stack Overflow's 2025 Developer Survey. Among the ACID properties, atomicity is especially important in our case, as the synchronization between Smartotum and the Privacy Dashboard performs multiple updates that must either complete entirely or be rolled back to avoid any inconsistency, as discussed in Section 6.6.

4.3 Backend Technologies

4.3.1 Monolithic Architecture and Layered Backend Design

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 [21]. 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 [21], 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 [15]. 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 [22]. 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, Flask has become one of the most popular Python web application frameworks. As a "microframework" [23], 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. 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. This, in turn, helps to increase the levels of interactivity and responsiveness [24].

In order to dynamically update its components, the SPA does not need to receive HTML content, but rather some structured information and a fitting example of this required data is JSON objects. 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.

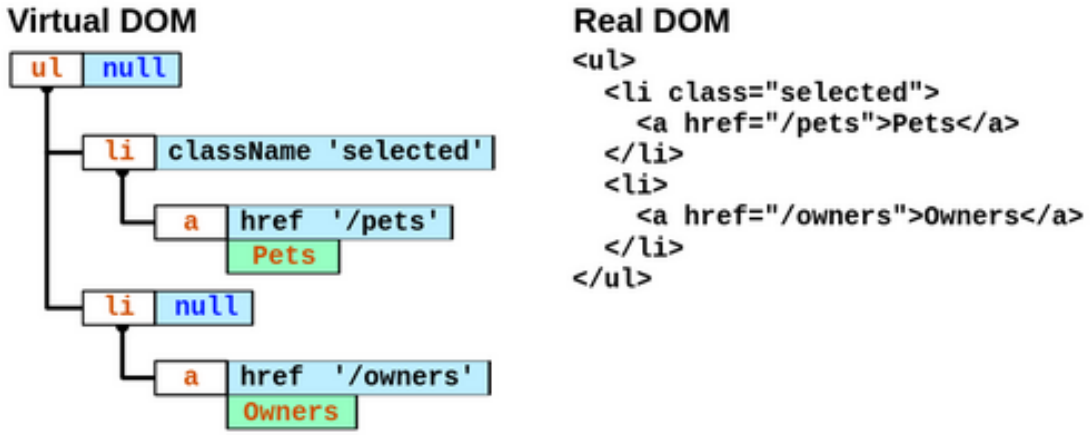


Figure 4.4: Example of virtual DOM compared to browser's DOM.

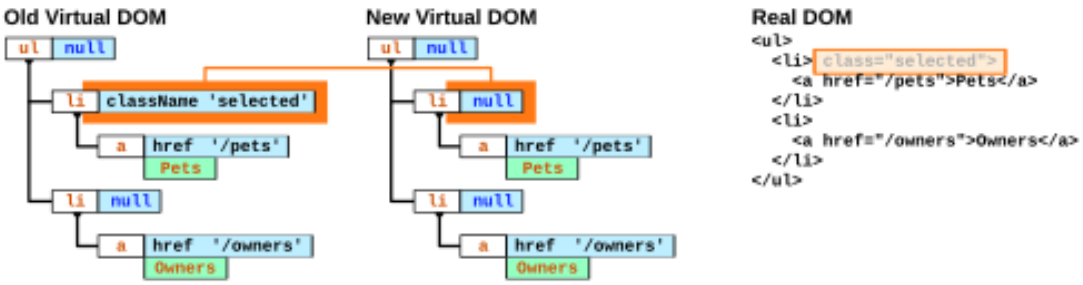


Figure 4.5: Example of update to virtual DOM reflected to browser's DOM.

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 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 [25]. 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 [26]. 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.

Chapter 5

Frontend Architecture and Shared State Management

5.1 Design Objectives

The main challenge in designing the Privacy Dashboard frontend was to provide a unified and easy-to-use interface for users with different roles, while still allowing each of them to access different information and functionalities.

A clear difference between roles can be seen by comparing the Data Subject and the Data Controller. The Data Subject interacts with the dashboard from the perspective of a specific smart home and its installed applications, while the Data Controller uses it from the perspective of the applications they manage and the related GDPR documentation. While the Data Subject can only view some documents, such as privacy notices and questionnaire evaluations, the Data Controller can view and edit all GDPR-related documents for applications under their responsibility.

Another important distinction is between the Data Controller designated as the application owner in the dashboard and the other Data Controllers delegated to manage the same application. Since some actions, such as assignment management and local application editing, are restricted to the owner, the frontend had to support this distinction as well.

As a result, the frontend had to support not only different pages, but also different available actions. A summary of the functionalities available for the different roles is shown in Table 3.1.

Another effort during the design of the Privacy Dashboard was made not only to make the frontend modular and maintainable by separating UI components, but also by creating dedicated contexts that enable the dashboard to keep user, home, and application information consistent across different pages.

5.2 Frontend Architecture

Building on the objectives presented in the previous section, the frontend architecture was also designed according to the principle of separation of concerns, similarly to the backend discussed in Section 4.3.1.

While in the backend the Controller–Service–Repository pattern separates request handling, business logic, and database access, the frontend follows a similar principle. In the frontend, however, this separation is more naturally presented from the bottom up, because pages and components depend on lower layers to obtain and prepare backend data.

5.2.1 API layer

The API layer is the lowest level of the frontend and it has the only responsibility of interacting with the REST endpoints exposed by the Privacy Dashboard backend. For this reason, it contains the logic required to communicate with the backend through HTTP calls and to manage details such as HTTP methods, URL construction, credentials handling, and raw response parsing.

5.2.2 Service layer

On top of the API layer, the frontend defines a service layer. While the backend service layer is responsible for business logic, the frontend service layer is responsible for transforming backend responses into objects that better support frontend logic and presentation.

More specifically, it takes backend responses, validates their expected structure, and converts them into objects that are easier to use in the frontend, such as `UserModel`, `AppModel` and `HomeModel`. If an error occurs, it propagates it to the upper layers so that it can be handled and displayed in the user interface.

This prevents React components from depending directly on raw JSON payloads and keeps this logic outside the presentation layer.

5.2.3 Shared State Layer

Between the Presentation Layer and the Service Layer, a further layer is needed to keep information consistent and to provide it across multiple pages and components while the user navigates through the application. This role is fulfilled by the shared state layer, which is discussed extensively in the next subsection.

5.2.4 Presentation layer

At the highest level, the frontend is composed of pages and reusable UI components, which are presented to the user differently depending on their role. Pages represent the main functional areas of the dashboard, such as installed or managed applications, privacy notices, questionnaires, policy translation, messages, and right requests. Reusable components are then used to implement more specific interface elements inside these pages, such as forms, detail panels, lists, and editors. This organization makes it possible to reuse components across multiple pages while avoiding duplication of interface logic.

5.3 From Local State to Shared State

As anticipated, between the Presentation Layer and the Service Layer the frontend requires a further layer capable of keeping information consistent across multiple pages while the user navigates through the application.

Before discussing the specific implementation adopted for the shared state layer in the Privacy Dashboard, it is useful to introduce some React concepts that support this design.

5.3.1 React States

In React, the memory of a component is represented by its *state*, which is used to store information whose changes affect what must be rendered. State is used to store information that survives component re-rendering, whereas ordinary JavaScript variables would be recreated at each render.

React provides a built-in way to declare and update state through the `useState` Hook. This Hook allows a component to define a state variable together with the function used to update it, and React re-renders the component whenever that state changes.

A simple example is the content of a form field, which can be stored in state through `useState`, as shown in Listing 5.1. Whenever the user types in the field, the state is updated through the corresponding setter function, and the new value is reflected in the user interface.

```
const [field, setField] = useState("hello");
```

Listing 5.1: Example of state declaration with `useState`

5.3.2 React Hooks

As anticipated `useState` is one of the React functions called *Hooks*. More generally, Hooks are functions that let components use React features.

When the same logic used to store, load, and update information is needed in different parts of the application, it is convenient to extract it into a custom hook instead of duplicating it. A custom hook does not automatically make two components share the same state: it only reuses the same logic inside each component that calls it. In the Privacy Dashboard a custom hook is `useRightsRequests`, which encapsulates both the current list of requests and the logic required to load and update it.

5.3.3 React Context

There are different ways to share information across multiple parts of a React application:

- **Props**: used to share data from parent to child components but it is inconvenient when it must cross many intermediate components that do not use it directly, this is referred to as `Prop Drilling`
- **URL**: used to encode part of the information in the URL, which is useful when that information is used for navigation inside the component, but limited information can be represented in this way.
- **Context**: used to make a value available to an entire group of descendant components without passing it manually.

When a Context is created, it is also created a component called `Provider` that stores the shared information and, as the name suggests, provides it to other components that are nested inside it, as shown in Listing 5.2. The nested components can then read the shared information by using the `useContext` Hooks. An example that represents a simplified version of the HomeContext used in the Privacy Dashboard is shown in Listing 5.3. An example of a component that reads the Home context is also shown in Listing 5.4

```
function App() {
  return (
    <HomeProvider>
      <HomePage />
    </HomeProvider>
  );
}
```

Listing 5.2: Example of context wrapping

```
type HomeContextValue = {
  currentHomeUuid: string | undefined;
  updateCurrentHomeUUID: (uuid: string) => void;
};

const HomeContext = createContext<HomeContextValue | undefined>(undefined);

function HomeProvider({ children }: { children: React.ReactNode }) {
  const [currentHomeUuid, setCurrentHomeUuid] = useState<string | undefined>(
    undefined);

  const updateCurrentHomeUUID = (uuid: string) => {
    setCurrentHomeUuid(uuid);
  };

  return (
    <HomeContext.Provider value={{ currentHomeUuid, updateCurrentHomeUUID }}>
      {children}
    </HomeContext.Provider>
  );
}

function useHome() {
  return useContext(HomeContext);
}
```

Listing 5.3: Simplified example of the Home context

```
function HomeHeader() {
  const { currentHomeUuid } = useHome();

  return <div>Current home: {currentHomeUuid}</div>;
}
```

Listing 5.4: Example of context reading inside a component

5.4 Shared State in the Privacy Dashboard

To satisfy the objectives described in Section 5.1, the Privacy Dashboard needs a way to store some information available across multiple pages. An important example is the currently selected application. Once an application is selected in the installed or managed applications page, it becomes part of the shared frontend state and can be read and updated by different components. This allows the user to navigate the Privacy Dashboard while keeping the same application selected and opening different related sections, such as the privacy notice, the privacy impact assessment or the associated right requests. Another motivation for adopting a solution to share data is that, without it, navigating across pages would often require the frontend to retrieve the same information again from the backend when needed. In the case of the Privacy Dashboard, using props would have introduced excessive prop drilling, for this reason the Privacy Dashboard relies on React *Context*.

In the Privacy Dashboard, the shared state layer is implemented through three main contexts: `AuthContext`, `HomeContext`, and `AppsContext`. Their providers are nested in the following order:

`AuthProvider` → `HomeProvider` → `AppsProvider`

This order is required by the fact that home and application information can only be loaded when the current user is known after authentication and, for Data Subjects, application information also depends on the currently selected smart home.

Each context is now described in detail, followed by a table to summarize the main data and actions it exposes.

Context	**Dependencies**	**Responsibility**	**What it affects**
`AuthContext`	Backend authentication API	Authenticated user, role information, and session-related actions	Routing, visible pages, permissions, synchronization, and profile operations
`HomeContext`	`AuthContext`, homes service	Data Subject homes and current home selection	Installed apps, consent states, and right requests must be shown in multiple views
`AppsContext`	`AuthContext`, `HomeContext`, application-related services	Visible applications, selected application, and application resources	Selected application and its related resources are reused across different pages and panels

Table 5.1: Context hierarchy and functional responsibilities in the frontend.

5.4.1 AuthContext

`AuthContext` is the entry point of the shared state layer. Its main role is to store the authenticated user after it has retrieved the current session and derive the role flags used for authorization in the frontend, such as `isDataSubject`, `isDataController`, `isDpo` and `isManagerRole`. It also exposes the main operations related to the authentication session, namely login, registration, profile update, logout, and user deletion. Another important functionality is that it enables the Privacy Dashboard to start the synchronization process after login.

Element	Type	Role in the frontend
`user`	`UserModel | undefined`	Stores the currently authenticated user
`isDataSubject`, `isDataController`, `isDpo`, `isManagerRole`	`boolean`	Provide role information used by routing and available actions
`loading`	`boolean`	Indicates whether the context is still resolving the current session
`loginEventId`	`number`	Allows the synchronization after login to run exactly once
`login`, `register`, `updateProfile`, `logout`, `deleteUser`	`function`	Provide a unified interface for session-related operations

Table 5.2: Main states and functions exposed by `AuthContext`.

5.4.2 HomeContext

`HomeContext`, as the name suggests is responsible for providing information and actions to manage the current home of the user. Unlike `AuthContext`, it is populated only when the user is a Data Subject. The selected home determines which installed applications must be displayed and which consent states and right requests need to be loaded. The context enables also to preserve the currently selected smart home if it is still available after syncing.

5.4.3 AppsContext

`AppsContext` is the most extensive context in the Privacy Dashboard. Its main responsibility is to expose the list of applications visible to the current user and to manage the loading and synchronization of all the information related to them. It does so by exposing a single shared state for both the Data Subject's installed

Element	**Type**	**Role in the frontend**
`homes`	`HomeModel[] | undefined`	Stores the list of homes visible to the current Data Subject
`current_home_uuid`	`string | undefined`	The selected smart home, used by several pages
`loadingHomes`	`boolean`	Indicates that the frontend is still loading the home list
`updatingHomes`	`boolean`	Indicates that an explicit home synchronization is in progress
`errorHomes`	`string | undefined`	Propagates errors about home loading or synchronization
`homesSyncMessages`	`SyncMessage[]`	Stores warnings or errors occurred during home synchronization
`updateCurrentHomeUUID`	`(uuid: string) => void`	Allows the UI to change the current home
`syncHomes`	`() => Promise<void>`	Forces a backend refresh of the smart homes of the user

Table 5.3: Main data and actions exposed by `HomeContext`.

applications and the Data Controller's or Data Protection Officer's managed applications, so that the presentation layer does not need to differentiate between them. It also provides caching functionality both for applications in general and for privacy notices, privacy impact assessments, questionnaire and consent information. These caches make it possible to reuse information when the user navigates across different pages and to reduce API calls to the backend, while still allowing explicit refreshes using a force flag.

5.4.4 Error Handling

The implemented contexts are not only useful to expose shared data and actions, but also for collecting errors generated while loading. For example, `AppsContext` exposes the `errorApps` state, which is updated by the main loading operations, such as `loadApps`, `loadPrivacyNotice`, `loadPIA`, `loadQuestionnaire` and `loadConsents`. This makes the context a central point for error collection, making these errors available to the upper UI layers.

Element	Type	Role in the frontend
`apps`	`AppModel[] | undefined`	Stores the applications visible to the current user
`selectedAppUUID`	`string | undefined`	The selected application, used by several pages and panels
`selectedApp`	`AppModel | undefined`	Stores the currently selected application obtained from the apps list
`loadingApps`, `updatingApps`	`boolean`	They represents app loading and synchronization status
`errorApps`	`string | undefined`	Propagates errors about applications loading or synchronization
`appsSyncMessages`	`SyncMessage[]`	Stores warnings and errors during apps synchronization
`PrivacyNoticeByAppUUID`	`Record<string, PrivacyNotice | null | undefined>`	Stores privacy notices
`PIAByAppUUID`	`Record<string, PIA | null | undefined>`	Stores privacy impact assessments
`QuestionnaireByAppUUID`	`Record<string, Questionnaire | null | undefined>`	Stores questionnaire results and summaries
`ConsentStatesByAppUUID`	`Record<string, ConsentModel[] | undefined>`	Stores both available consent information and status for Data Subjects
`isSelectedAppOwner`	`boolean`	Allows the UI to distinguish ownership specific actions

Table 5.4: Main data and caches exposed by `AppsContext`.

Exposed action	Responsibility	Lower layer used
`loadApps`	Loads the full apps list or a specific application and updates shared app state	`ApplicationService`
`syncApps`	Refreshes installed apps for Data Subjects and records synchronization warnings or errors	`ApplicationService`
`createManagedApp`	Creates a local app, inserts it into the shared app state and updates it	`ApplicationService`
`loadPrivacyNotice`	Loads or reuses a privacy notice and updates document availability in the app list	`PrivacyNoticeService`
`loadPIA`	Loads or reuses a privacy impact assessment and updates document availability in the app list	`PrivacyImpactAssessment Service`
`loadQuestionnaire`	Loads or reuses questionnaire data and updates summary and evaluation fields on the app model	`QuestionnaireService`
`loadConsents`	Loads or reuses app available consent and status for Data Subjects	`AvailableConsentService`
`setSelectedAppUUID`	Updates the selected app used by several pages and detail panels	None

Table 5.5: Actions exposed by `AppsContext` and their connection with the service layer.

Chapter 6

Integrating Smartotum

This chapter provides additional details on the integration of Smartotum into the Privacy Dashboard, focusing on the authentication system, how the platform interfaces and synchronizes data from the user's smart home and how policy rules are pushed to Smartotum.

6.1 Practical Challenges

As anticipated in Section 2.1.3, Smartotum adopts a local-first architecture where the core intelligence resides inside the household and does not rely on cloud services. Consistently with the SIFIS-Home approach, Smartotum stores and exposes smart home data (Section 3.2.1) through a DHT, so accessing the information requires interacting with it. In practice, this means connecting directly to an entity that can relay the relevant information. In Smartotum, this entity is not a cloud service but the hub deployed inside the user's smart home.

The first challenge to tackle was the design of a single authentication mechanism that supports both Data Subjects and Data Controllers, while relying on Smartotum for Data Subject authentication and avoiding any server-side storage of Smartotum credentials or access information. Before presenting the proposed solution, we briefly introduce Smartotum's authentication mechanism, which relies on JWT, an IETF standard widely used in modern web applications and APIs [27].

6.2 JSON Web Token

The open standard JWT (JSON Web Token) defines a compact and self-contained way for securely transmitting information between parties as a JSON object. This information can be verified and trusted because it is digitally signed. JWTs can be signed using a secret (with the HMAC algorithm) or a public/private key pair

using RSA or ECDSA [28]. Smartotum specifically uses RS256, a commonly used name to abbreviate the complete algorithm: RSASSA PKCS1 v1.5 using SHA-256. The process works as follows.

After the user submits their email and password, Smartotum responds with a `st_access_token`, a JWT that the Privacy Dashboard can use to access their protected endpoints requiring authentication.

JWT format JWTs follow a standardized format composed of three parts separated by dots: the Header, the Payload and the Signature, all encoded using Base64URL.

```
<base64url(header)>.<base64url(payload)>.<base64url(signature)>
```

Listing 6.1: Smartotum access token structure

The first part, the header, once decoded, contains three important fields:

- **typ**: The type of the token, in this case JWT.
- **alg**: The cryptographic algorithm (asymmetric or symmetric) used to sign the token, such as HMAC SHA256 or RSA, in this case RS256.
- **kid**: The key identifier used to select the public key required to verify the token signature, as described in Paragraph 6.2, ensuring integrity and authenticity.

```
{
  "kid": "d-<key-id>",
  "typ": "JWT",
  "version": "3",
  "alg": "RS256"
}
```

Listing 6.2: Smartotum JWT header example

The second part, the payload, once decoded, contains some standard fields and others defined by the provider of the service, in this case Smartotum. The most important for our use case are:

- **iss**: identifies the authority that generated and signed the JWT (issuer), in this case Smartotum, used in the verification process to ensure the token comes from the expected provider.
- **exp**: the timestamp of the token expiration time. It is used in the verification process to ensure that the token is still valid.
- **sub**: the unique Smartotum ID of the authenticated user to whom the token refers. It is used by the Privacy Dashboard to globally identify the Data Subject.

```
{
  "iat": 1700000000,
  "exp": 1700003600,
  "sub": "<user-uuid>",
  "sessionHandle": "<session-handle>",
  "refreshTokenHash1": "<refresh-token-hash>",
  "parentRefreshTokenHash1": null,
  "antiCsrfToken": null,
  "iss": "https://api.smartotum.com/auth",
  "st-role": {
    "v": [
      "standard"
    ],
    "t": 1700000000123
  },
  "st-perm": {
    "v": [],
    "t": 1700000000456
  }
}
```

Listing 6.3: Smartotum JWT payload example

The last part, the signature, is generated by Smartotum through the following steps:

1. The header and payload are serialized as JSON, encoded with Base64URL, and concatenated with a dot.
2. The resulting string is signed using the RS256 algorithm, which applies SHA-256 and then signs the result with Smartotum's RSA private key using PKCS#1 v1.5 padding. [1]
3. The resulting signature is finally encoded with Base64URL and appended as the third part of the token.

After the token is retrieved, the process called validation is needed to make sure it is structurally correct.

[1] PKCS#1 v1.5 defines the encoding method applied to the hashed message before the RSA signature is computed; for the formal definition of RSASSA-PKCS1-v1_5 and EMSA-PKCS1-v1_5, see RFC 8017, Sections 8.2.1 and 9.2: `https://www.rfc-editor.org/rfc/rfc8017`.

Validation JWTs Following the terminology used by jwt.io [28], the validation process generally refers to checking:

- **Structure**: Ensuring the token has the standard three parts (header, payload, signature) separated by dots
- **Format**: Verifying that each part is correctly encoded (Base64URL) and that the payload contains expected claims.
- **Content**: Checking if the claims within the payload are correct, such as expiration time (exp), issued at (iat), not before (nbf), among others, to ensure the token isn't expired, isn't used before its time, etc.

Verification JWTs In our case the JWT received is signed but not encrypted, so the header and payload can be decoded by anyone in possession of the token and possibly used maliciously. Also, decoding and validating the token alone are not sufficient to establish trust in it because it may have been altered.

For these reasons, we need to protect the token during transport to guarantee confidentiality, and an additional verification process to make sure integrity and authenticity are preserved. Firstly, confidentiality is provided by the transport layer if the token is received over a TLS-protected channel. Secondly, integrity and authenticity are provided by signature verification that leverages the RSA cryptographic algorithm and the use of the public key.

To retrieve the Smartotum set of public keys, called JWKS (JSON Web Key Set), a dedicated endpoint is used (`https://api.smartotum.com/auth/jwt/jwks.json`), for which an example response is shown in Listing 6.4:
In practice, the steps in the verification process are the following:

1. **Header kid extraction**: extract the kid from the decoded JWT header (Listing 6.2).
2. **JWKS public key selection**: select the corresponding JWK from the JWKS document by matching the kid extracted from the header (Listing 6.4).
3. **Retrieve n and e**: from the selected JWK, the RSA modulus (n) and the RSA public exponent (e) are extracted, Base64URL-decoded and converted to integers.
4. **Reconstruct the public key**: reconstruct the RSA public key using `n` and `e`.
5. **Signature decoding**: decode the signature of the JWT from Base64URL.

6. **RSA verification**: RSA verification is applied to the decoded JWT signature using the reconstructed public key, producing a PKCS#1 v1.5-compliant encoded message. The digest extracted from this message is then compared with the digest obtained by computing SHA-256 over the first two parts of the JWT, including the dot (see Listing 6.1).

```
{
  "keys": [
    {
      "kty": "RSA",
      "kid": "d-<key-id>",
      "n": "<base64url-modulus>",
      "e": "AQAB",
      "alg": "RS256",
      "use": "sig"
    },
    {
      "kty": "RSA",
      "kid": "s-<key-id>",
      "n": "<base64url-modulus>",
      "e": "AQAB",
      "alg": "RS256",
      "use": "sig"
    }
  ]
}
```

Listing 6.4: JWKS Smartotum public keys examples

Only if these verification steps succeed, the token claims in the payload (see Listing 6.3) can be trusted.

At that point, the backend also enforces basic claim checks, such as validating the token issuer (`iss`) and ensuring that the token has not expired (`exp`). Once all checks pass, the token can be trusted and the subject identifier (`sub`) can be used to identify the authenticated Smartotum user.

Now that we understand how authentication works, the solution that enables Data Subjects, Data Controllers and Data Protection Officers to authenticate can be presented.

6.3 Privacy Dashboard Authentication System

The Privacy Dashboard authentication system is divided into two parts for the two categories of users:

- **Data Subjects**: Data Subjects correspond to smart home owners who want to manage the privacy of their smart homes, so their account must be linked to the underlying home automation system that for now is only Smartotum
- **Data Controllers and DPOs**: Data Controllers and Data Protection Officers, unlike Data Subjects, they do not access or view individual homes: they manage privacy-related assets (consents and GDPR documents) for the applications they control across all users who install them.

To support these different roles, and to keep the platform open to future home automation systems, the Privacy Dashboard adopts a dual registration and login system.

Data Subjects Upon registration, the user provides their email and password which are forwarded by the backend to Smartotum over a TLS protected channel. Smartotum responds with a JWT, which is then validated and verified as described in Paragraph 6.2.

Once validated, the resulting `sub` claim (Smartotum user identifier) and the user email are stored in the Privacy Dashboard database, the former being used to link the user account to the Smartotum ecosystem.

During subsequent logins, the `sub` claim is read from the decoded and verified JWT and used to look up and retrieve the user's information from the database, which is returned in the response if authentication is successful.

The Smartotum access token is stored on the user's browser as a protected session cookie and automatically attached to subsequent requests.

Data Controllers and DPOs Unlike Data Subjects, Data Controllers and DPOs authenticate directly against the Privacy Dashboard and do not rely on Smartotum for credential verification.

During registration, their credentials are stored locally, in particular the password is saved as a salted hash for provided security.

At login time, the backend validates the provided email and password against the stored hash and, if authentication is successful, issues an internally signed JWT.

Similarly to the Smartotum access token, it is stored on the user's browser as a protected session cookie and automatically attached to subsequent requests and also used to retrieve the corresponding user record from the database for any

endpoint that requires it. The difference is that, in this case, the token is fully managed by the Privacy Dashboard, so it is issued and validated locally, and can also be refreshed when approaching expiration.

As a result, while Data Subjects rely on Smartotum issued tokens verified via Smartotum public keys, Data Controllers and DPOs rely on internally issued tokens validated locally.

Security trade-off Although Smartotum credentials transit through the backend, they are not stored in any capacity. This poses a trust assumption on the backend, however it was chosen to match Smartotum's available authentication interface and to keep token handling confined to secure browser cookies rather than client-side storage.

Performance trade-off For Data Subjects, the same verification and lookup process is performed not only at login but also for every Privacy Dashboard endpoint that requires an authenticated user and needs access to their information to handle the request. This introduces an additional overhead, but the implementation reduces it through two caching layers. First, the client used to retrieve Smartotum JWKS caches the retrieved key set and reuses it across different requests, so that it is not fetched every time. Second, once the token has been decoded within a request, both the claims and the resolved current user are cached for that request. This makes the verification process lighter without introducing a persistent cache of previously verified tokens.

6.4 Distributed Hash Table (DHT)

A DHT (Distributed Hash Table) is a decentralized storage and lookup mechanism that provides an abstraction similar to a hash table, but distributes data across multiple nodes instead of storing them in a single central component. More specifically, similarly to a hash table, a DHT supports a `put` operation to store an item and a `get` operation to retrieve it through an associated key. When an item is inserted, the associated key is used to derive an identifier, typically by applying a hash function. These identifiers belong to a common logical space which is divided among the participating nodes. As a result, each node is responsible for only a portion of the information stored in the DHT [29].

To make retrieval possible, every node maintains a partial view of the system and uses it to forward requests towards the node responsible for the requested item, typically reaching it in $O(\log N)$ hops. This approach is useful in large scale distributed systems because it allows data to be stored and located efficiently without relying on a central server and without requiring requests to be propagated

across the whole network [29].

As a result, the main properties required for a working DHT are decentralization, scalability, and fault tolerance as nodes join, leave, and fail over time [29].

6.4.1 Interacting with Smartotum DHT

In Smartotum, these principles are applied internally by the DHT, while external components interact with it through a topic-based interface used to store and retrieve smart home information. In practice, entities such as rooms, devices, and rules are represented as DHT entries, while installed applications are retrieved through a dedicated Smartotum endpoint.
Each entry follows a uniform JSON structure:

```
{
  "topic_name": "<topic-name>",
  "topic_uuid": "<topic-uuid>",
  "value": { }
}
```

Listing 6.5: Smartotum topic structure example

- **topic_name**: The type of entry stored in the DHT, such as `domo_room`, `domo_camera`, `domo_light`, `privacy_rule`.
- **topic_uuid**: The UUID (Universally Unique Identifier) of the specific entry within the DHT, is used to globally identify a device or entity
- **value**: The entry content; every entry with the same **topic_name** share the same structure

For each topic, the Privacy Dashboard retrieves a list of entries and interprets the `value` field. In practice, all entries with the same `topic_name` share the same `value` structure. In the following sections, we describe the most relevant topics.

Rooms

Rooms are represented with the `domo_room` topic name.
An example of a room is shown here:

```
{
  "topic_name": "domo_room",
  "topic_uuid": "<living-room-topic-uuid>",
  "value": {
    "name": "living Room",
    "category": "living",
    "order_id": 0,
    "homeassistant_area_id": "stanza_1"
  }
}
```

Listing 6.6: Smartotum `domo_room` example

Rooms enable the Privacy Dashboard to present a location aware view of the smart home and are important in supporting the creation of policies at a higher level than a single device. In practice, with the link between devices and rooms we can create policies targeting a whole room instead of single devices.

Devices

Devices are represented with a `topic_name` that identifies their logical category:

- `domo_light`
- `domo_light_dimmable`
- `domo_rgbw_light`
- `domo_switch`
- `domo_roller_shutter`
- `domo_camera`

An example of a `domo_light` is shown here:

```
{
  "topic_name": "domo_light",
  "topic_uuid": "<light-topic-uuid>",
  "value": {
    "name": "Living Room Light",
    "status": false,
    "power": 0,
    "energy": 0,
    "area_name": "<living-room-topic-uuid>",
    "note": "living room light"
  }
}
```

Listing 6.7: Smartotum `domo_light` example

From the perspective of the Privacy Dashboard, devices are primarily used for:

1. Presenting the list of controllable objects to the user in the policy translation GUI.
2. Resolving rule targets during policy translation.
3. Linking devices to their specific room through the `area_name` field that contains the UUID of the corresponding room topic.

Rules

At the Smartotum enforcement layer, device-level rules are represented with the `privacy_rule` topic name. Within the Privacy Dashboard, rules published to the Smartotum DHT are used to disable a single target device under specific conditions. An example of a rule is shown here:

```
{
  "topic_name": "privacy_rule",
  "topic_uuid": "<rule-uuid>",
  "value": {
    "target_topic": "domo_camera",
    "target_uuid": "<target-camera-topic-uuid>",
    "time_start": "10:00",
    "time_end": "14:00",
    "days": ["Friday"],
    "expiration_date": "2026/08/15"
  }
}
```

Listing 6.8: Smartotum rule topic example (`privacy_rule`)

As shown, the rule has some fields to reference the target device:

- `target_topic`: the target device topic, for example a camera
- `target_uuid`: the UUID of the specific target topic (a specific device for example a camera)

Optionally, temporal constraints can be included to restrict the validity of the rule, such as a time window (`time_start`, `time_end`), a set of days (`days`), and an overall expiration date (`expiration_date`).

6.5 Fetching Installed Applications

The installed applications inside the smart home are retrieved with a dedicated endpoint: `GET /app/systems/{system_id}/installed_apps`. A snippet of the response is shown below:

```
[
  {
    "id": "com.smartotum.smartotum-camera-manager",
    "name": "Smartotum-camera-manager",
    "description": "for managing cameras",
    "data_controllers_email": ["controller.smartotum@example.com"],
    "available_consents": [
      {
        "content": "processing of personal information to provide the
   service",
        "sifis_action_id": "sifis_record_video_action"
      }
    ]
  },
  {
    "id": "com.smartotum.smartotum-certbot",
    "name": "Smartotum-certbot",
    "description": "for managing SSL HTTPS certificates",
    "data_controllers_email": ["controller.smartotum@example.com"],
    "available_consents": [
      { "content": "processing of personal information to provide the
   service" }
    ]
  },
  ....
]
```

Listing 6.9: Smartotum installed applications example

Each entry represents an application installed in the smart home environment and includes:

- `id`: The unique identifier, the value after `com.` indicates the application source and the application ID.
- `name`: The name of the application.
- `description`: A brief description of what the app does.
- `data_controllers_email`: The email addresses of all Data Controllers assigned to the application, with the first one being the owner.
- `available_consents`: The consents required for the application to operate. A consent can be linked to a specific `sifis_action_id` so that, when it is revoked, a corresponding deny rule is created, targeting all devices that expose that action, as described in Section 3.2.3.

The Smartotum camera manager application shown in Listing 6.9 is used as a recurring example to show how a user consent decision can be translated into an enforcement rule in the smart home.

6.6 Synchronization Architecture

We now describe the synchronization architecture that allows the dashboard to maintain an updated view of the user's smart homes. Given the amount of information to retrieve, synchronization is explicitly triggered by the frontend whenever needed, rather than being implicitly performed by the backend on every request. As a result, the information shown to the user represents a snapshot taken the last time synchronization was executed and is persisted in the Privacy Dashboard database for subsequent queries.

This design keeps the backend endpoints responsive and avoids making interactions that do not modify the home state dependent on the availability and response time of the DHT. If the DHT is temporarily unavailable, for example due to connectivity issues with the home hub, the dashboard remains usable on locally stored data.

It is important to note, however, that accessing protected functionality still requires Smartotum authentication, and any operation that pushes changes to the smart home requires the DHT to be available. To further reduce the time required for a complete synchronization, we introduced two endpoints, each of which focuses on a subset of the data.

6.6.1 Homes Synchronization

The endpoint `POST /api/homes/refresh` triggers the synchronization of the smart homes associated with the authenticated user, updating the local snapshot of homes, rooms, devices and rules.

First, the list of systems (homes) is retrieved from Smartotum and for each of them, the synchronization process begins.

Home The home synchronization follows an *upsert + prune* pattern.

In practice, for each home returned by Smartotum, a home record is created in the local database if it does not already exist, otherwise the existing record is updated. The process also ensures that the home is linked to the user. For each home some important information are stored in the database, such as the UUID, name, address, ZIP code and country for future reference.
At the end of the whole synchronization process, the homes currently owned by the user are compared with the Smartotum provided list. If a home is no longer present, the ownership relation for that user is removed, but the home is not deleted until it has no remaining owners. This flow was chosen because Homes can be shared among Data Subjects, so detaching ownership avoids deleting shared data prematurely.

Rights requests are not lost in this case, and also if the home is later deleted, because they are treated as audit records and so they store the minimal identifiers needed to remain available (Data Subject, application and smart home).

Rooms The rooms synchronization follows an *upsert + prune* pattern.

For the selected home, the dashboard retrieves the full list of rooms from the Smartotum DHT. For each room returned, a room record is created in the local database if it does not already exist, otherwise the existing record is updated. The process also ensures that each room is linked to the corresponding home. The database record for each room stores the room identifier and room name for future reference.
At the end of this phase, the rooms currently stored for that home are compared with the Smartotum provided list. If a room is no longer present, it is removed from the local database to match the current Smartotum state. A dedicated local room named `Unassigned devices` is also maintained to represent devices that are not associated with a specific room.

Devices still linked to the previous room temporarily lose the association, which is restored during the device synchronization phase.

Devices Devices are usually the largest collection in the DHT. For this reason, they are fetched by iterating over the supported device topics one at a time. Once

the full list has been collected, device synchronization follows the same *upsert + prune* pattern used for rooms. Each device is linked to its corresponding room if available, or associated with the local `Unassigned devices` room. This updated device snapshot is then used by the subsequent synchronization steps, in particular for ontology regeneration and rule alignment.

Home Ontology After devices are updated, the home ontology is regenerated so that it reflects the current structure of the smart home. The home ontology can be seen as a structured model of the house, which describes the available rooms, devices, and their relationships. This updated model is then used by the following synchronization steps, in particular for rule management and alignment.

Rules After regenerating the home ontology, the dashboard retrieves the current list of rules from the Smartotum DHT and aligns the local rule representation. This phase is more complex than the previous synchronization steps because rules exist at two different levels of abstraction:

- `enforcement layer`: Smartotum DHT rules are device-level rules, since each DHT rule refers to a single target device.
- `policy layer`: The Privacy Dashboard may manage higher-level logical rules that conceptually apply to multiple devices but locally are represented as a single logical rule. Examples include consent-related rules that apply to the entire home and user-defined privacy rules that may refer to the whole home or to a specific room.

For this reason, the synchronization process must support both single and multiple device rules:

- Rules that already correspond to a single target device can be synchronized directly.
- When multiple DHT entries correspond to the same logical policy-layer rule, they are grouped together and shown as a single local rule in the Privacy Dashboard.

Finally, rules that are no longer consistent with the current smart home devices and DHT rules are removed. An example of the mapping between logical Privacy Dashboard rules and Smartotum device-level rules is shown in Table 6.1.
In this way, the Privacy Dashboard provides a coherent representation of rules while remaining compatible with the device-level rule model used in Smartotum.

6.6.2 Applications Synchronization

The endpoint `POST /api/applications/refresh` triggers the synchronization of the applications installed in the smart homes associated with the authenticated user. This procedure updates the local snapshot of installed applications, responsible Data Controllers, available consents and consent-related enforcement rules.

First, the list of systems (homes) is retrieved from Smartotum and, for each of them, the synchronization process begins.

Installed Applications The installed applications synchronization follows an *upsert + prune* pattern.

For each home returned by Smartotum, the dashboard retrieves the list of installed applications. If the home is not yet available in the local database, a targeted home synchronization is performed first, so that the home becomes available for the application update. Each application returned is added in the local database if it does not already exist, otherwise it is updated. In any case, the process ensures that the application is linked to the authenticated user and to the corresponding home as an installed application. The database record for each application stores the application identifier, title, description and owner reference for future reference.

At the end of this phase, the applications currently associated with the user for that home are compared with the list provided by Smartotum. If an application is no longer present, only the installation relation for the current user and home is removed to match the current Smartotum state. The application record remains in the local database even if all relations to users are deleted, because the same application can also be linked to Data Controllers, DPOs, GDPR documents, or consent records. Even if the application also loses its links to Data Controllers and DPOs, it is still preserved in the local database for historical reasons.

To summarize, synchronization removes only the installation relation for the current user-home pair, while preserving the shared application entity and its related data.

Application Ownership For each application, the dashboard also processes the responsible Data Controllers returned by Smartotum. The first user in the list is treated as the application owner, while the others as additional managers or delegates. This information is applied only initially when the application is first synchronized.

After that, the Privacy Dashboard serves as the primary local reference for role management, in line with its role as the dashboard for privacy and consent management.

Available Consents The available consents synchronization follows an *upsert + prune* pattern. These records define the set of privacy choices that can be presented to the user for that application. For each application, the dashboard retrieves the list of available consents together with the optional mapping between each consent and the corresponding Smartotum action. Missing consents are added to the local database, while existing ones are preserved and their action mapping is updated when needed. If an available consent is no longer present in Smartotum, the dashboard removes it from the local catalog only after the checks described in the following step have been carried out.

This approach is aligned with the philosophy of the Privacy Dashboard as the center for managing privacy and consents in smart homes.

Alignment of Consent State and Enforcement Rules Once the catalog of available consents has been updated, the dashboard aligns the corresponding enforcement rules according to the current consent state and then updates the device-level rules on the DHT. More in detail, the given or not given state of each consent is taken from the local database, where it is maintained by the Privacy Dashboard.

- If a consent is currently given, any corresponding policy rule and device-level rules are removed.
- If a consent is not given and an action mapping is available, the corresponding policy rule is created in the local database if missing, and one deny rule for each device is ensured in the smart home DHT.
- If a consent has been removed in the previous step, the dashboard first removes any related policy rule and the corresponding device-level rules before deleting the consent from the local catalog.

In this way, the consent state managed by the dashboard is reflected in the device-level enforcement rules published to Smartotum. In the recurring example of the camera manager application, this means that withdrawing the consent related to video recording results in the presence of the corresponding deny rule, while granting the same consent removes it.

Case	**Frontend**	**Backend**	**DHT**
Single-device deny	Record Video, one device, deny	Derives DHT `target_uuid` and `target_topic` from the selected device	One `privacy_rule` for that device
Room-level deny	Record Video, one room, deny	Resolves all devices in that room that expose `sifis_record_video_action`	One `privacy_rule` per compatible room device
Home-level deny	Record Video, entire home, deny	Resolves all devices in the home that expose `sifis_record_video_action`	One `privacy_rule` per compatible home device
Home-level permit	Record Video, entire home, permit	Resolves all devices in the home that expose `sifis_record_video_action`	**No rules** written to the DHT

Table 6.1: From Privacy Dashboard PTP rules to Smartotum device-level rules

Chapter 7

Marketplace Proof of Concept

7.1 Applications Marketplace

In this thesis, we have already discussed two main sources of applications for the Privacy Dashboard:

- `Smartotum`: The dashboard periodically syncs the installed applications from Smartotum and stores a local snapshot in the database, as shown in Section 6.5.
- `Manual creation`: A Data Controller can manually create applications, as shown in Section 3.2.5.

We now describe a third way to add applications to the Privacy Dashboard, which was briefly introduced in Section 3.2.4. The Marketplace is meant to be an external way for a Data Controller to register applications into the dashboard as the owner. Once the application is registered, the Data Controller can manage the application, define available consents and handle all GDPR requests and documents.

App Type	App Metadata	Owner	Managers	DPO	Consent Catalog	Consent State	sifis_ action_id	DHT Rules
SMARTOTUM	Smartotum	Smartotum (Initial)	Smartotum (Initial) + Dashboard	Dashboard	Smartotum	Dashboard	Smartotum	Yes
MARKETPLACE	Marketplace (Initial + updates)	Marketplace (Initial)	Marketplace (Initial) + Dashboard	Dashboard	Dashboard	Dashboard	No	No
LOCAL	Dashboard (Initial + updates)	Dashboard (Initial)	Dashboard	Dashboard	Dashboard	Dashboard	No	No

Table 7.1: Comparison of responsibilities between application sources

7.1.1 Marketplace API

The Marketplace can interact with the dashboard using two dedicated endpoints:

`POST /api/integrations/marketplace/apps`

To register an application for a Data Controller.

```
{
  "id": "com.marketplace.smart-camera-manager",
  "name": "Smart Camera Manager",
  "description": "App for managing IP cameras in a smart home",
  "owner_email": "controller.general@example.com",
  "delegates_email": ["controller.smartotum@example.com"]
}
```

Listing 7.1: Marketplace API Post example

The payload follows a similar pattern to the applications synced from Smartotum. The main differences are that the Data Controller owner and delegates are represented by two separate fields (`owner_email` and `delegates_email`), while the catalog of available consents is managed locally from the Privacy Dashboard after the application has been created. Some parameters are required (`id`, `name` and `owner_email`) while others (`delegates_email` and `description`) are optional. The register endpoint has the following properties:

- The application identifier must start with `com.marketplace`, so that Marketplace applications can be distinguished from manually created applications and Smartotum ones.
- The endpoint is create only: if the Marketplace sends the same application again, the dashboard does not create a second copy and does not overwrite any local changes.
- The request is accepted only if the Data Controller owner and the delegates, if present, are already registered in the dashboard.
- The request is rejected if the application is already present but linked to a different owner.

`PATCH /api/integrations/marketplace/apps/<application_id>`

To update an application already registered in the dashboard. The payload only supports `title` and `description`. No other parameters, such as available consents, are supported because, once the application has been registered, the Privacy Dashboard becomes the source of truth for GDPR documents, role management and available consents.

```
{
  "title": "Smart Camera Manager (v2)",
  "description": "Marketplace app for managing IP cameras in a smart
   home"
}
```

Listing 7.2: Marketplace API Patch example

7.1.2 Authentication

The Marketplace authenticates using a static Bearer token loaded by the backend at startup from the `MARKETPLACE_INTEGRATION_TOKEN` environment variable. Both Marketplace endpoints require the header of the HTTP request to contain the token used to authenticate the Marketplace. This approach is sufficient for demo purposes, but for production deployment it should be replaced with a more robust solution (e.g., OAuth2[1]).

7.1.3 Use Cases

Marketplace ingestion of a new application

This use case describes how an external Marketplace registers a new application into the Privacy Dashboard for a specific Data Controller.

Primary actor External Marketplace

Preconditions

- The Marketplace knows a valid Bearer token.
- The application identifier must match the format `com.marketplace.<slug>`.
- The target Data Controller is already registered in the Privacy Dashboard.
- If delegates are specified in the request, they must already be registered in the Privacy Dashboard.
- The application identifier is not yet present in the dashboard, or it is already present with the same owner.

[1]OAuth 2.0, short for Open Authorization, is a standard that allows a website or application to access resources hosted by other web applications on behalf of a user: `https://auth0.com/intro-to-iam/what-is-oauth-2`.

Main success scenario

1. The Marketplace sends a `POST /api/integrations/marketplace/apps` request with the application information listed in Listing 7.1.
2. The Privacy Dashboard validates the Bearer token.
3. The Privacy Dashboard validates the payload fields, including the application identifier, owner email, and optional delegates.
4. The Privacy Dashboard checks that the owner exists and has role Data Controller.
5. If delegates are provided, the Privacy Dashboard checks that they also exist and have role Data Controller.
6. The Privacy Dashboard creates the new application in the local database.
7. The Privacy Dashboard links the owner to the application as its main Data Controller.
8. The Privacy Dashboard links the optional delegates to the same application.
9. The application becomes visible among the managed applications of the owner and delegates inside the dashboard.

Alternative flows

- If the application already exists with the same owner, the dashboard does not create a duplicate and does not overwrite local data.
- If the application already exists with a different owner, the request is rejected.
- If the owner or one of the delegates does not exist, the request is rejected.
- If one of the provided users is not a Data Controller, the request is rejected.

Example scenario The Marketplace wants to register the application: `com.marketplace.smart-camera-manager`. The dashboard validates the request, creates the application, assigns the selected owner and delegates, and makes the application available in the managed applications area of the selected Data Controllers.

Marketplace update of an existing application

This use case describes how the Marketplace updates the metadata of an application that was previously registered in the Privacy Dashboard.

Primary actor External Marketplace

Preconditions

- The Marketplace knows a valid Bearer token.
- The application is already registered in the Privacy Dashboard.
- The application identifier must match the format `com.marketplace.<slug>`.
- At least one updatable field is provided in the payload.

Main success scenario

1. The Marketplace sends a `PATCH /api/integrations/marketplace/apps/<application_id>` with possible fields listed in Listing 7.2.
2. The Privacy Dashboard validates the Bearer token.
3. The Privacy Dashboard validates the application identifier and checks that the application exists.
4. The Privacy Dashboard reads the patch payload.
5. If a new `title` is provided, the dashboard updates the display title of the application.
6. If a new `description` is provided, the dashboard updates the textual description of the application.
7. The dashboard commits the changes and makes the updated application immediately visible to the users managing it.

Alternative flows

- If the application does not exist, the request is rejected.
- If none of `title` or `description` is provided, the request is rejected.
- If the identifier does not match the Marketplace namespace, the request is rejected.

Example scenario A Marketplace updates the application `com.marketplace.smart-camera-manager` by changing its title to `Smart Camera Manager (v2)` and its description. The dashboard updates the application metadata, while the consent catalog continues to be managed locally from the Privacy Dashboard.

7.1.4 Limitations

For Marketplace applications, available consents can still be defined, but they cannot be linked to any `sifis_action_id` because, due to a limitation of Smartotum, these applications cannot currently be installed in a smart home. For this reason, the Marketplace is currently limited to the registration of applications for Data Controllers and DPOs and does not support the complete flow in which a Data Subject can browse Marketplace applications and install them directly into their smart home.

Chapter 8

Continuous Integration, Delivery and Deployment

8.1 Overview

Before describing how we integrated CI (Continuous Integration), continuous delivery and continuous deployment into our workflow, we first provide a brief overview of the underlying concepts.

8.1.1 Continuous Integration

CI (Continuous Integration) refers to the practice of automatically and frequently integrating code changes into a shared source code repository. In a continuous integration process, code changes are considered correctly integrated when, after being merged, they are automatically built and validated through tests at different levels, so that existing functionality remains unaffected. Continuous integration is important because it helps reduce bugs and workflow complexity while improving efficiency and reducing the time between code changes and feature releases [30].

In short, continuous integration tries to answer the question: *"Can the new code be integrated safely?"*.

8.1.2 Testing Levels

We now briefly introduce the types of tests that should be considered when introducing a CI process:

Unit Tests Unit testing is the lowest testing layer among those considered, as it refers to the verification of small parts of an application, down to individual

functions or classes, in isolation from the rest of the system. For this reason, it provides a fast way to detect regressions and precisely locate defective code. Unit testing is also useful to ensure that individual components behave correctly even when receiving uncommon inputs.

Integration Testing Integration Testing covers larger parts of the application that include multiple components and focuses on how they interact with each other. In practice, these tests verify that different parts of the system work correctly together, such as application code and the database. Integration tests are important because many defects do not occur within individual components but arise from the interactions between them.

End-to-End Testing (E2E) E2E (End-to-End Testing) covers even larger part of the system for example it includes an entire application workflow from beginning to end. In particular is meant to verify that integrated components such as the front end, back end, databases, and third-party services work correctly together while reflecting realistic user scenarios [31].

For these reasons E2E testing is a good approach to obtain the most complete picture of how an application behaves from start to finish, including the user perspective.

8.1.3 Continuous Delivery and Continuous Deployment

Continuous Delivery Continuous delivery takes a step further than continuous integration by trying to answer the following question: *Is the validated software now ready to be released?*
After code changes have been successfully integrated into a shared source code repository and validated through a comprehensive automated test suite, the application is built and released in a deployable form. The final deployment step is not necessarily automatic, but it should be fast and require minimal effort [30].

Continuous Deployment Continuous deployment extends this approach by automating the final step as well, so that once the application has been built, tested, and released in a deployable form, it is also automatically deployed to the target environment, usually production where it is usable by customers. Because there is no manual approval step, continuous deployment relies heavily on well-designed test automation [30]. In short, continuous deployment tries to answer the question: *Can the validated software be deployed automatically to the target environment?*

8.1.4 Privacy Dashboard End-to-End Testing

In the Privacy Dashboard we focus on the creation of a comprehensive E2E testing suite to validate the most critical user workflows from start to finish. The tests are implemented as a Postman collection executed with Newman and cover scenarios such as authentication, consent management, GDPR requests, and the automatic creation and removal of policy and devices rules.

Video Recording Consent End-to-End Testing

A particularly meaningful testing scenario, which concludes the recurring example presented throughout the thesis, is the consent granting and withdrawal process for the Smartotum camera manager application. It verifies that granting or withdrawing the consent related to video recording is correctly reflected both in the dashboard state and in the corresponding rules stored in the DHT. The workflow is composed of the following tests:

1. `GET /api/applications/home/{home_uuid}`
   Retrieves the applications installed in the data subject's smart home and checks that the application `com.smartotum.smartotum-camera-manager` is present. Its application UUID is then stored for the following requests.

2. `GET /api/consents/home/{home_uuid}/application/{app_uuid}`
   Retrieves the list of consents associated with the camera manager application and checks that the expected consent is present and marked as not granted.

3. `GET /api/dht/homes/{home_uuid}/topic_name/domo_camera`
   Retrieves the camera devices available in the smart home. The UUIDs of these devices are stored because they are needed in step 5 to verify that the consent-related rule is applied to the correct targets.

4. `GET /api/sifis/homes/{home_uuid}/rules`
   Retrieves the current SIFIS rules of the home and verifies that a deny rule associated with the selected consent already exists. The UUID of this rule is also stored.

5. `GET /api/dht/homes/{home_uuid}/topic_name/privacy_rule`
   Retrieves the device rules currently applied in the smart home. Then, using the information collected in the previous steps, it verifies that the expected devices rules exists for the target camera devices and also that its associated data is correct and consistent.

6. `PUT /api/consents/home/{home_uuid}/application/{app_uuid}`
   Grants the selected consent, making sure that the operation is successful.

7. `GET /api/consents/home/{home_uuid}/application/{app_uuid}`
   Reads the consent state again after the update and checks that it is now marked as granted.

8. `GET /api/sifis/homes/{home_uuid}/rules`
   Retrieves the SIFIS rules after the consent has been granted and checks that the deny rule associated with the selected consent has been removed.

9. `GET /api/dht/homes/{home_uuid}/topic_name/privacy_rule`
   Retrieves again the device rules currently applied in the smart home and makes sure that the rules previously associated with the consent are no longer present.

10. `POST /api/requests/submit`
    Submits a GDPR request of type `WITHDRAW_CONSENT` and checks that a request UUID is returned.

11. `GET /api/consents/home/{home_uuid}/application/{app_uuid}`
    Reads the consent state after the withdrawal request and checks that it is now marked as revoked.

12. `GET /api/sifis/homes/{home_uuid}/rules`
    Retrieves the SIFIS rules after the withdrawal and checks that the deny rule associated with the selected consent has been recreated.

13. `GET /api/dht/homes/{home_uuid}/topic_name/privacy_rule`
    As in step 5, now that the consent has been withdrawn again, it verifies that the expected device rules exists for the target camera devices and also that its associated data is correct.

This E2E Test clearly shows why such an approach is valuable to obtain the most complete picture of the system behavior: while it appears to validate a single functionality, it actually exercises many endpoints and multiple integrated components, so any inconsistency in one step causes the whole test to fail.

8.2 Pipeline Architecture

To make this process possible, a pipeline structured into multiple stages was implemented using GitLab CI/CD. This choice was made because the Privacy Dashboard repository is hosted on GitLab, so the platform already provided the native automation mechanism used to validate, release, and deploy the project. The following section describes how these stages are organized and how they contribute to the validation, release and deployment of the Privacy Dashboard.

8.2.1 Conceptual Structure

Before describing the implemented pipeline in detail, it is useful to clarify the main concepts used by GitLab to structure a CI/CD workflow. The following definitions are based on the terminology used in the official GitLab documentation [32]:

- **Pipeline.** A pipeline is the central structure of the automation workflow, since it defines the overall CI/CD process. Different pipeline types can be created depending on the triggering event. In this project, the relevant ones are merge request pipelines and branch pipelines described in Section 8.2.3.
- **Job.** Inside a pipeline, the actual executable units are jobs. They are independent from each other and are executed by runners. Depending on the pipeline configuration, they can be organized to run sequentially or in parallel [33].
- **Stage.** Jobs can then be organized into stages, that define their execution order.

The Privacy Dashboard pipeline is organized into four stages, `build`, `test`, `publish`, and `deploy`, and each of them contains one job. At a higher level, these stages are reused by two different pipeline types: a merge request pipeline and a branch pipeline.

8.2.2 Pipeline Stages

Build Stage The `build` stage is implemented through the `build_image` job. Its purpose is to build the Docker image of the Privacy Dashboard starting from the current source code version. Conceptually, it acts as the first technical gate of the workflow: if the application cannot be built, no further validation, release, or deployment step should be performed.

Test Stage The `test` stage is implemented through the `test_image` job. It runs the built image in testing mode and executes the automated end-to-end test suite against a controlled local test environment:

- Smartotum authentication, API, and home DHT calls are replaced through the integration layer with local fixture data.
- Generated device rules are not pushed to the DHT, but are kept in memory for the duration of the run.
- Once the testing environment has been successfully started, the Postman collection is executed through Newman.

This stage checks the backend's actual behavior by testing the application through realistic workflows, while keeping external dependencies controlled and avoiding any reliance on third-party APIs.

Publish Stage The `publish` stage is implemented through the `push_image` job. Its purpose is to tag and publish the validated Docker image to Docker Hub, making it available as a release candidate. From a CI/CD perspective, this stage is the one that most directly expresses continuous delivery, because it turns a tested software version into a deployable image stored in a dedicated registry.

Deploy Stage The `deploy` stage is implemented through the `deploy_demo` job. It pulls the published image from Docker Hub, deletes the previous version of the Privacy Dashboard and deploys the new one with the updated image. It is the stage that completes the process by propagating the validated image to the demo environment. Unlike the testing setup, the demo environment is not meant to use local Smartotum fixtures, but to behave as the actual demonstration instance of the system.

8.2.3 Pipeline Types

During the CI/CD workflow, from the same configuration file used to manage the pipeline (`.gitlab-ci.yml`), two different pipelines are created depending on the triggering event. This overall workflow is summarized in Figure 8.1.
Figure 8.2 instead shows an example of the two pipelines after a successful execution.

Merge Request Pipeline It is generated when a merge request is created that targets the `frontend` branch. In this case, the `build` stage runs first and creates the image. If it succeeds, the `test` stage runs to verify that the proposed changes can be safely integrated without breaking the application.

Since direct commits are disabled on the `frontend` branch and changes can be merged only after the pipeline succeeds, this pipeline type represents the main continuous integration step of the project, because it validates all changes before they are integrated into the shared branch.

Frontend Branch Pipeline It is generated when a commit is present on the `frontend` branch. In this case, the pipeline executes all four stages: `build`, `test`, `publish`, and `deploy`. Conceptually, this second pipeline is the one that completes the full CI/CD cycle, since it combines validation, image publication, and final automated deployment.

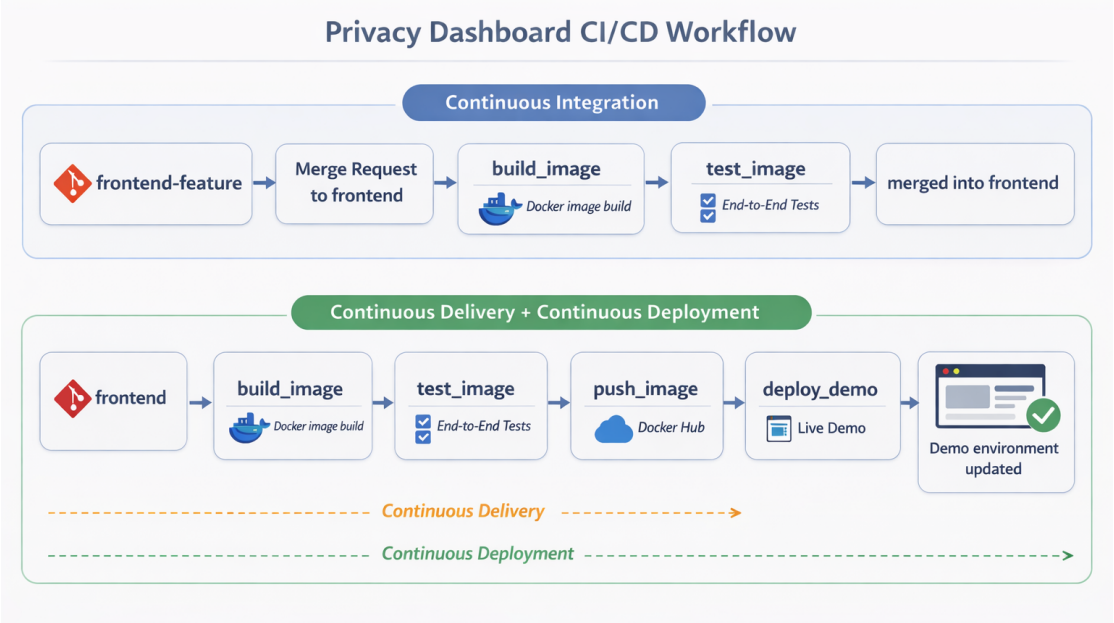


Figure 8.1: CI/CD workflow of the Privacy Dashboard

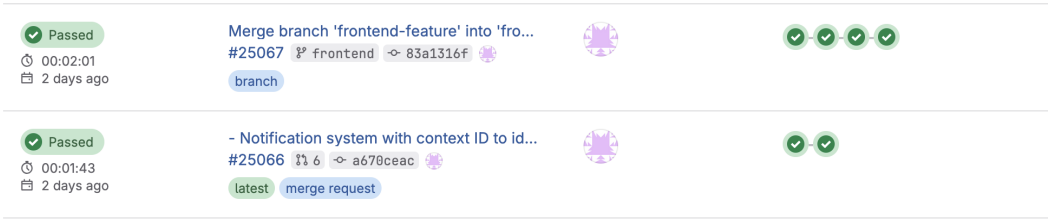


Figure 8.2: Example of successful execution of both CI/CD pipelines

8.3 Server Architecture

We briefly analyze the server architecture used to deploy the Privacy Dashboard in the demo environment. At a high level, this setup is composed of four categories of components: a virtual machine managed by a virtualization platform, a container based runtime for the application stack, a CI/CD job executor, and a reverse proxy.

8.3.1 Proxmox, VMs, Docker

Proxmox Virtual Environment is an open-source server management platform for enterprise virtualization designed to manage virtual machines and containers [34]. In this project, Proxmox is used to orchestrate a dedicated Linux virtual machine which provides isolation from the underlying host and keeps the deployment setup

easier to reproduce and manage. This virtual machine is dedicated to the Privacy Dashboard infrastructure and hosts both Docker and the GitLab Runner used by the pipeline.

Inside the virtual machine, Docker is used as the application runtime. Docker describes containers as standardized units that package an application together with its dependencies, so that the same software can run consistently across environments [35]. This is particularly suitable for the Privacy Dashboard, because the deployed application is composed of a small number of services, mainly an application container running the Privacy Dashboard, which includes both frontend and backend components, and a PostgreSQL database.

8.3.2 Privacy Dashboard Stack

Although the Privacy Dashboard is conceptually a single web application, its deployment relies on multiple coordinated runtime components, which together form the application stack. The stack is orchestrated through Docker Compose, which Docker defines as a tool for working with multi-container applications [36]. In practice, the demo Docker Compose file (`docker/docker-compose-demo.yml`) describes how the stack is composed:

- **Application container**: runs the Privacy Dashboard web application, starting from the Docker image produced by the pipeline.
- **PostgreSQL container**: provides the relational database used by the application.
- **Docker network**: allows the application container and the database container to communicate internally without exposing the database service directly.
- **Docker volume**: is used to store the database data separately from the application container.

Overall, this results in a simple single-node architecture, well suited for a demo deployment, in which application logic, persistent data, and external access are kept clearly separated.

8.3.3 GitLab Runner

GitLab Runner is the component that executes the jobs defined in a GitLab pipeline [37]. Within the server architecture of this project, the runner acts as the link between the GitLab repository and the demo infrastructure: it receives the pipeline jobs and executes the project scripts that build the image, run the temporary testing environment, publish the validated image, and deploy the demo instance.

In the current setup, the runner itself is deployed as a Docker container inside the dedicated virtual machine together with the Privacy Dashboard deploy stack. As a consequence, the same virtual machine is used both to execute the runner container and to host the Privacy Dashboard containers that are built, tested, and deployed by the pipeline. The current setup keeps the operational model simple and easy to maintain, but at the same time it also makes the runner a sensitive component, because it can directly control the build, test, and deployment workflow on the same virtual machine and it holds deployment permissions and credentials such as the Docker Hub access token.

8.3.4 Nginx Reverse Proxy

Finally, the last component required to make the Privacy Dashboard reachable through the Internet is a reverse proxy. A reverse proxy is a server that receives client requests and forwards them to one or more upstream application servers [38]. In the adopted architecture, Nginx acts as the entry point of the demo environment: it exposes a stable public endpoint and forwards incoming traffic to the application container. In the current setup, it also centralizes HTTPS configuration and TLS termination, and manages the certificates used to expose the Privacy Dashboard securely over the Internet.

Chapter 9

Conclusion

9.1 Results Achieved

As discussed in Chapter 3, this work started from the analysis of the main limitations of the original Privacy Dashboard and aimed at transforming it into a more complete and convenient platform for privacy management in smart homes.

The first major result achieved 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 policy translation protocol as implemented in the SIFIS-Home project [39], that was not only integrated, but also substantially redesigned and reimplemented 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 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. This mechanism has been extensively illustrated throughout the thesis through the recurring example of the Smartotum camera manager application.

From the point of view of Data Controllers and Data Protection Officers, the new Privacy Dashboard enables them to work with a more complete set of tools, including role management, available consent management and the creation, editing, and download of GDPR documents.

9.2 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 in mind. For this reason, further work would be required to assess its applicability to other smart home systems. Linked to Smartotum login architecture was the decision to let Smartotum credentials transit through the backend during authentication, 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, as described in Section 6.6, 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 retried 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.

A final limitation is related to the GDPR functionality for personal data export that is not yet available as a direct feature and still depends on rights requests and manual handling by the Data Controller and the DPO.

9.3 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 System 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.

Finally, a further development would be the introduction of a direct feature for exporting personal data, so that it is no longer handled only through rights requests.

With these improvements in place, together with a more complete testing suite including unit and integration tests, the Privacy Dashboard could evolve from its current demo environment into a real world platform.

Bibliography

[1] Information Commissioner's Office (ICO). *Your household smart products must respect your privacy – including your air fryer.* Accessed: 2026-03-03. June 2025. URL: https://ico.org.uk/about-the-ico/media-centre/news-and-blogs/2025/06/your-household-smart-products-must-respect-your-privacy-including-your-air-fryer/ (cit. on p. 1).

[2] *State of IoT 2025: Number of connected IoT devices growing 14% to 21.1 billion globally.* IoT Analytics. Oct. 28, 2025. URL: https://iot-analytics.com/number-connected-iot-devices/ (visited on 02/27/2026) (cit. on p. 7).

[3] 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. 8).

[4] 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. 8).

[5] Jana Arbanas, Paul Silverglate, Susanne Hupfer, Jeff Loucks, Prashant Raman, and Michael Steinhart. *Data privacy and security worries are on the rise, while trust is down.* Deloitte Insights. Sept. 5, 2023. URL: https://www.deloitte.com/us/en/insights/industry/telecommunications/connectivity-mobile-trends-survey/2023/data-privacy-and-security.html (visited on 02/27/2026) (cit. on p. 8).

[6] 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. 9).

[7] 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. 9).

[8] 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. 10).

[9] *SIFIS-Home Project website*. SIFIS-Home. URL: https://www.sifis-home.eu/#About (visited on 02/28/2026) (cit. on p. 10).

[10] *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. 10).

[11] *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 pp. 10, 11).

[12] 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. 13).

[13] 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. 14).

[14] 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. 14, 19).

[15] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures*. University of California, Irvine, 2000 (cit. on pp. 44, 45, 51).

[16] R Fielding, M Nottingham, and J Reschke. *RFC 9110 HTTP Semantics*. 2022 (cit. on p. 44).

[17] 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. 47, 48).

[18] Thomas M Connolly and Carolyn E Begg. *Database systems: a practical approach to design, implementation, and management*. Pearson Education, 2005 (cit. on p. 49).

[19] Jim Gray and Andreas Reuter. *Transaction processing: concepts and techniques*. Elsevier, 1992 (cit. on p. 49).

[20] Michael Stonebraker and Lawrence A Rowe. «The design of Postgres». In: *ACM Sigmod Record* 15.2 (1986), pp. 340–355 (cit. on p. 49).

[21] Lloyd Shirley Jerry Welch James McDonald. *FROM MONOLITH TO MICROSERVICES: A SPRING BOOT–DRIVEN APPROACH TO MODULAR BACK-END SYSTEMS* (cit. on p. 50).

[22] 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. 52).

[23] Miguel Grinberg. *Flask web development*. " O'Reilly Media, Inc.", 2018 (cit. on p. 52).

[24] 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. 53).

[25] Pete Hunt. *Why did we build React?* June 5, 2013. URL: `https://legacy.reactjs.org/blog/2013/06/05/why-react.html` (visited on 03/01/2026) (cit. on p. 55).

[26] 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. 55).

[27] Michael Jones, John Bradley, and Nat Sakimura. *JSON Web Token (JWT)*. RFC 7519. RFC Editor, May 2015. DOI: `10.17487/RFC7519`. URL: `https://www.rfc-editor.org/rfc/rfc7519` (visited on 03/04/2026) (cit. on p. 68).

[28] *JSON Web Token Introduction - jwt.io*. Auth0. URL: `https://www.jwt.io/introduction` (visited on 03/04/2026) (cit. on pp. 69, 71).

[29] Klaus Wehrle, Stefan Götz, and Simon Rieche. «Distributed Hash Tables». In: *P2P Systems and Applications*. Ed. by Ralf Steinmetz and Klaus Wehrle. Vol. 3485. Lecture Notes in Computer Science. Springer, 2005, pp. 79–93 (cit. on pp. 74, 75).

[30] *What is CI/CD?* Red Hat. URL: `https://www.redhat.com/en/topics/devops/what-is-ci-cd` (visited on 03/16/2026) (cit. on pp. 91, 92).

[31] *What is End-to-End (E2E) Testing?* IBM. URL: https://www.ibm.com/think/topics/end-to-end-testing (visited on 03/16/2026) (cit. on p. 92).

[32] *Pipelines*. GitLab. URL: https://docs.gitlab.com/ci/pipelines/ (visited on 03/09/2026) (cit. on p. 95).

[33] *Jobs*. GitLab. URL: https://docs.gitlab.com/ci/jobs/ (visited on 03/16/2026) (cit. on p. 95).

[34] *Proxmox VE Overview*. Proxmox. URL: https://www.proxmox.com/en/proxmox-virtual-environment/overview (visited on 03/09/2026) (cit. on p. 97).

[35] *Docker Docs – What is a container?* Docker. URL: https://docs.docker.com/get-started/docker-concepts/the-basics/what-is-a-container/ (visited on 03/09/2026) (cit. on p. 98).

[36] *Docker Docs – Compose application model*. Docker. URL: https://docs.docker.com/compose/intro/compose-application-model/ (visited on 03/09/2026) (cit. on p. 98).

[37] *GitLab Docs – GitLab Runner*. GitLab. URL: https://docs.gitlab.com/runner/ (visited on 03/09/2026) (cit. on p. 98).

[38] *NGINX Docs – Reverse Proxy*. NGINX. URL: https://docs.nginx.com/nginx/admin-guide/web-server/reverse-proxy/ (visited on 03/09/2026) (cit. on p. 99).

[39] SIFIS-Home. *Policy Translation Point*. GitHub repository. 2025. URL: https://github.com/sifis-home/policy-translation-point (visited on 03/14/2026) (cit. on p. 100).