Politecnico di Torino

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

A.Y. 2025/2026

Graduation Session March 2026

Privacy Dashboard: Developing the Policy Translation Point Application

Supervisors:
Luca Ardito

Candidate:
Eric Yugo Hioki

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, this project, in collaboration with two other colleagues, aims 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 domotic system, and the Distributed Hash Table (DHT), providing a centralized platform to manage consents, GDPR-related documents for the apps installed in the Smart Home, and assist both end users and data controllers or 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 by showing an overview of the existing software and the analysis 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, focusing on the implementation of the Policy Translation Point (PTP). This section details the logic required to bridge the semantic gap between high-level user preferences and machine-enforceable XACML policies, utilizing Semantic Web technologies for context awareness and Formal Verification methods to ensure rule safety. Finally, a retrospective of our work is conducted, in order to examine the constraints of our software and to discuss how it could be further improved in the future.

Table of Contents

List of Tables VII

List of Figures VIII

Glossary X

1 Introduction 1
- 1.1 Context . . . 1
- 1.2 Project Goals . . . 2
- 1.3 Thesis Goals . . . 4
- 1.4 Thesis Structure . . . 5

2 Background and State of the Art 7
- 2.1 Background . . . 7
  - 2.1.1 Smart Home and IoT . . . 7
  - 2.1.2 Privacy Threats in Smart Homes . . . 9
  - 2.1.3 Smartotum and the SIFIS-Home Framework . . . 10
  - 2.1.4 GDPR (General Data Protection Regulation) . . . 11
  - 2.1.5 Access Control Policies . . . 13
  - 2.1.6 The Semantic Web . . . 15
  - 2.1.7 Formal Verification Methods . . . 16
- 2.2 Legacy System Analysis . . . 19
  - 2.2.1 Overview of the Existing Policy Translation Point . . . 19
  - 2.2.2 Limitations of the Existing Policy Translation Point . . . 23

3 System Features and User Interface 25
- 3.1 User Authentication and Account Access . . . 25
- 3.2 Smart Home Selection . . . 26
- 3.3 Policy Overview . . . 26
- 3.4 Rule Creation . . . 28
- 3.5 Policy Record Schema . . . 28

3.6 Conflict Detection 30
3.7 Rule Download 30
3.8 Rule Deletion 30
3.9 End-to-End Usage Scenario 32

4 System Architecture and Technologies **33**
4.1 High-Level Architecture 33
4.2 Database Technologies 36
4.2.1 The Relational Data Model 36
4.2.2 PostgreSQL 38
4.3 Backend Technologies 39
4.3.1 Monolithic Architecture and the Controller-Service-Repository Pattern 39
4.3.2 Flask 41
4.4 Frontend Technologies 42
4.4.1 Single Page Application Architecture 42
4.4.2 React 43

5 Implementation **46**
5.1 Backend Implementation 46
5.1.1 Rule Ingestion and Sanitization 46
5.1.2 The Role of OWL Files 48
5.1.3 Hydration and Dehydration 49
5.1.4 Implementing Semantic Colored Petri Nets 50
5.1.5 XACML Generation 52
5.1.6 DHT Integration 52
5.2 Frontend Implementation 54
5.2.1 Component Architecture 54
5.2.2 State Management and Validation 55
5.2.3 Error Handling 56
5.2.4 Payload Construction 56
5.3 End-to-end Testing of the Policy Translation Point 58
5.3.1 Sequential Test Pipeline and Variable Propagation 58
5.3.2 Rule Creation and Schema Validation 59
5.3.3 DHT Synchronization and Rule Persistence 59
5.3.4 Conflict Seeding Strategy 60
5.3.5 XACML Translation and Binary Validation 61
5.3.6 Error Boundaries 61

6 Conclusion 62
6.1 Results Achieved 62
6.2 Limitations 63
6.3 Future Work 64

Bibliography 66

List of Tables

3.1 Policy Record Schema: Mapping UI fields to the underlying data model. 29

5.1 Data dictionary for the hydration/dehydration process. 50
5.2 Sequential test pipeline for the Policy Translation Point API. 58

List of Figures

2.1 The three layer model of IoT architecture. 8
2.2 Example of an OWL ontology graph illustrating relationships between entities. Adapted from [13]. 16
2.3 Structure of a basic Petri Net. 18
2.4 SCPN topology modeling policy interactions for conflict detection. Adapted from [18]. 19
2.5 Policy list view in the legacy PTP. 20
2.6 Action service selection view in the legacy PTP. 21
2.7 Trigger service selection view in the legacy PTP. 22
2.8 Rule creation form in the legacy PTP. 22
2.9 Conflict detection report in the legacy PTP. 23

3.1 Authentication interface. 26
3.2 Smart home selection interface. 27
3.3 Overview of created high-level policies with action buttons. 27
3.4 Rule creation editor with dynamic capability selection. 28
3.5 Policy verification report displaying categorized conflicts (inconsistencies and redundancies) with inline resolution actions. 31
3.6 Deletion confirmation dialog. 31

4.1 The Client-Server Architecture. 35
4.2 Example of an HTTP request and response. 35
4.3 Example of cross-reference between two tables. 37
4.4 Example of virtual DOM compared to browser's DOM. 44
4.5 Example of update to virtual DOM reflected to browser's DOM. . . 45

Glossary

PTP
: Policy Translation Point

DHT
: Distributed Hash Table

XACML
: eXtensible Access Control Markup Language

XML
: Extensible Markup Language

SCPN
: Semantic Colored Petri Nets

CPN
: Colored Petri Nets

UI
: User Interface

PEP
: Policy Enforcement Point

OWL
: Web Ontology Language

UX
: User Experience

RDF
: Resource Description Framework

OWA
: Open World Assumption

DL
: Description Logic

JAXB
: Jakarta XML Binding

PCB
: Polychlorinated Biphenyls

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

MVCC
: Multi-Version Concurrency Control

REST
: Representational State Transfer

HTML
: HyperText Markup Language

API
: Application Programming Interface

JSON
: JavaScript Object Notation

WSGI
: Web Server Gateway Interface

MPA
: Multi-Page Application

SPA
: Single Page Application

DOM
: Document Object Model

URI
: Uniform Resource Identifier

UUID
: Universally Unique Identifier

IRI
: Internationalized Resource Identifier

ORM
: Object–Relational Mapping

CI
: Continuous Integration

CD
: Continuous Delivery

SIFIS-Home
: Secure Interoperable Full-Stack Internet of Things for Smart Home

NSSD
: Not-So-Smart Devices

RBAC
: Role-Based Access Control

ABAC
: Attribute-Based Access Control

PAP
: Policy Administration Point

PDP
: Policy Decision Point

Chapter 1

Introduction

The fast evolution of IoT (Internet of Things) technologies has changed the concept of what a home is. It has transformed from a passive space into a *Smart Home*, an active system of intelligent devices. While these technologies offer new levels of convenience, energy efficiency, and automation, they also create a significant amount of data collection that raises concerns about privacy.

This thesis explores the important link between this growth in technology and the strict data protection rules set by the European Union, the GDPR (General Data Protection Regulation). It outlines the design and implementation of a Privacy Dashboard and PTP (Policy Translation Point) as part of the SIFIS-Home project. My contribution on this project aims to connect high-level user goals with the low-level, machine-enforceable rules needed to manage various IoT devices. By migrating old prototypes into a React and Flask-based architecture, this thesis helps create a strong, concise, scalable solution for handling privacy in today's Smart Homes.

1.1 Context

Connected devices have quietly integrated themselves into the typical aspects of our daily lives, with thermostats learning our habits, voice assistants responding to questions from the kitchen, and cameras watching our activity for safety and convenience. However, all of these convenient services are made possible by the continuous flow of our personal data. This has led to a home environment where data collection is everywhere, invisible to the user, and governed by a complex web of device manufacturers, cloud service providers, and local app developers. This has led to a situation where privacy in the Smart Home environment is both highly important and highly problematic. The same mechanisms that make the service valuable to the user also provide multiple opportunities for the user's personal

information to be compromised and misused.

The regulatory and ethical response to this reality is strong and explicit. The GDPR, the EU regulation that regulates the processing of personal data, imposes strict obligations on any organization that collects and processes personal data, ensuring that data subjects have specific rights such as access, erasure, purpose limitation, and minimization. Nevertheless, these obligations are not ideal; they have direct technical implications for logging, consent, transparency, and the ability to prove compliance.

Apart from the legal aspect, there are practical issues with Smart Homes that make privacy a challenge to handle in real life. The devices in a Smart Home are likely to have a great variety in terms of functionality, vendors, and communication patterns. Some devices are cloud-based, while others are local. Many of them are connected to third-party platforms. From the user's perspective, this is not an issue, but it has a direct impact on how their data is managed. Handling privacy in such a scenario is more than just a matter of legal compliance.

The SIFIS-Home project provides the technical context in which this thesis is placed. It is a European project aiming at securing, as well as making more privacy-friendly, the Smart Home technology stack by integrating system architecture, developer tools, and privacy-friendly analytics into a unified ecosystem. The project states that privacy should be a concern for architecture: Instead of adding compliance as an afterthought, tools for enforcement, transparency, and guidance should be integrated directly into the framework that runs on home devices.

This thesis is part of a group student project conducted in the context of SIFIS-Home. The group project is intended to extend and integrate the existing SIFIS-Home components in a way that privacy management can be operated and assessed more easily in a real-world setting. In this chapter, we introduce the shared context and objectives of the project; in the following chapters, the scope of the implementation and evaluation will be clarified to indicate which aspects were worked on jointly and which aspects are the individual contributions of the students.

1.2 Project Goals

The collaborative project in which this thesis was part of has the goal of improving and modernizing the privacy management infrastructure originally developed in the SIFIS-Home initiative. Although the conceptual foundations, architectural guidelines, and reference implementations were already defined in the project deliverables, the available prototypes needed to be consolidated, refactored, and functionally extended in order to provide a coherent and maintainable software solution.

One of the main objectives of the project was the analysis of the existing applications, namely the Privacy Dashboard and the PTP, in order to identify architectural limitations, technological constraints, and missing functionalities. The original implementations, developed as separate Java-based components, represented different phases of the research project and were not intended to work as a single platform. Most importantly, the prototypes were working in a standalone manner, without functional integration with the core components of the SIFIS-Home infrastructure. They were not able to dynamically fetch device information from the Smartotum, a domotic framework, nor were they able to publish enforcement policies to the DHT (Distributed Hash Table). This situation made their integration, maintenance, and further development unnecessarily complex.

Based on this analysis, the objective of the team was to redesign the system architecture around a single platform. The objective was not only to move code from one language to another but also to achieve a better separation between backend and frontend, facilitating a clearer API, better modularity, and easier experimentation. The choice of a Python-based backend with the Flask framework and a React frontend was driven by the need for rapid development, readability, and maintainability.

Moreover, the project focused on usability and understandability. Privacy management tools are only useful if they can be easily comprehended and used by non-expert users. In this respect, the project tried to enhance the support for the representation, visualization, and management of privacy artefacts, such as policies, consents, and obligations, in the application. Special emphasis was placed on ensuring that high-level user interactions are kept consistent with the technical enforcement mechanisms specified in the SIFIS-Home architecture.

Finally, the project tried to ensure that the re-engineered system is consistent with the SIFIS-Home ecosystem. This includes compatibility with the architectural principles, developer guidelines, and privacy-aware analytics components, as defined in the official deliverables. Instead of developing a self-contained research prototype, the goal was to develop a system that could realistically integrate into the planned Smart Home infrastructure and enable further research and development.

In conclusion, the shared objective of the group was to develop a privacy-aware Smart Home management platform from a set of loosely interconnected research prototypes, while maintaining the original vision of SIFIS-Home and enhancing its usability and consistency. While foundational decisions such as the choice of the technology stack and the general system architecture were taken collectively, each team member was individually responsible for a distinct area of the platform:

- One colleague focused on the core backend infrastructure, including the ORM and repository layer, the controller layer with request validation and serialization, token-based authentication, end-to-end testing, and frontend components

for GDPR document management such as Privacy Notices and Privacy Impact Assessments.

- Another colleague handled the integration with the Smartotum domotic framework, including the dual authentication system, the synchronization architecture that keeps the local database aligned with the DHT, the Marketplace module, and the CI/CD pipelines for automated deployment.

- My personal contribution, which constitutes the subject of this thesis, covered the entire Policy Translation Point: the semantic policy translation engine, the formal verification layer based on SCPN, the XACML 3.0 policy generation pipeline, and the bidirectional rule publishing and synchronization with the DHT.

1.3 Thesis Goals

Having established this division of responsibilities, this section narrows the focus to the specific goals of this thesis: the design, implementation, and integration of the Policy Translation Point. The PTP is tasked with filling the semantic gap that exists between high-level user preferences and the low-level enforcement mechanisms of the SIFIS-Home ecosystem.

The Policy Translation Point is the mediator between the user-friendly dashboard and the underlying IoT infrastructure. The PTP's task is to take high-level abstract rules, validate them based on the current home topology, and then translate them into machine-enforceable instructions. To accomplish this, my contribution was focused on four specific objectives:

1. **Architectural Consolidation and Logic Integration**

   In order to support the project's aim of having a unified Python-based stack, my first objective was to migrate the legacy Java-based translation logic and integrate it directly into the Privacy Dashboard application. This involved designing an internal architecture for the application that is modular, with the PTP being a distinct logical component of the application's Flask backend. By integrating the translation engine directly into the main application, the system enjoys lower latency and easier deployment, while still making its capabilities available to the React frontend through a special set of RESTful APIs.

2. **Semantic Policy Translation**

   The functional requirement of the thesis is the implementation of the translation pipeline. My objective was to use Semantic Web technologies, in particular,

OWL ontologies, to translate high-level concepts, such as "Kitchen," "Video Recording", into concrete XACML (eXtensible Access Control Markup Language) elements. This step ensures that a given user preference is properly translated to include all relevant devices, taking into account their special capabilities and constraints.

3. **Formal Verification and Conflict Detection**

   One of the essential safety requirements is to ensure that the user-defined rules do not conflict with each other or cause instabilities in the system. My goal was to develop a Conflict Detection Layer that would examine policies before they are applied. By applying Formal Verification techniques, and more specifically, SCPN (Semantic Colored Petri Nets), the system is able to mathematically represent the interaction of rules to:

   - Detect Conflicts: Rules that both allow and deny the same action at the same time.
   - Detect Redundancies: Multiple rules that enforce the same constraint.

   This phase ensures that "unsafe" or ambiguous policies are reported to the user instead of being silently applied.

4. **Standardization and Compliance**

   Lastly, the implementation was intended to strictly follow the SIFIS-Home architectural guidelines. This includes proper usage of the SIFIS-Home Ontology for data modeling and the generation of standard XACML 3.0 policies that can be processed by any XACML 3.0-compliant PEP (Policy Enforcement Point). This ensures that the PTP enables interoperability, allowing the privacy rules to be enforced on any device, regardless of the manufacturer.

   Through the completion of these goals, my individual work seeks to provide the "technical linchpin" that transforms the Privacy Dashboard into an operational control center, capable of making real, tangible changes to the behavior of the Smart Home.

1.4 Thesis Structure

This thesis consists of six chapters, designed to take the reader from the general context to the technical implementation. In view of the collaborative nature of this project, each following chapter description will clearly outline the boundaries of the work, separating the collaborative efforts of the team from the individual work presented in this thesis.

- **Chapter 1: Introduction**

  This chapter introduces the research context, describing privacy issues in the IoT environment and the legal framework of the GDPR. It defines the general objectives of the Privacy Dashboard project and describes the specific objectives assigned to the Policy Translation Point.

- **Chapter 2: Background and State of the Art**

  This chapter provides the theoretical and technical background of the work. It describes in detail the SIFIS-Home ecosystem, the legal requirements of the GDPR, and the specific technologies used, such as XACML, OWL, and SCPN. Additionally, it describes an analysis of the legacy Java-based application, pointing out the severe limitations that led to the re-engineering effort.

- **Chapter 3: System Features and User Interface**

  This chapter describes the main user interactions supported by the PTP, following the typical lifecycle of a policy from visualization to creation, verification, translation, and deletion. It presents the functional capabilities of the redesigned application, demonstrating the user experience from authentication and home selection to policy management.

- **Chapter 4: System Architecture and Technologies**

  This chapter describes the high-level architecture of the redesigned system, explaining the move to the new Python/Flask and React technology stack. It covers the database technologies, the backend and frontend architectural patterns, and the communication model between the client and server components.

- **Chapter 5: Implementation**

  This chapter represents the heart of the individual research effort. It goes into the technical details of the PTP backend implementation, describing the algorithms and pipelines employed to process high-level policies, conduct formal verification with SCPN, and translate semantic policies into machine-enforceable XACML 3.0 policies.

- **Chapter 6: Conclusion**

  The final chapter summarizes the outcome of the project, discussing the degree to which the original goals have been fulfilled. It acknowledges the current limitations and discusses possible future work on the PTP.

Chapter 2

Background and State of the Art

2.1 Background

2.1.1 Smart Home and IoT

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

The adoption of such class of smart devices is playing a key role in the modernization and digital transformation of institutions, businesses and industries, and is becoming more significant in private households too: indeed, it is estimated that the number of IoT devices all over the world will almost be tripled by 2030, increasing from 8.74 billion in 2020 to more than 25.4 billion in 2030 [1]. Among said sectors, we focus on the usage of IoT within domestic environments, which leads to the concept of *Smart Home*.

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

between devices and users can happen over the cloud and allow home owners to easily monitor and rule their Smart Home from a centralized software accessible from a device connected to the Internet, such as a smartphone or a Personal Computer. Such kind of system can be implemented using a three-layer architecture: the perception layer, the network layer, and the application layer [2]. The perception layer is the bottom layer where different home appliances collect data from the surrounding environment. Collected data are sent to the network layer through a Smart Home gateway. The network layer, in turn, sends the data in the cloud to the application layer, which is the only one the user directly interacts with, using the dedicated software mentioned above. In many real-world implementations, the application layer relies on cloud services as a central point of aggregation and accessibility of all the information regarding the Smart Home. A graphical representation of the three-layer architecture is illustrated in Figure 2.1.

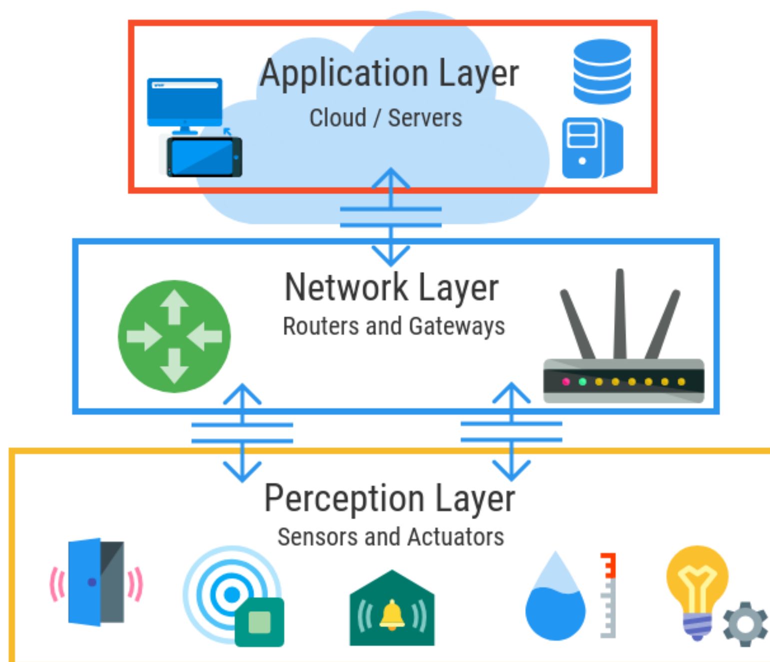

Figure 2.1: The three layer model of IoT architecture.

This type of architecture enables Smart Homes to make autonomous decisions without owner manual intervention and provide assistive and personalized services, which ultimately aims to the main goal of improving the owner's quality of life [3]. For instance, when the homeowner leaves, the Smart Home can automatically turn off lights and enable security alerts if unusual activity is detected.

2.1.2 Privacy Threats in Smart Homes

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

Data Threats The fact that these devices are a continuous source of personal data collected inside our homes makes them a valuable target for attackers. A Smart Home system might be victim of different types of threats against data, which the ENISA (European Union Agency for Cybersecurity) has classified and defined as follows [4]:

- **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 [5].

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 [6].

2.1.3 Smartotum and the SIFIS-Home Framework

SIFIS-Home The *SIFIS-Home* Project was created to address these concerns by developing a secure-by-design and consistent software framework across all stack levels [7]. From a design standpoint the SIFIS-Home Framework is organized into dedicated frameworks (*i.e.* Smart Device, NSSD (Not-So-Smart Devices), Application, Cloud, Development Tools), each comprising the set of software components executed on a specific platform [8].

The overall architecture follows a microservices approach based on Docker containers: most of the components can expose a RESTful API when needed for integration. Instead of relying on the cloud as the single aggregation point of Smart Home information, SIFIS-Home relies on a distributed data plane deployed inside the home network, enabling components to exchange information without a central broker. This layer is implemented through a DHT (Distributed Hash Table) that supports both volatile messages (delivered to running applications) and persistent messages stored locally to preserve critical data such as settings and policies across reboots. When remote access is required, SIFIS-Home includes a bridging component that can exchange messages and configurations between the DHT and a cloud-side platform. This design makes it possible to keep the Smart Home state and policies primarily within the domestic environment, while still enabling remote interaction when needed [8].

Smartotum Smartotum originated as a technology transfer initiative that brings the SIFIS-Home vision into a deployable home automation system [9]. 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.[9]

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

2.1.4 GDPR (General Data Protection Regulation)

The GDPR (General Data Protection Regulation), adopted by the European Union in April 2016 and enforceable since May 2018, 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 Access Control Policies

The increasing need to define privacy preferences in a heterogeneous IoT environment makes it often inadequate to rely solely on Access Control Lists or RBAC (Role-Based Access Control) systems, which do not have the necessary level of granularity to define context-dependent rules, such as allowing camera access only when the user is at home. To address this limitation, the SIFIS-Home system adopts ABAC (Attribute-Based Access Control), directly implemented through the XACML (eXtensible Access Control Markup Language), an OASIS standard that defines both a declarative XML policy language and a reference processing architecture [10]. The fact that XACML separates authorization logic from application code gives it a special status that makes it possible to centrally manage, update, and analyze privacy rules without modifying the devices themselves.

Processing Model

The XACML specification defines a reference architecture where different components are responsible for distinct phases of the access control lifecycle. The PAP (Policy Administration Point) is the component where policies are authored and

managed; the PDP (Policy Decision Point) evaluates incoming access requests against those policies and produces an authorization decision; and the PEP enforces the resulting decision on the target resource. In the context of this project, the PTP fulfills the role of a PAP, as it allows end users to author high-level privacy rules and translates them into compliant XACML policies for downstream evaluation and enforcement.

Core Structural Components

The XACML data model is hierarchical, designed to provide a manageable way to organize complex sets of rules. Specifically, the core structure used in this project follows a three-level hierarchy: the *PolicySet*, which is the top-level container that groups policies according to specific domains such as "Video Recording"; the *Policy*, which is the basic unit of administration; and the *Rule*, which is the atomic unit of logic that defines a specific condition and its corresponding effect [10].

The Target: Defining Scope

One of the most important concepts in XACML is that of the *Target*, which acts as a filter that defines where a Policy or a Rule is applicable. Before the system proceeds to analyze the complex logic of a rule, it first checks the Target to determine if the rule is even relevant to the current request, matching attributes in four major categories: *Subject* (who is requesting access), *Resource* (which device or data is being accessed), *Action* (what operation is being performed), and *Environment* (contextual conditions such as time or location) [10]. In the context of the PTP, these Targets are constructed based on the particular devices and locations chosen by the user in the frontend, ensuring that a rule scoped to the kitchen is never accidentally applied to devices in the bedroom.

Rules and Combining Algorithms

After a Target has been matched and the Rules have been evaluated, there is a need for a resolution mechanism. A Rule is essentially a boolean expression that returns one of four possible effects: *Permit*, *Deny*, *NotApplicable*, or *Indeterminate*. However, because a Policy may contain multiple Rules whose individual decisions may contradict each other, XACML defines *Combining Algorithms* to resolve these contradictions deterministically. The PTP uses two complementary algorithms depending on the policy type: *permit-unless-deny*, which grants access by default unless an explicit "Deny" rule is matched, following a blacklist approach well-suited for general privacy preferences; and *deny-unless-permit*, which denies access by default unless an explicit "Permit" rule is matched, following a whitelist approach that enforces stricter control where needed [10].

2.1.6 The Semantic Web

Ontologies and Web Ontology Language

While the RDF (Resource Description Framework) structure itself is sufficient for the basic graph syntax necessary for resource relationships, it is often not enough on its own because it lacks the formal semantic level necessary for defining the logic and expressiveness of a domain. To fill this gap, we rely on ontologies. Based on Gruber's classic definition of an ontology as "an explicit specification of a conceptualization," its special status offers a structured vocabulary that facilitates information exchange between different systems [11].

This work uses the OWL (Web Ontology Language), a W3C recommendation created to express complex knowledge by enabling the definition of Classes as sets of individuals, Individuals as actual objects from a domain, and Properties that define relationships between two individuals or connect an individual to a specific data value [12]. Unlike a database schema, an OWL ontology supports the OWA (Open World Assumption), where the lack of information does not entail its falsehood, a key aspect for dynamic Smart Home systems where devices can dynamically join and leave the network, allowing the model to still be valid even when incomplete information about the devices is available [13].

Semantic Reasoning

A closer examination of OWL reveals its foundation in DL (Description Logics), which provides a mathematical basis for the application of Semantic Reasoners, such as HermiT or Pellet, to reason with the logical rules expressed in the ontology [11]. Specifically, we examined the role of these reasoning engines in facilitating critical tasks such as consistency checking, which ensures that the ontology does not contain any logical inconsistencies, such as an individual being both a PublicSpace and a PrivateSpace if those classes are disjoint, as well as inference and classification, which enable the automatic derivation of new knowledge from existing facts. For instance, if the ontology states that any device with a lens is a Camera, and a new device instance is added with the property stating it has a lens, the reasoner will automatically classify it as a Camera, even if it is not explicitly stated as such.

Example

As illustrated in Figure 2.2, the system employs three different types of relationships to establish entities, which connect specific entities to more general ones via hierarchical "is-a" relationships analogous to software inheritance, establish locations via geographical relationships such as "inhabits," and describe interactions between classes via transversal relationships. From the diagram, it is very easy to trace a

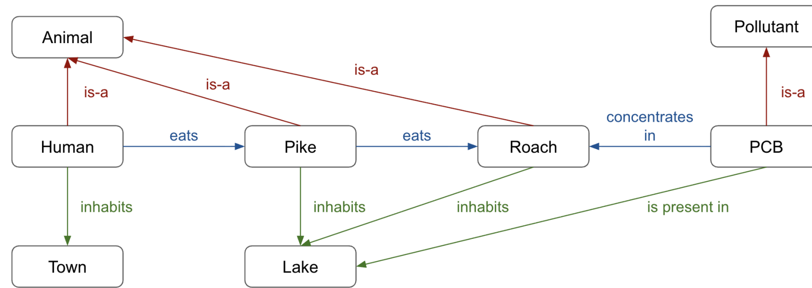

Figure 2.2: Example of an OWL ontology graph illustrating relationships between entities. Adapted from [13].

sequence of events where PCB (Polychlorinated Biphenyls) entering a lake becomes part of the food chain, which means that a Human consuming a Pike from that lake will surely be affected. The key benefit of applying an ontology, therefore, is that it enables the machine to make this logical inference; by employing a Reasoner, the system automatically traces these paths to discover hidden facts, such as a health hazard, without the need for a human to manually program all possible outcomes[13].

Owlready2

The integration of these semantic functionalities within a contemporary software framework will be discussed, utilizing the Owlready2 library to mitigate the shortcomings of conventional APIs in processing RDF triples as mere data. The incorporation of the paradigm of Ontology-Oriented Programming by Owlready2 gives it the capability to map OWL classes and individuals directly to Python classes and objects, making it possible a Python based application to manipulate the ontology seamlessly [14]. In ensuring that OWL Classes and Properties are made accessible as conventional Python attributes and enabling Individuals to be created through conventional constructors, this abstraction layer plays a pivotal role in the "Hybrid" architecture of the Privacy Dashboard, effortlessly connecting the static, high-performance world of relational databases with the dynamic, inferential world of the Semantic Web.

2.1.7 Formal Verification Methods

In a Smart Home setting, the interdependence of user-defined privacy rules, such that they do not operate alone but as part of a dynamic environment of sensors, actuators, and other active policies, makes it inadequate to rely on an individual rule

verification. A rule may appear valid when considered in isolation, such as allowing the turning on of a light when motion is detected, but may lead to redundancy or inconsistencies when considered in the context of other rules. In this case, to avoid such runtime errors, we apply Formal Verification, using mathematically precise methods to verify the correctness of a system with respect to a given specification [15].

Taxonomy of Policy Conflicts

In the context of the SIFIS-Home architecture, formal verification is employed to identify types of policy anomalies, which are categorized into two main groups as defined in the project's initial design specifications. Specifically, we considered inconsistency when two rules are triggered by the same condition but prescribe opposite actions, such as allowing and denying video recording simultaneously for the same camera, which represents a logical contradiction in which the system is placed in two mutually exclusive states. Moreover, a rule is considered redundant if it achieves an effect already ensured by another existing rule, and thus the identification of such redundancies is of vital importance for ensuring system performance and minimizing the computational burden on the PEP (Policy Enforcement Point).

Petri Nets: Modeling Concurrency and State

To detect these conflicts, we utilized Petri Nets, a graphical modeling language developed by Carl Adam Petri in 1962. They are particularly suited for modeling distributed systems such as the IoT because they are designed to model concurrency, asynchronicity, and causal dependencies between events that are difficult to model using traditional state machines [Murata, 1989].

This section begins with a description of the general structure of a Petri Net, which is a directed bipartite graph with four basic elements:

- **Places:** Represent conditions or states, modeled visually as circles.
- **Transitions:** Represent events or actions, modeled visually as bars or rectangles.
- **Arcs:** Directed lines connecting Places to Transitions (or vice versa), dictating the flow of the system.
- **Tokens:** Markers residing in Places that represent the current state of the system.

The dynamic execution process is simulated when a Transition "fires," moving these tokens from input to output Places [16]. Figure 2.3 shows a simple Petri Net

where Places, Transitions, Arcs, and Tokens interact to model a sequence of state changes.

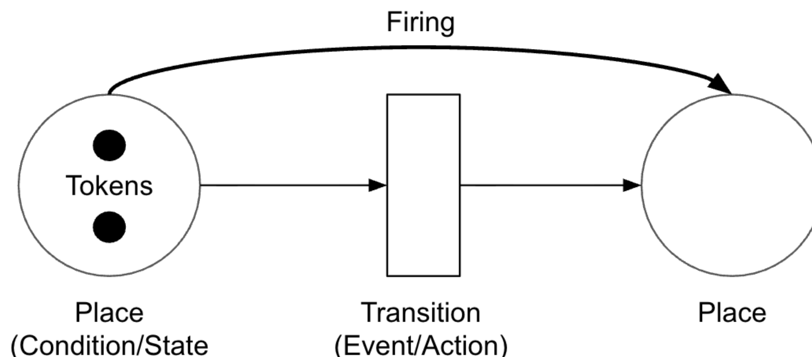

Figure 2.3: Structure of a basic Petri Net.

In formal terms, the definition of a Petri Net structure as a tuple $N = (P, T, A, W)$, where P represents Places, T Transitions, A Arcs, and W the weight of those arcs, grants it a central role that enables us to represent a privacy rule not only as static text, but as a computable logic flow. In this model, the *Trigger* serves as a pre-condition for a transition, and the *Action* serves as a post-condition [15].

Application of Petri Nets to Conflict Detection

To ensure that user-defined privacy rules do not cause runtime conflicts in a dynamic smart environment, the PTP utilizes a SCPN (Semantic Colored Petri Net) formalism [17]. While standard Petri nets excel at describing non-deterministic concurrent environments, CPNs (Colored Petri Nets) extend this capability by integrating the strengths of high-level programming languages.

In the SCPN model adopted for this architecture, tokens do not merely act as blank state markers; they assume different semantic "colors" as they move through the net. These colors represent specific classes extracted from the SIFIS-Home ontology. This semantic layer allows the PTP to infer rich contextual information during the simulation of the net, providing the necessary logic to discriminate between safe and problematic high-level policies [18].

To analyze these rules, the PTP translates the high-level policies into a specific SCPN topology. The mapping is structured as follows:

- **Places:** Triggers and actions from the high-level policies are modeled as distinct places in the net (e.g., $P_{trigger}$ and P_{action}). Each place is labeled with

its corresponding OWL class extracted from the ontology. Action places can be directly reused by different policies that share the same output behavior.

- **Policy Transitions (*TPolicy*):** These transitions serve as the direct logical link connecting the trigger place to the action place of the exact same policy.
- **Copy Transitions (*TCopy*):** When a single trigger is shared among multiple distinct policies, the system duplicates the associated places. A dedicated *TCopy* transition connects them; when a token arrives at the original trigger place, the *TCopy* transition replicates that token into each copied place, allowing the concurrent policies to evaluate simultaneously.
- **Activate Transitions (*TActivate*):** These transitions map chained events. If the action of one high-level policy serves as the trigger for another, a *TActivate* transition connects the initial P_{action} to the subsequent $P_{trigger}$.

An example of the resulting topology is depicted in Figure 2.4, where triggers and actions from multiple high-level policies are connected through the different transition types described above.

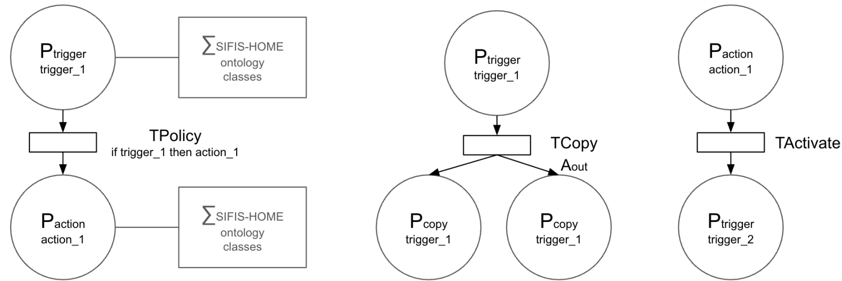

Figure 2.4: SCPN topology modeling policy interactions for conflict detection. Adapted from [18].

2.2 Legacy System Analysis

2.2.1 Overview of the Existing Policy Translation Point

The software we designed and developed during this project was not built from scratch, but rather consisted of refactoring an existing two web applications; therefore, the first step to perform was to understand its architecture and analyze its functionalities, leaving the discussion of its structural limitations to the subsequent section. The fact that, originally, we had two distinct, isolated web applications,

the Privacy Dashboard and the PTP, demands a targeted analysis. In this scenario, since my personal contribution to the unified project revolves specifically around the implementation of the Policy Translation Point into the new app, this section exclusively addresses the architecture and functionalities of this original app.

The original PTP was initially developed as a research prototype within the context of the SIFIS-Home project, designed specifically to bridge the semantic gap between high-level user preferences and low-level access control. The deployment of this original PTP as a monolithic Java application, developed with the Spring Boot framework, structured the legacy system as a dense environment of tightly coupled components. The system's foundation, which combined a simple Angular frontend developed in TypeScript with a Java backend and a MySQL persistence layer, relied heavily on the OWL API for ontology processing and the HermiT reasoner for inference, keeping ontologies entirely in memory to serialize the generated low-level XACML policies using JAXB (Jakarta XML Binding) into downloadable ZIP archives.

In this context, this part of the discussion starts by showing an overview of the old user interface, which, from a user's perspective, had a small but highly focused feature set centered around policy lifecycle management. In particular, the primary interface enabled users to access a comprehensive list of their existing policies, providing a centralized view to monitor and manage the high-level rules previously authored and stored within the MySQL database, as shown in Figure 2.5. In this view the user could also perform actions, such as translate and download the XACML policies, delete policies and check if a certain policy has conflicts.

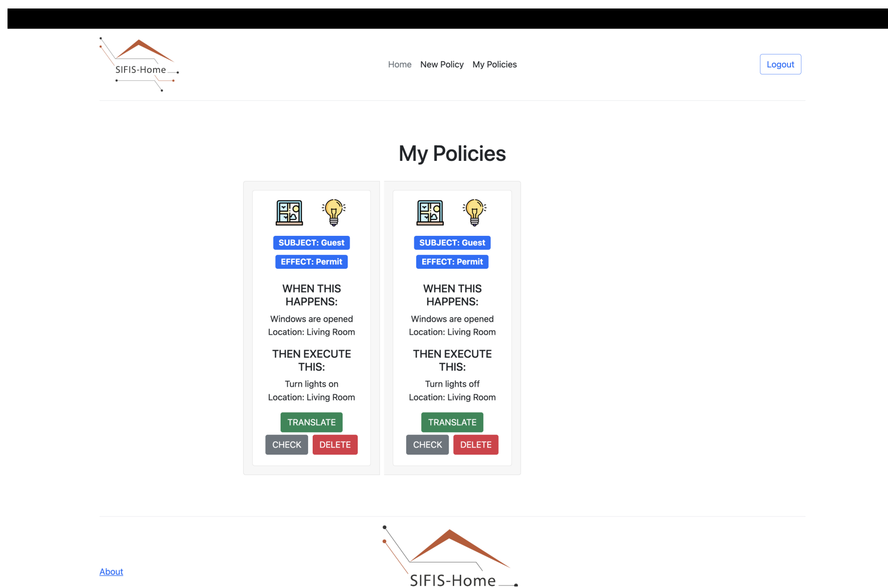

Figure 2.5: Policy list view in the legacy PTP.

The core of the application revolved around the creation of new privacy constraints, a process that seamlessly integrated the exploration of system capabilities directly into a unified form-based authoring flow. Rather than relying on an isolated directory to understand which devices exposed specific functions, users dynamically browsed available trigger and action services extracted directly from the in-memory ontology while actively constructing their high-level IF-THEN rules, ensuring that only valid device behaviors were selected during the authoring process. Figures 2.6 and 2.7 illustrate how the legacy interface presented the available action and trigger services, while Figure 2.8 shows the unified rule creation form.

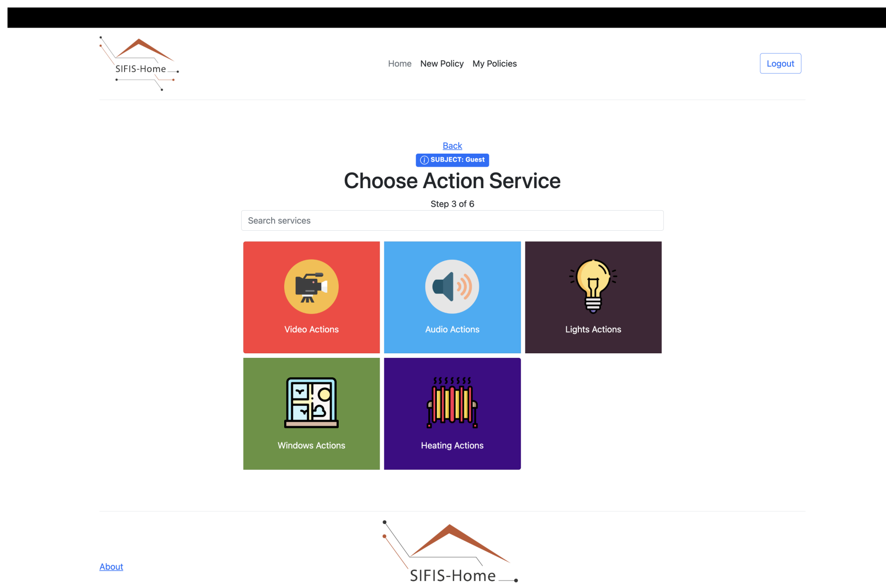

Figure 2.6: Action service selection view in the legacy PTP.

As stated before, one of the main characteristics of the legacy PTP was that, after being created, a rule could be submitted for a conflict test, triggering the backend to execute a Petri Net-based simulation to detect redundancies, or inconsistent behaviors. This simulation evaluated the newly proposed logic against the existing rule set and returned a human-readable report directly to the interface, as shown in Figure 2.9, alerting the user of any logical anomalies before translation.

Finally, after successful conflict validation and translation, the application generated the corresponding XACML artifacts, providing them directly to the user for export as an archive.

A more in-depth analysis of the data plane shows that the PTP relied on the in-memory ontology model as the single source of truth for the semantics of triggers, actions, and devices, while MySQL was strictly responsible for storing user-authored rules and metadata. In the end, since the frontend was delivered as static assets

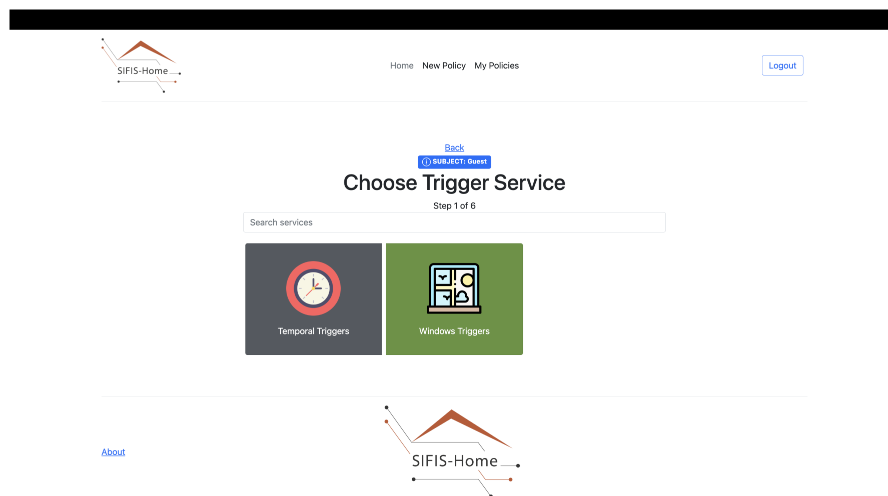

Figure 2.7: Trigger service selection view in the legacy PTP.

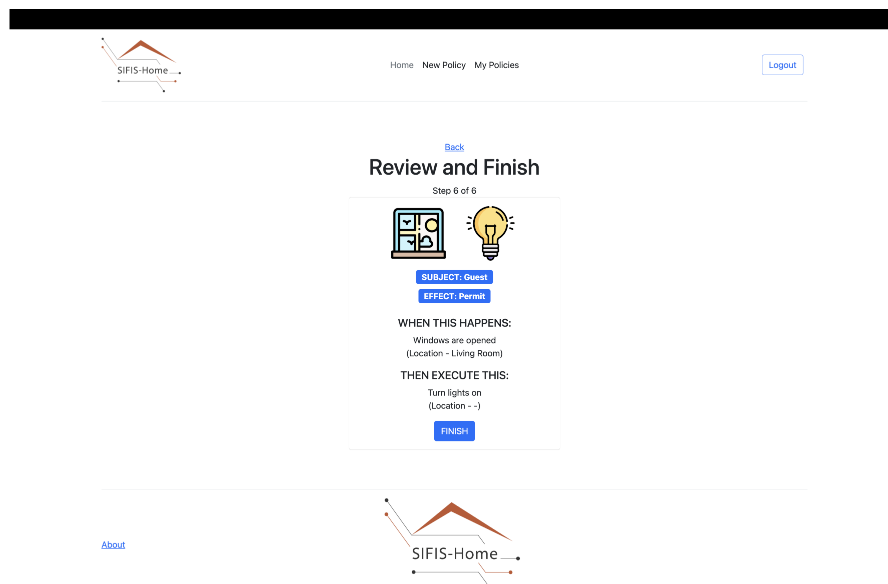

Figure 2.8: Rule creation form in the legacy PTP.

packaged directly within the Spring Boot application's resources suite, the system was fully deployed and executed as a single Java process.

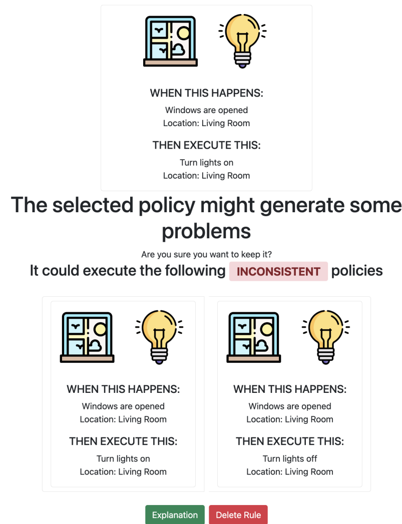

Figure 2.9: Conflict detection report in the legacy PTP.

2.2.2 Limitations of the Existing Policy Translation Point

One of the main limitations that the existing PTP had was the missing interaction with the overall SIFIS-Home runtime environment, particularly Smartotum and the DHT (Distributed Hash Table). The fact that the previous PTP was working mostly on a static ontology and stored rules, without any runtime interaction with real-world device information, gave it a tough restriction that did not allow the creation of context-aware policies. In this case, the application was simply a standalone tool where, because of the complete lack of any kind of external connectivity, all kinds of environmental and device information had to be completely hardcoded.

A more in-depth analysis of the architectural and functional constraints reveals that bundling the frontend, backend, and persistence into a single monolithic Spring Boot process made independent scaling and deployment exceptionally difficult. Furthermore, from a usability perspective, the interface was structured across an excessive number of static pages, resulting in a prolonged and confusing interaction flow. Moreover, the legacy architecture suffered from a strictly user-centric design rather than a home-centric one, meaning it erroneously considered each user as having only a single smart environment and consequently generated only one OWL file per user, a rigid approach that entirely prevented multi-home scalability. Additionally, several core functionalities did not operate properly during complex evaluations; for example, the Petri Net-based conflict detection mechanism would occasionally produce misclassifications, struggling to accurately distinguish between logical inconsistencies and simple policy redundancies when analyzing overlapping

rules.

Ultimately, these constraints meant the old PTP was functionally useful as a controlled research demonstration but not robust enough for production-like integration into an end-to-end enforcement pipeline. These limitations, spanning functional, architectural, and integration concerns, set the foundations for the architectural redesign and the development of the modernized application, which will be presented in the following chapter.

Chapter 3

System Features and User Interface

The redesigned Policy Translation Point application provides a web-based interface through which users can create, verify, translate, and manage privacy-related rules within a Smart Home environment. Unlike the original prototype, which offered excessive static pages and dispersed views, the modernized version consolidates the interaction into a more compact and dynamic workflow-oriented interface, reducing visual noise and improving clarity.

This chapter describes the main user interactions supported by the PTP, following the typical lifecycle of a policy—from visualization to creation, verification, translation, and deletion.

3.1 User Authentication and Account Access

In this scenario, the interaction with the system begins with a standard authentication flow, where individuals perform login operations using existing credentials or sign up by providing basic information to create a new personal profile (Figure 3.1).

Within the broader SIFIS-Home project and the *Privacy Dashboard*, accounts are assigned specific roles (e.g., Data Controller, Data Protection Officer) to govern administrative access. However, it is crucial to clarify the terminology used within the PTP module specifically. When authoring policies, the user of the PTP is typically a Smart Home resident. In the context of the PTP's access control logic, this user acts as the *subject* of the policy, the entity requesting or restricting access to a domotic resource. This should not be confused with the strict legal definition of a "Data Subject" under the GDPR, which refers to the identified or identifiable natural person to whom personal data relates. While a homeowner using the PTP is inherently a GDPR data subject, within the PTP interface, their role is

implemented simply as the authenticated *subject* applying home-level constraints.

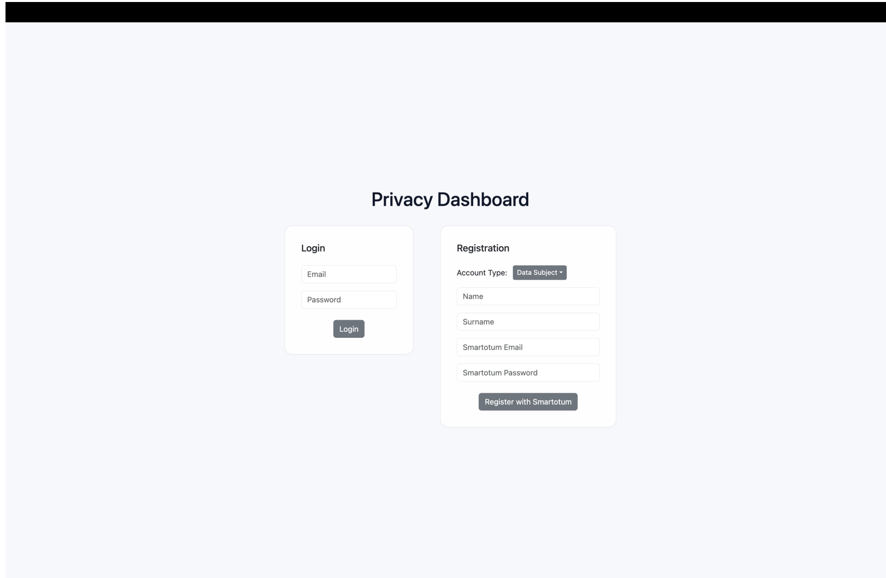

Figure 3.1: Authentication interface.

3.2 Smart Home Selection

After authentication, users are required to select the Smart Home environment they intend to manage (Figure 3.2). The interface displays a comprehensive list of available Smart Homes alongside metadata, such as home names or identifiers, to give the system the ability to support multi-home scenarios, ensuring that all subsequent rule creation and translation operations are context-aware and strictly bound to a specific physical topography and device inventory.

3.3 Policy Overview

Once a home is selected and the user navigates to the Policy Translation tab, a dashboard is presented functioning as the central management interface, showing the list of existing high-level policies (Figure 3.3). We focused on providing a transparent overview by displaying crucial details, such as the policy trigger, trigger details, action, action details, effect (deny or permit), and expiration date, alongside action buttons like Download, Check, and Delete to allow users to quickly locate and manage specific rules. Moreover, we added a “Create New Policy” button to link directly into the rule creation flow.

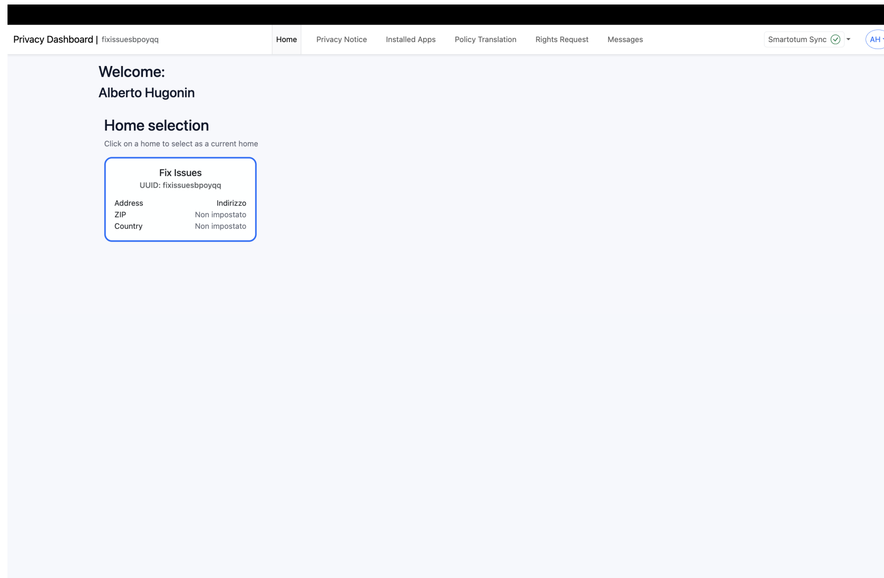

Figure 3.2: Smart home selection interface.

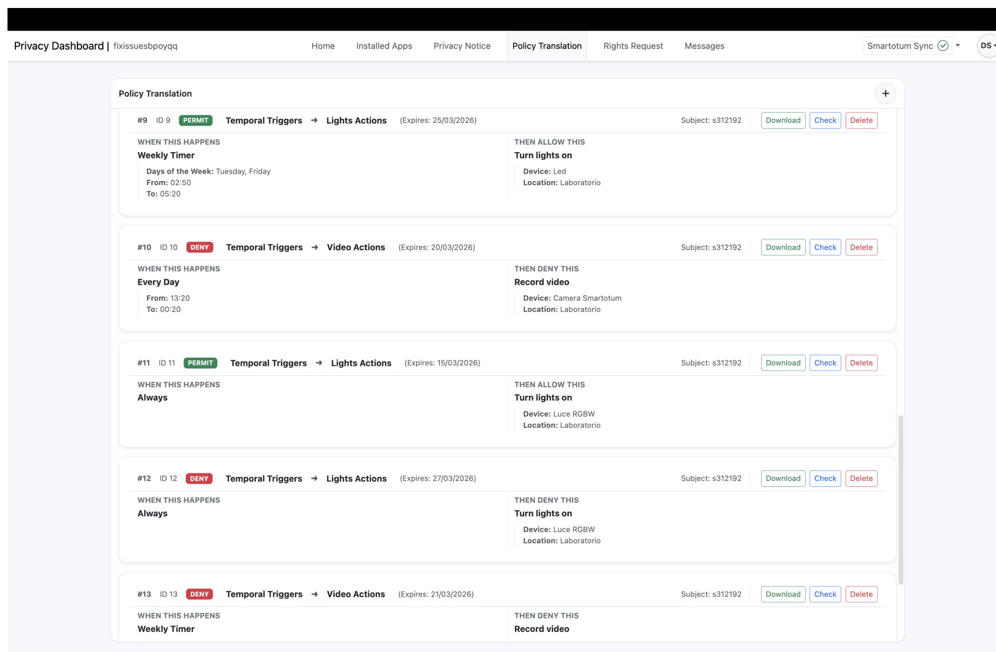

Figure 3.3: Overview of created high-level policies with action buttons.

3.4 Rule Creation

The rule creation is performed through dynamic components that allow users to effortlessly create a new policy, defining triggers, trigger details, actions, action details, effect (deny or permit), and expiration date into a single page (Figure 3.4). Unlike the legacy system, the new interface retrieves available capabilities dynamically from the integrated ecosystem, ensuring that users only select triggers and actions corresponding to currently available devices.

Figure 3.4: Rule creation editor with dynamic capability selection.

3.5 Policy Record Schema

Having detailed the rule creation interface, it is helpful to formalize the underlying data structure that these screens build. The PTP abstracts complex semantic reasoning into a user-friendly form. As the user interacts with the UI (selecting dropdowns, inputting times), they are systematically building a **Policy Record**.

This record serves as the bridge between the frontend React application and the backend Flask services. Table 3.1 explicitly defines this schema, demonstrating how each human-readable UI field maps to the structured data payload required by the backend and the underlying SIFIS-Home ontology.

Table 3.1: Policy Record Schema: Mapping UI fields to the underlying data model.

UI Field / Concept	Schema Property	Data Type & Description
Effect	`effect`	`String`: Either `"permit"` or `"deny"`. Defines the core authorization decision.
User (Hidden)	`subject`	`String`: The email address of the authenticated user creating the rule. Automatically injected by the UI context.
Expiration	`expirationDate`	`String`: An ISO 8601 date (e.g., `"2026-12-31"`).
Trigger/Action	`trigger.urlaction.url`	`String`: The ontological IRI representing the selected event or operation (e.g., `...#sifis_record_video_action`).
Time Parameter	`details[].value`	`String`: representing hours/minutes when the `type` is `"time"` (e.g., `"22:00"`).
Days Parameter	`details[].value`	`Array of Strings`: when the `type` is `"days"` (e.g., `["Monday", "Friday"]`).
Location	`details[].value`	`Object`: Contains `name` (human-readable room, e.g., `"Kitchen"`) and `url` (the semantic ontology URI).
Device (Entity)	`details[].value`	`Object`: Contains `name` (e.g., `"IP Camera"`) and `url` (the specific device instance URI).

3.6 Conflict Detection

When clicking the "Check" button on the policy overview page, users can initiate 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 3.5, 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 same action under the same trigger conditions). The report groups these conflicts together, displaying the exact devices and locations involved.

- **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.

3.7 Rule Download

Another functionality presented on the policy overview page is the "Download" button. Clicking it immediately triggers the backend translation engine, which converts the selected rule into the corresponding XACML policy files and packages them into a ZIP archive. The browser then automatically begins downloading the archive without requiring any additional confirmation, allowing users to rapidly export enforceable policies for external deployment.

3.8 Rule Deletion

To maintain a clean working environment, users can delete rules directly from the policy overview dashboard via the "Delete" button, an action that requires explicit confirmation to prevent accidental removal (Figure 3.6).

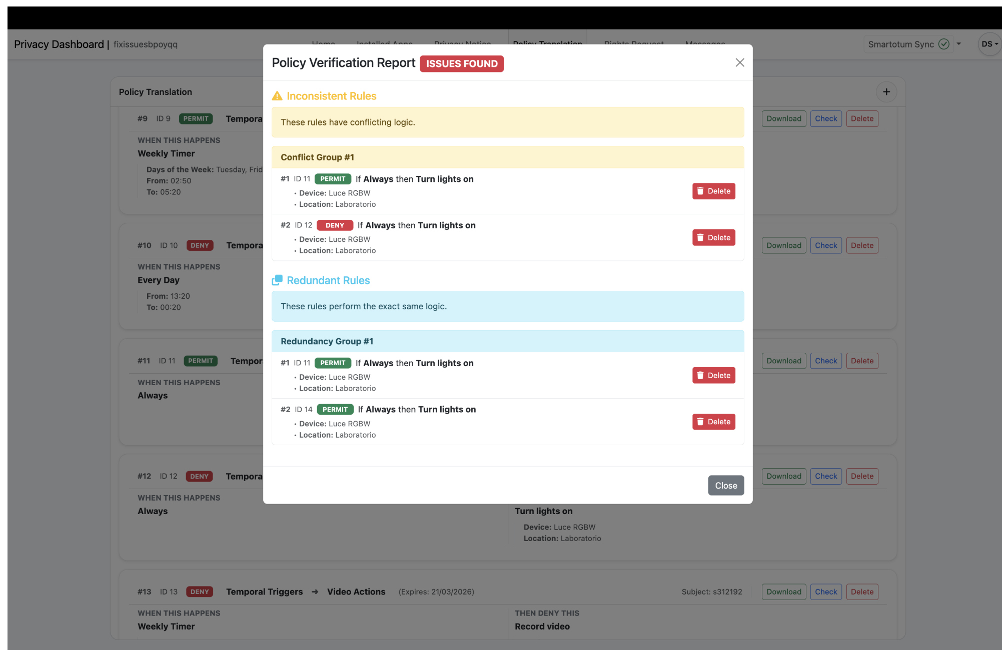

Figure 3.5: Policy verification report displaying categorized conflicts (inconsistencies and redundancies) with inline resolution actions.

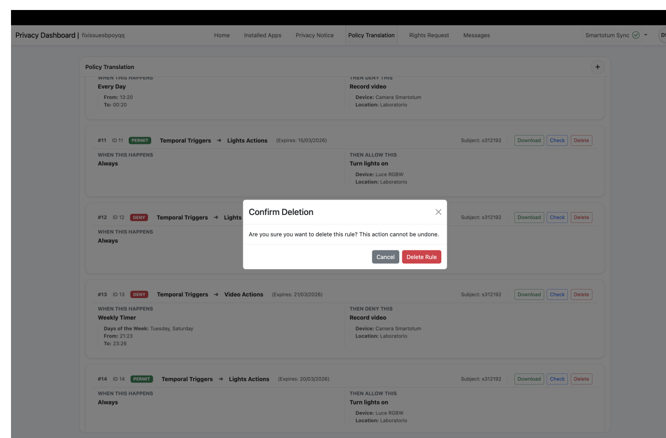

Figure 3.6: Deletion confirmation dialog.

3.9 End-to-End Usage Scenario

To contextualize how these individual features combine into a cohesive workflow, consider a practical end-to-end scenario of a user securing their Smart Home against unauthorized video recording.

The scenario unfolds through the following sequence:

1. **Initialization:** The user logs into the PTP dashboard using their credentials (an email address acting as the policy *subject*) and selects their primary residence from the Smart Home selection menu.

2. **Exploration:** The interface loads the home's ontological context, and the Policy Overview displays a clear list of existing rules. Noticing that weekend security rules are currently missing, the user clicks "Create New Policy".

3. **Authoring:** Inside the Rule Editor, the user specifies a "Deny" effect to act as a strict blacklist constraint. They begin by selecting the *Temporal Triggers* service, choosing the specific *Weekly Timer* trigger, and then setting the trigger details to active days "Saturday" and "Sunday" between 22:00 and 06:00. For the action side, they select the *Video Actions* service, choose the *Record video* action, and finally specify the action details by selecting the target location ("Living Room") and the specific device ("IP Security Camera"). Lastly, they define an expiration date.

4. **Verification and Adjustments:** After saving, the new policy appears on the dashboard. The user clicks "Check" to ensure this new constraint does not clash with existing behaviors. The Policy Verification Report pops up, highlighting a yellow *Inconsistent Rule* warning: another family member had previously created a "Permit" rule that allows the camera to record on all days.

5. **Resolution:** Using the inline "Delete" button provided directly within the yellow warning box of the verification report, the user immediately removes the older, overly permissive rule, resolving the conflict.

6. **Deployment:** Finally, the user clicks "Download" on their new policy. The backend translates the semantic entities into a standard XACML format rule file and packages it into a ZIP archive, ready for deployment to the home's Policy Decision Point (PDP) for formal enforcement.

This scenario demonstrates the platform's core capability: empowering non-technical end-users to manage complex, location-aware privacy constraints through an iterative, feedback-driven graphical interface, without ever needing to interact with underlying query languages or raw data structures.

Chapter 4

System Architecture and Technologies

4.1 High-Level Architecture

Since our software operates over the Internet, it can be considered as a *network-based application*. There are many different architectural patterns that could be adopted to implement such type of application, and one of the most common is the *client–server architectural model*, which is also the one we chose. The main idea behind this distributed architecture is to have a clear separation of concerns between the two main components, the *client*, *i.e.* the service consumer, and the *server*, *i.e.* the service provider. The server component, offering a set of services, listens for requests upon those services. The client component, which desires a service to be performed, sends a request to the server via a connector. The server either rejects or performs the request and sends a response back to the client [19], as depicted in Figure 4.1.

The client is where the computation and rendering of the user interface happens, so it results in what the end user sees and interacts with in the browser, while the server has the main function of processing and storing data that goes to and comes from the client that is sending and receiving requests. Within this context, the client is often referred to as the *frontend*, whereas the server is often called the *backend*. The separation of concerns between client and server brings many advantages, where one of them is the possibility for frontend and backend to evolve independently: the former can have its portability improved to work across multiple platforms, the latter can have its scalability improved by simplifying its components [19]. The communication between client and server happens via the Internet and cannot be set up arbitrarily by the developer or the end user, but it has to follow a predefined message exchange protocol, the HTTP (Hypertext Transfer Protocol).

Fielding et al. formally defined HTTP as [20]:

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

All requests we mentioned earlier sent by the client are HTTP requests and receive from the server an HTTP response, and they have to follow a precise structure. HTTP requests contain different fields, among which we find: an URL (Uniform Resource Locator), used to identify the resource in the server we want to point to; a method, which describes the type of action we want the server to perform (*e.g.* read or write some desired data); an optional request body, for requests that write or update data stored in the server. Similarly, HTTP responses also have a specific structure, with different fields being, among the others: a status code, which informs the client on the outcome of its request (*e.g.* whether it has failed or completed successfully), and a response body, which contains the data requested by the client. An example of an HTTP request with the corresponding response is given in Figure 4.2. As mentioned in the definition of HTTP, one core property of this protocol is statelessness: each HTTP request is independent, since it must contain all of the information necessary to the server for understanding it, and cannot take advantage of any stored context on the server. This improves the properties of visibility, reliability, and scalability. However, the main disadvantage is that it may decrease network performance by increasing the repetitive data sent in a series of requests, since that data cannot be left on the server in a shared context [19].

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

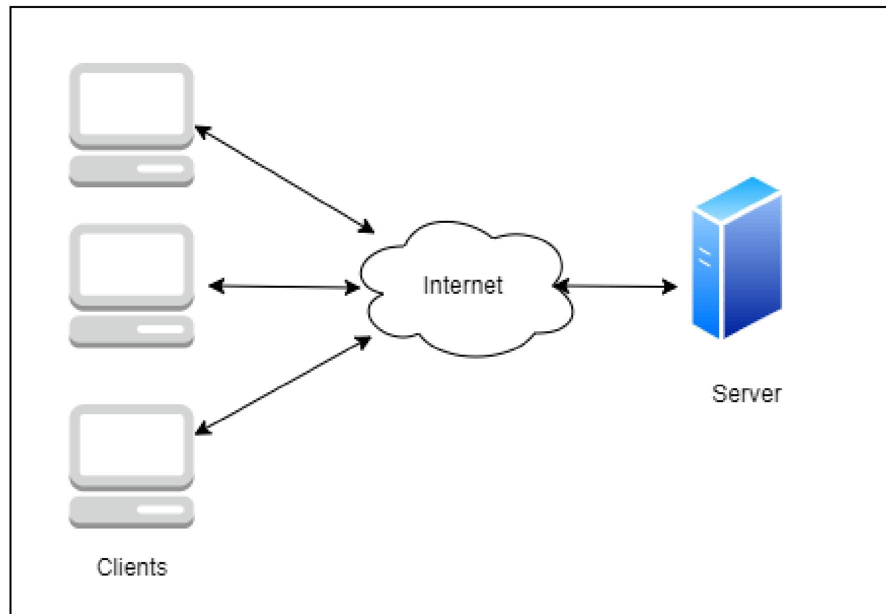

Figure 4.1: The Client-Server Architecture.

```
▼ GET http://127.0.0.1:5050/api/privacy-notices/managed-application/0979ab4a-14cd-46f9-b58d-95de17bf5d60    200
  ▶ Network                                                                                                  Show
  ▶ Request Headers
  ▼ Request Body ↗
    ∅
  ▶ Response Headers
  ▼ Response Body ↗
    {
      "content": "Updated Privacy Notice for Application 1",
      "uuid": "170b94bc-ed61-42ec-9928-7d77a24dc4d4"
    }↵
```

Figure 4.2: Example of an HTTP request and response.

4.2 Database Technologies

4.2.1 The Relational Data Model

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

Relational databases are a class of databases that manage data relying on the *relational model*, which was first proposed in 1970 by E. F. Codd [21]. 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 [21]. 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 [21]. 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 [21] 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 [22]. A DBMS allows user to perform various operations, starting from the creation of the database *schema*, *i.e.* the definition of the tables the user wants to create, specifying attributes, data types and constraints for each of them. It then allows to access such data, by enabling users to insert, retrieve, update and delete rows from the tables - this set of basic operations is often referred to as *CRUD (Create, Read, Update, Delete)*. In addition to a system that provides CRUD operations, Connolly and Begg listed other important systems that every DBMS shall include [22]:

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

The systems described above collectively enable a DBMS to provide reliable transactional behavior. A *transaction* can be defined as a sequence of operations executed as a single logical unit of work and is characterized by the ACID (Atomicity, Consistency, Isolation, Durability) properties, which Gray and Reuter defined as follows [23]:

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

In the context of our project, it was then important to choose a DBMS that would fully support ACID properties, and that is the reason why we adopted *PostgreSQL*, a free and open-source relational DBMS. First designed at the University of California, Berkeley in 1986 [24], PostgreSQL has now become one of the world's most used DBMS, according to Stack Overflow's 2025 Developer Survey [25]. It was given that name to emphasize its full support to SQL (Structured Query Language), which is the most widespread language to define and manage tables, perform CRUD operations and more. Moreover, PostgreSQL is fully compliant to ACID properties through its MVCC (Multi-Version Concurrency Control) mechanism, so it enables high levels of concurrent access without compromising consistency. Its extensible architecture allows for the definition of custom data types, operators, and indexing methods, making it suitable for a wide range of applications, including our *Privacy Dashboard*.

4.3 Backend Technologies

4.3.1 Monolithic Architecture and the Controller-Service-Repository Pattern

Even though PostgreSQL provides the mechanisms necessary for persistent data storage and reliable transaction management, it represents only the bottom layer of the backend infrastructure. A complete backend system must also define how HTTP requests are handled, how application logic is structured, and how interactions with the DBMS are organized. For this purpose, architectural patterns are adopted to ensure a more structured and consistent design and development process that leads to a more maintainable result. There are various architectures that may be implemented to design and structure a backend system, and the two most common are *monolithic* and *microservices*. Monolithic architecture refers to the traditional approach of building software systems as a single, unified application, so all components are packaged and deployed as a single executable unit, typically sharing the same runtime environment; on the other hand, microservices architecture refers to the more modern approach of structuring applications as a collection of small, autonomous services, each responsible for a distinct business capability, so components are free to evolve, deploy, and scale independently [26]. 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 [26], 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 [19]. 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 [27]. 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 [25], Flask has become one of the most popular Python web

application frameworks. As a "microframework" [28], Flask provides the essential components required to handle HTTP requests with native JSON support, define routing mechanisms, and generate responses, while leaving architectural decisions and application structure to the developer. This minimalist design aligns well with the Controller–Service–Repository pattern described in 4.3.1, as it allows a clear separation between request handling, business logic, and DBMS access without imposing a rigid framework structure.

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

4.4 Frontend Technologies

4.4.1 Single Page Application Architecture

Having analyzed the main technologies behind our backend, we can now describe the outermost layer of client-server architecture: the *frontend*. As introduced in 4.1, the main role of the client is to send and receive API requests and responses to the server and, based on the data obtained, render a page on the web browser that the end user can interact with. As we have seen for the backend, there exist different architectural patterns to design and develop a frontend system too, where the two most common are *MPA (Multi-Page Application)* and *SPA (Single Page Application)*. In web applications developed following the traditional MPA architecture, the frontend is based on a multi-page interface model, in which on each navigation event requiring new content, the entire page is reloaded: the server is responsible for generating complete HTML documents, which are transmitted to the browser in each API response. This server-side rendering can lead to increased latency and reduced interactivity compared to more dynamic approaches. The SPA pattern was later developed to improve the user experience by focusing on not refreshing the entire web page at every API request: in this new model, the SPA interface is composed of individual components which can be updated and replaced independently, so that the entire page does not need to be reloaded on each user action. This, in turn, helps to increase the levels of interactivity and responsiveness [29], since most of the rendering logic is shifted from the backend to the frontend, so the user no longer has to wait for the backend to generate full HTML documents for each interaction. In order to dynamically update its components, the SPA does not need to receive HTML content, but rather some structured information that it can map to a web layout, and a fitting example of this required data is JSON objects: a RESTful API that returns JSON objects, such as the one we have described in the previous section, is then a type of backend that matches well with a SPA. Whenever the user asks for some new information, the SPA sends a API request to the RESTful backend, which in turn returns the desired data in a JSON

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

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

4.4.2 React

The implementation of a SPA is often carried out adopting a dedicated frontend framework that provides a structured set of tools and abstractions that facilitate the development of interactive user interfaces executed directly within the web browser. This client side execution is enabled by the fact that modern browsers embed a JavaScript engine, which is capable of running an application written in a programming language such as JavaScript or TypeScript. There exist many dedicated frontend frameworks useful to build such kind of client application, and the one we adopted for our project is *React*. Developed in 2013 by Meta (formerly Facebook), React is a free and open-source library, and according to Stack Overflow's 2025 Developer Survey [25] 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 [30]. React enhances modularity by following a declarative programming pattern, since the developer describes, for each component, how it is structured and how it should behave on changes in the application state. Each component may maintain an internal state, representing dynamic data that determine how it is rendered in the browser. The structure of each component can also include other nested components, and, in order to allow changes in a parent component's state to propagate to its child components through updated properties (*props*), React allows downward state propagation, which results in a flow of updated data from higher to lower levels of the component hierarchy.

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

change on the corresponding branches and nodes of its virtual DOM. It then runs an algorithm to compute the differences between its updated virtual DOM and the browser's DOM so that it can proceed to update the latter, modifying only the affected nodes and branches, rather than building the entire tree again from its root [31]. This is how React implements the SPA feature, introduced in the previous section, of not reloading the entire page at every user's event, but rather only making as little changes as possible to the rendered page to maximize efficiency and responsiveness. Figure 4.5 shows an example of this feature in action: the first list item element had a class "selected" which was then removed, and this is reflected in the "New Virtual DOM"; the Real DOM is then updated by only removing such class from the corresponding HTML element, rather than re-building the entire HTML tree from the root.

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

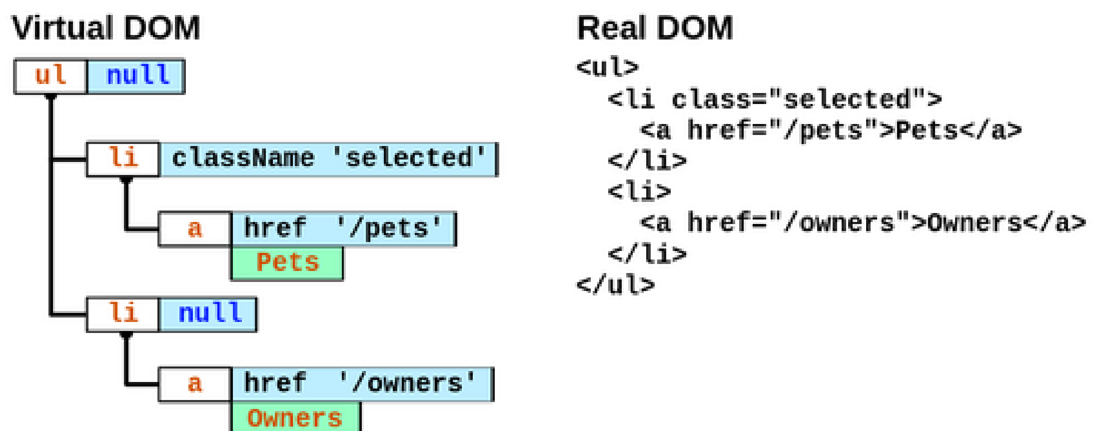

Figure 4.4: Example of virtual DOM compared to browser's DOM.

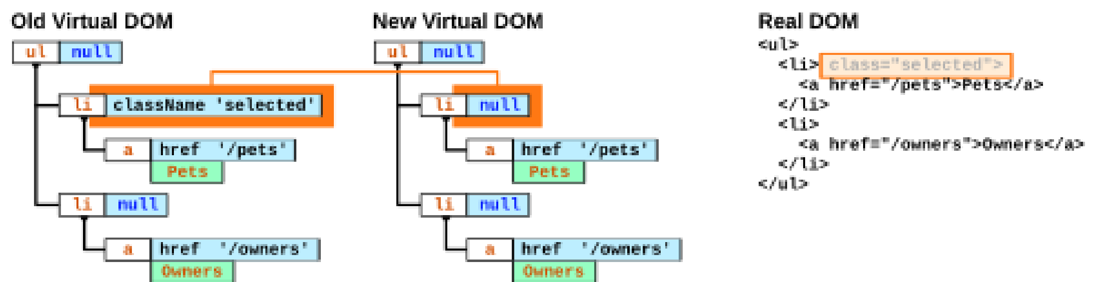

Figure 4.5: Example of update to virtual DOM reflected to browser's DOM.

Chapter 5

Implementation

This implementation chapter details the practical translation of the architectural concepts discussed so far into a functioning software system. The development effort focused on transitioning from a standalone system to a robust Client-Server model. This involved building a SPA utilizing React for the frontend user interface, and establishing a monolithic RESTful API using Flask for the backend. The core challenge addressed in this chapter is the practical realization of this architecture, particularly how complex semantic reasoning requirements are handled within a modern web application framework.

5.1 Backend Implementation

The backend component, developed with Flask, serves as the computational core of the Policy Translation Point. Following the Controller-Service-Repository pattern, the application strictly separates HTTP request handling from core processing.

5.1.1 Rule Ingestion and Sanitization

The primary ingestion point for policies defined by the user is the `SifisController`. This interface layer receives the incoming HTTP POST requests from the React frontend, handling initial authentication and routing. Once authorized, the controller passes the raw payload down to the `RuleService` layer, which validates the request against a rigidly defined schema. Listing 5.1 shows the complete payload schema enforced by the backend's `RuleRequestSchema` via the `marshmallow` validation library:

```
1  class RuleDetailSchema(Schema):
2      url = fields.Str(required=True)         # Ontological IRI of
       the detail
```

```
3       name = fields.Str(required=False)          # Human-readable
    label
4       type = fields.Str(required=True)           # "time", "days", "
    location", "entity"
5       value = fields.Raw(allow_none=True)        # Type-dependent: str
    , list, or dict
6
7   class RuleTriggerSchema(Schema):
8       url = fields.Str(required=True)            # IRI of the trigger
9       details = fields.List(fields.Nested(RuleDetailSchema),
    dump_default=[])
10
11  class RuleActionSchema(Schema):
12      url = fields.Str(required=True)            # IRI of the action
13      details = fields.List(fields.Nested(RuleDetailSchema),
    dump_default=[])
14
15  class RuleRequestSchema(Schema):
16      trigger = fields.Nested(RuleTriggerSchema, required=True)
17      action = fields.Nested(RuleActionSchema, required=True)
18      effect = fields.Str(required=True)         # "permit" or "deny"
19      subject = fields.Str(allow_none=True)      # User email
20      expirationDate = fields.Str(allow_none=True) # ISO date
```

Listing 5.1: Backend rule payload schema enforced via marshmallow validation.

The `value` field in each `RuleDetailSchema` is polymorphic: it holds a string for `time` details (e.g., `"22"`), an array of strings for `days` details (e.g., `["Monday", "Friday"]`), and a nested object with `name` and `url` keys for `location` and `entity` details. When the `RuleService` receives this validated payload, it applies a sanitization process before persisting it to the database. A specific challenge was ensuring that data remained both semantically accurate for future reasoning and human-readable for UI presentation. During development, a sanitization step was implemented to handle location data: the frontend often transmits locations as raw ontological URIs or complex string formats, and the `RuleService` intercepts these payloads to extract the clean, human-readable room name (e.g., transforming `http://example.org/ontology#Kitchen` to simply `Kitchen`):

```
1   elif detail["type"] == "location":
2       location_value = detail.get("value", {})
3       if location_value:
4           # Extract both the semantic URL and the human-readable
    name for the DB
5           location_url = location_value.get("url")
6           db_detail = DbDetail(
7               url=detail.get("url"),
8               type=detail["type"],
```

```
9                value=location_value.get("name"), # Human readable
    name
10               value_url=location_url            # Semantic URI
11           )
12           db_action.details.append(db_detail)
```

Listing 5.2: Location data sanitization within the RuleService layer.

5.1.2 The Role of OWL Files

The core of the semantic integration relies on distinct OWL files handling different levels of abstraction. To understand how home automation features are mapped to ontology definitions, it is necessary to establish the semantic relationships: each `Device` possesses one or more `Service` nodes; each `Service` provides `Command` nodes; each `Command` allows the execution of specific `Action` nodes; and each `Channel` offers available `Action` nodes.

The application manages this hierarchy across three distinct structural layers:

1. `eupont.owl`: This upper-level, generic internet-of-things ontology serves as the foundational bedrock. It provides the strict abstract class definitions for all major conceptual entities: `Device`, `Service`, `Command`, `Action`, and `Channel`.

2. `sifis-home.owl`: This project-specific overarching schema inherits from `eupont.owl`. It populates the conceptual space by instantiating the universal instances of `Action` and `Channel` entities that are common across the entire system.

3. Environment-Specific Ontologies (`<home_uuid>.owl`): Finally, the application dynamically manages distinct files for each Smart Home cluster. These files store the exact physical instance data for a particular real-world environment, specifically creating the explicit instances of `Device`, `Service`, and `Command` that correspond to the actual hardware operating in that home.

```
1 def _get_locations(self, home_uuid: Optional[str] = None,
    action_id: Optional[str] = None):
2     # Load home ontology in separate world context
3     from owlready2 import World
4     home_world = World()
5     home_onto = home_world.get_ontology(f"file://{
    home_ontology_path}").load()
6     # Run reasoner to infer implicit properties before querying
7     sync_reasoner_hermit(home_world, infer_property_values=True)
8     # ... logic to filter Location instances based on action
    allowTo properties ...
```

Listing 5.3: Dynamic ontology loading and semantic reasoning in OWLService.

5.1.3 Hydration and Dehydration

To effectively operate the integrated DB/OWL architecture, the backend must constantly translate states between flat relational tables and complex semantic graphs. This core software pattern, which we refer to as "dehydration" and "hydration", is primarily the responsibility of the `RuleService`.

When a rule is ingested and verified, it exists as a rich semantic object (`Rule`) containing full URIs, nested properties, and contextual metadata. To save this efficiently to PostgreSQL, the system "dehydrates" the object, removing the semantic weight and extracting only the critical relational markers (e.g., rule type, trigger ID, exact string of the target device).

By contrast, when the backend needs to evaluate existing rules, such as during conflict detection (`PetriNetService`), policy generation (`XACMLService`), or even simply returning the comprehensive list of a user's rules to populate the frontend dashboard via the `GET /rules` endpoint, the system must "hydrate" these flat database records back into fully realized semantic objects. The `_rehydrate_rule` function achieves this by fetching the raw `DbRule` and intersecting it with live data from the `OWLService`:

```
1  def _rehydrate_rule(self, db_rule: DbRule, home_uuid: str) ->
       Optional[Rule]:
2      # Extract the raw flat references from the database
3      db_trigger = db_rule.triggers[0]
4      db_action = db_rule.actions[0]
5      # Dynamically inject the rich semantic data back into the
       object from OWL
6      trigger = self.owl_service.get_trigger_by_url(db_trigger.url,
       home_uuid)
7      self._hydrate_details(trigger, db_trigger, home_uuid)
8      action = self.owl_service.get_action_by_url(db_action.url,
       home_uuid)
9      self._hydrate_details(action, db_action, home_uuid)
10     # Reconstruct the full semantic Rule object for downstream
       analysis
11     return Rule(
12         db_id=db_rule.id,
13         trigger=trigger,
14         action=action,
15         effect=db_rule.effect,
16         # ... logic
17     )
```

Listing 5.4: Semantic object hydration logic in RuleService.

Table 5.1 summarizes which data fields are persisted in the relational database during dehydration and which are resolved from the ontology at runtime during hydration.

Table 5.1: Data dictionary for the hydration/dehydration process.

Field	Source	Stored as	Notes
`effect`	PostgreSQL	`VARCHAR`	"permit" or "deny"
`subject`	PostgreSQL	`VARCHAR`	User email
`timestamp`	PostgreSQL	`BIGINT`	Unix ms at creation
`expiration_date`	PostgreSQL	`VARCHAR`	ISO date string
`rule_uuid`	PostgreSQL	`UUID`	For DHT sync
`trigger.url`	PostgreSQL	`VARCHAR`	Ontological IRI
`trigger.name`	OWL	—	Resolved via `get_trigger_by_url`
`trigger.service`	OWL	—	Parent service metadata
`action.url`	PostgreSQL	`VARCHAR`	Ontological IRI
`action.name`	OWL	—	Resolved via `get_action_by_url`
`action.service`	OWL	—	Parent service metadata
`detail (time)`	PostgreSQL	`VARCHAR`	Value stored as string
`detail (days)`	PostgreSQL	`VARCHAR`	JSON-encoded array
`detail (location)`	Both	`VARCHAR`	Name in DB, alternatives from OWL
`detail (entity)`	Both	`FK → Device`	`device_id` in DB, full metadata from OWL

5.1.4 Implementing Semantic Colored Petri Nets

A critical feature separating the PTP from standard applications is its ability to detect structural and logical conflicts among privacy rules. As theorized in Chapter 2, this formal verification is implemented using Semantic Colored Petri Nets within the `PetriNetService`, following the SCPN topology described in Section 2.1.7.

The verification leverages an asynchronous, user-driven workflow. Rather than evaluating every rule synchronously before allowing a DHT submission, the user interface provides a dedicated button that triggers a verification check against the current ruleset. When initiated, the `PetriNetService` constructs an `ExtendedPetriNet` graph containing all existing rules plus the target rule, mapping

`Trigger` instances to `TriggerPlace` vertices, `Action` instances to `ActionPlace` vertices, and connecting them through `RuleTransition`, `CopyTransition`, and `TriggerTransition` edges as described in the SCPN formalism.

Once the graph is built, the `check_rule` method detects conflicts by identifying convergences: if two `RuleTransition` edges arrive at the same `ActionPlace` from the same `TriggerPlace`, their effects are compared. Identical effects indicate a redundancy, while opposing effects (e.g., `Permit` vs. `Deny`) indicate an inconsistency. The resulting `RuleProblems` report is returned to the frontend, allowing the user to make an informed decision.

```python
def check_rule(self, target_rule, existing_rules, home_uuid):
    rule_problems = RuleProblems()
    rules_after = [r for r in existing_rules if r.db_id != target_rule.db_id]
    rules_after.append(target_rule)

    # Build the full SCPN graph spanning all rules
    net = self.get_net(home_uuid, rules_after, refresh=True)
    target_action_place = net.get_action_place(target_rule.action)
    target_trigger_place = net.get_trigger_place(target_rule.trigger)

    if target_action_place and target_trigger_place:
        # Find all RuleTransitions converging on the same ActionPlace
        for arc in net.output_arcs:
            if arc.place == target_action_place \
               and isinstance(arc.transition, RuleTransition):
                rule = arc.transition.get_rule()
                if rule.db_id == target_rule.db_id:
                    continue
                if net.get_trigger_place(rule.trigger) == target_trigger_place:
                    if target_rule.effect == rule.effect:
                        rule_problems.add_redundant_rules([target_rule, rule])
                    else:
                        rule_problems.add_inconsistent_rules([target_rule, rule])

    return rule_problems.to_dict()
```

Listing 5.5: Structural conflict detection via convergence analysis on the SCPN graph.

5.1.5 XACML Generation

While the frontend Rule Editor allows users to define policies using intuitive, high-level concepts (e.g., "If Kitchen Window is Open, Turn Off Radiator"), the final enforcement nodes require a rigorously structured format. The `XACMLService` bridges this gap by translating internal SIFIS rules into the XACML 3.0.

The Python module parses the complex JSON rule payloads into standard XML element trees. A key implementation challenge was distinguishing between specific resource policies and generalized "Entire Home" policies. When localized actions are targeted, the service maps the exact SIFIS entity IDs to internal `ResourceMatch` elements. The service breaks down intuitive triggers into specific XACML context blocks: `PreCondition` (evaluating device state), `OngoingCondition` (evaluating environmental bounds like `time-in-range`), and `PostCondition` (enforcing subject constraints).

```
1 def _create_simple_condition(self, decision_time, cond_type, value
    , attr_id, category):
2     # Construct strictly compliant XACML 3.0 elements
3     condition = Element(QName(XACML_NS, "Condition"), {"
    DecisionTime": decision_time})
4     root_apply = SubElement(condition, QName(XACML_NS, "Apply"),
5                             {"FunctionId": "urn:oasis:names:tc:
    xacml:1.0:function:and"})
6     # ... nested apply and AttributeDesignator matching Java JAXB
    structure ...
7     return condition
```

Listing 5.6: Translating JSON constructs into standard XACML 3.0 conditions.

5.1.6 DHT Integration

To function within the decentralized SIFIS-Home ecosystem, the PTP must act as a synchronized node, communicating via a DHT (Distributed Hash Table). This integration is primarily managed by the `DHTService` and the `RuleService`.

Ingestion, Alignment, and Ontology Mapping (GET)

To ensure the PTP's worldview is accurate, it must align its local relational database with the global state of the DHT. Rather than passive polling, the system actively synchronizes rules via `RuleService.sync_rules_for_home` when a user requests on their dashboard view.

A critical hurdle in this synchronization is that the DHT simply stores raw JSON rule payloads, devoid of Semantic Web context. Therefore, when the application executes a GET request against the Smartotum DHT environment, it must map these

raw remote payloads back into the local relational database while simultaneously linking them to the core ontologies (`eupont.owl` and `sifis-home.owl`).

The `sync_rules_for_home` function achieves this mapping by extracting strings like device IDs or trigger IDs from the DHT payload. It leverages the `OWLService` to verify that these literal strings correspond exactly to valid `iri` (Internationalized Resource Identifier) nodes existing within the current home's ontology file. If the ontology confirms the entity's existence, a local `DbRule` record is instantiated or updated. Rules that exist in the DHT but not locally (or whose ontology nodes have been revoked) are managed accordingly, ensuring the user interface always reflects a true, ontologically backed state of the distributed environment.

```
1  def sync_rules_for_home(self, home: SmartHome, smartotum_rules:
       List["SmartotumRule"], access_token: str = None):
2       # 1. Map existing local PostgreSQL rules
3       local_db_rules = self.rule_repo.get_rules_by_home_and_type(
       home, "sifis")
4       local_rules_map = {r.rule_uuid: r for r in local_db_rules if r
       .rule_uuid}
5       # 2. Map incoming raw JSON payloads from the DHT
6       smartotum_rules_map = {str(r.rule_uuid): r for r in
       smartotum_rules}
7       # ... logic for handling deletions ...
8       # 3. Instantiate rules present in the DHT but missing locally
9       for rule_uuid, s_rule in smartotum_rules_map.items():
10          if rule_uuid not in local_rules_map:
11              try:
12                  # This function (create_rule_from_smartotum) maps
       the raw strings
13                  # against the SIFIS-Home ontology before saving to
        PostgreSQL
14                   self.create_rule_from_smartotum(home, s_rule)
15              except Exception as e:
16                  self.logger.error(f"Sync: Failed to create rule {
       rule_uuid}: {e}")
```

Listing 5.7: Synchronizing remote DHT payloads with local databases and ontologies.

Publishing Policies (POST)

When a user authors a new rule via the React interface, the `DHTService` translates the local database action into a distributed command. Upon saving to PostgreSQL, the service extracts the specific UUIDs and context data (e.g., target topics, active days, expiration dates) and packages them into a specialized JSON payload. A POST request is fired to the specific topic URL schema

(`<dht_endpoint>/topic_name/<rule_topic>/topic_uuid/<topic_uuid>`) accompanied by the user's secure Smartotum access token. This immediately propagates the new compliance rule across the Smart Home network for localized enforcement by target devices.

```
1 def publish_rule(self, topic_uuid: str, payload: dict, home_uuid:
    str, access_token: str):
2     headers = {"Content-Type": "application/json"}
3     if access_token:
4         headers["Authorization"] = f"Bearer {access_token}" #
    Smartotum Auth
5     # Constructing the dynamic DHT endpoint for the specific home
    and rule
6     base = self.dht_endpoint.replace("house_name", str(home_uuid))
7     url = f"{base}topic_name/{self.rule_topic}/topic_uuid/{
    topic_uuid}"
8     response = requests.post(url, json=payload, headers=headers,
    timeout=10)
9     return response.status_code in [200, 201]
```

Listing 5.8: Secure rule publication to the distributed hash table network.

5.2 Frontend Implementation

The frontend of the PTP, built with React, is responsible for abstracting the complex backend reasoning processes into an intuitive interface for the end-user. The primary feature of the SPA is the Rule Editor, a dynamic form that guides users through the creation of expressive Smart Home policies.

5.2.1 Component Architecture

The frontend is organized into a page-level container and three specialized components, each handling a distinct phase of the policy lifecycle:

- `PolicyTranslation.tsx` (page): The orchestrating container that manages the top-level application state, including the list of rules, the visibility of the editor form, and the verification report modal. It coordinates communication between the child components and the backend API.
- `RuleEditor.tsx` (component): A multi-step form that guides the user through rule creation: selecting a trigger service, choosing a specific trigger, configuring its contextual details (time, days, location), and then repeating the process for the action side. Each selection step dynamically fetches the available options from the `OWLService` via cascading API calls.

- `RuleList.tsx` (component): A card-based display that renders the user's existing policies. Each card shows the trigger-action pair, the effect badge (`PERMIT` or `DENY`), the subject, and the expiration date. It also provides action buttons for downloading XACML artifacts, triggering conflict checks, and deleting rules.
- `VerificationReport.tsx` (component): A modal dialog that presents the results of a Petri Net conflict check. It categorizes detected issues, each with a distinct severity color and inline delete buttons that allow the user to resolve conflicts directly from the report.

5.2.2 State Management and Validation

The Rule Editor manages a complex, multi-faceted application state. As a user selects different variables, navigating the rooms, selecting devices, and defining allowed actions, the React component tree must instantly reflect these conditional choices. This is achieved through a cascading chain of `useEffect` hooks, where each selection triggers an asynchronous fetch that populates the next level of options:

```
1  // When the user selects a trigger service, fetch its available
       triggers
2  useEffect(() => {
3    if (selectedTriggerService) {
4      getTriggers(homeUuid, selectedTriggerService).then(setTriggers
       );
5      setSelectedTrigger(""); // Reset downstream selections
6      setTriggerDetails([]);
7    }
8  }, [selectedTriggerService, homeUuid]);
9
10 // When a trigger is selected, fetch its configurable details
11 useEffect(() => {
12   if (selectedTrigger && selectedTriggerService) {
13     getTriggerDetails(homeUuid, selectedTriggerService, triggerId)
14       .then((details) => {
15         setTriggerDetails(details);
16         setTriggerDetailsValues({}); // Reset values for new
       details
17       });
18   }
19 }, [selectedTrigger, selectedTriggerService, homeUuid]);
```

Listing 5.9: Cascading state management via chained useEffect hooks in the Rule Editor.

This pattern ensures that downstream state is always consistent with upstream selections: changing the trigger service automatically clears the selected trigger

and its details, preventing stale data from persisting in the form.

A critical implementation obstacle addressed during development involved the management of asynchronous context, specifically pertaining to the `user` object. Early iterations of the interface encountered runtime errors when attempting to construct a rule payload before the system had fully resolved the authenticated user's session data from the overarching `privacy-dashboard` context. To resolve this, strict conditional checks were implemented within the `isFormValid` function to halt submission if the context is undefined, along with type-specific validation for each detail kind:

```
const isFormValid = () => {
  if (!selectedTrigger || !selectedAction || !user || !expirationDate)
    return false;
  for (const d of triggerDetails) {
    const val = triggerDetailsValues[d.url];
    if (d.type === "days" && (!Array.isArray(val) || val.length === 0))
      return false;
    if (d.type === "location" && (!val || !val.url)) return false;
    if (d.type === "time" && !val) return false;
  }
  return true;
};
```

Listing 5.10: Multi-type form validation ensuring all required fields are present.

5.2.3 Error Handling

Given that the Rule Editor relies on multiple sequential API calls to the backend, each asynchronous operation is wrapped in a `try/catch` block that isolates failures at the component level. If a fetch fails (e.g., the `OWLService` is temporarily unavailable), the affected dropdown is reset to an empty state rather than crashing the entire form. This defensive approach ensures that the interface remains usable even when individual backend services experience transient failures. Additionally, the parent `PolicyTranslation` page applies the same pattern to rule-level operations (save, delete, check), displaying error feedback to the user while preserving the current editor state.

5.2.4 Payload Construction

The final responsibility of the frontend is the translation of visual UI selections into the structured data format expected by the Flask backend. The Rule Editor seamlessly aggregates the user's intuitive choices (e.g., clicking "Allow", selecting

"Smart Bulb", choosing "Living Room") and constructs a rigidly defined JSON payload:

```
1  const rule = {
2    effect: "deny",
3    subject: "user@example.com",
4    expirationDate: "2026-12-31",
5    trigger: {
6      url: "http://elite.polito.it/ontologies/sifis-home.owl#
     sifis_time_trigger",
7      details: [
8        { url: "..#sifis_from_time", name: "From", type: "time",
     value: "22" },
9        { url: "..#sifis_to_time",   name: "To",   type: "time",
     value: "06" },
10       { url: "..#sifis_active_days", name: "Active Days", type: "
     days",
11         value: ["Monday", "Tuesday", "Wednesday", "Thursday", "
     Friday"] }
12     ]
13   },
14   action: {
15     url: "http://elite.polito.it/ontologies/sifis-home.owl#
     sifis_record_video_action",
16     details: [
17       { url: "..#sifis_location", name: "Location", type: "
     location",
18         value: { name: "Bedroom", url: "..#location-bedroom-uuid"
     } },
19       { url: "..#sifis_entity", name: "Device", type: "entity",
20         value: { name: "IP Camera 1", url: "..#d_camera_1" } }
21     ]
22   }
23 };
24 await onSave(rule);
```

Listing 5.11: Complete rule JSON payload example dispatched from the React frontend to the Flask backend.

This payload, validated by the backend's `RuleRequestSchema`, acts as the connection between the frontend and backend architectures. At the top level, `effect` specifies the authorization decision (`Permit` or `Deny`), `subject` identifies the authenticated user, and `expirationDate` defines the temporal validity of the policy. The `trigger` object contains the ontological IRI of the selected trigger and an array of `details` that provide its contextual parameters: `time` details specify temporal boundaries, while `days` details indicate the active weekdays as an array. Similarly, the `action` object references the target action's IRI along with its own details: the `location` detail identifies the room within the Smart Home, and the

`entity` detail specifies the exact physical device. This schema ensures that the specific details selected by the user are packaged accurately, holding to the contract expected by the `RuleService`, and providing the precise data points required for the hybrid database-ontology system to perform its later storage, conflict analysis, and XACML translation.

5.3 End-to-end Testing of the Policy Translation Point

Building upon the testing methodology established for the broader architecture described in the previous sections, the PTP module required its own focused validation campaign. Its hybrid database-ontology pipeline, live DHT synchronization, Petri Net conflict detection, and XACML translation introduce layers of complexity that demand systematic, reproducible verification. To this end, a dedicated “Policy Translation Point API” folder was appended to the existing Postman collection. Because this folder follows the Cleanup suite, which logs out the previous session, the first request is a Data Subject login that re-establishes the authenticated cookie before any PTP-specific endpoint is called.

5.3.1 Sequential Test Pipeline and Variable Propagation

A core design decision of the PTP test suite was to structure it as a strictly ordered pipeline, where each request depends on data produced by the previous one. Table 5.2 lists the twelve requests that compose the flow, grouped by their purpose.

Table 5.2: Sequential test pipeline for the Policy Translation Point API.

#	**Request Name**	**Purpose**
1	Login – Data Subject (PTP)	Re-authenticate after Cleanup
2	Successful Create Rule	Create the baseline `Deny` rule
3	Validation Error – Missing Effect	Negative: malformed payload
4	DHT Synchronization (Refresh)	Force live sync with Smartotum
5	Successful Retrieve Local Rules	Verify rule survives the sync
6	Seed Redundant Rule	Create identical `Deny` sibling
7	Conflict Detection – Redundancy	Assert redundancy detected
8	Seed Inconsistent Rule	Create opposing `Permit` sibling
9	Conflict Detection – Inconsistency	Assert inconsistency detected
10	XACML Generation (Download)	Validate ZIP archive output
11	Validation Error – Missing Home UUID	Negative: malformed path
12	Not Found – Invalid Rule ID	Negative: nonexistent resource

The chain is held together by Postman's `collectionVariables` API, which allows test scripts to persist values across requests without requiring a manually configured environment. Each creation endpoint extracts the generated database ID from its response and stores it under a named key (`test_rule_id`, `redundant_rule_id`, or `inconsistent_rule_id`). Downstream requests interpolate these keys directly into their URL paths and assertions, enabling fully automated execution of the entire pipeline.

5.3.2 Rule Creation and Schema Validation

The pipeline begins by posting a complete `Deny` rule payload to the `POST /api/sifis/homes/{hom` endpoint. The payload includes the ontological IRI of the trigger, the target action, location and entity details, and the user's email as the subject. Upon receiving a `200 OK` response, the test script validates the response body and extracts the rule's numeric ID for later reuse, as demonstrated in Listing 5.12.

```
1  pm.test("Response status code is 200", function () {
2      pm.expect(pm.response.code).to.eql(200);
3  });
4  var responseBody = pm.response.json();
5  pm.test("Response payload contains the generated rule ID",
       function () {
6      pm.expect(responseBody).to.have.property("id");
7      pm.expect(responseBody.id).to.be.a("number");
8  });
9  pm.test("Extract and persist Rule ID", function() {
10     pm.collectionVariables.set("test_rule_id", responseBody.id);
11 });
```

Listing 5.12: Postman assertion for rule creation, extracting the generated ID into a collection variable for propagation across subsequent requests.

Right after, a negative test sends a payload missing the mandatory `effect` attribute, asserting that the backend rejects it with a `422 Unprocessable Entity` and returns a precise schema validation error. This establishes both the positive and negative baselines for the creation endpoint before the pipeline proceeds.

5.3.3 DHT Synchronization and Rule Persistence

An important aspect of the test design is the inclusion of a `POST /api/homes/refresh` request between rule creation and rule retrieval. This endpoint forces the backend to query the live Smartotum DHT, reconcile the remote rule state with the local PostgreSQL database, and prune any orphaned records. It was included intentionally to verify that locally created `Deny` rules, which are published to the DHT

during creation, survive the full round-trip synchronization cycle without being accidentally deleted or duplicated.

After the synchronization completes with a `202 Accepted` response, the retrieval test calls `GET /api/sifis/homes/{home_uuid}/rules` and performs two checks: it validates that the response array conforms to a strict JSON Schema requiring fields like `db_id`, `type`, `trigger`, and `action`, and then searches the array for the specific `test_rule_id` persisted earlier, confirming end-to-end data integrity across the creation-sync-retrieval cycle.

5.3.4 Conflict Seeding Strategy

Testing the Petri Net conflict detection engine required a deliberate seeding approach, since the `GET .../check` endpoint analyzes an existing rule against *all other rules* in the same Smart Home. Rather than relying on pre-existing data in the database, the test pipeline creates its own conflicting rules on-the-fly:

1. **Redundancy:** A second rule is created with the exact same `Deny` effect, trigger, and action as the baseline. When the conflict check endpoint is called on this rule, the backend constructs a Semantic Colored Petri Net and identifies that the two rules produce an identical authorization outcome for the same trigger context, classifying the pair as redundant.

2. **Inconsistency:** A third rule is created with the opposing `Permit` effect but the same trigger and action. The Petri Net analysis detects that these two rules produce contradictory authorization decisions for the same context, flagging them as inconsistent.

Each seeding request extracts its own rule ID into a dedicated collection variable, and the subsequent conflict check assertions validate the structure of the response. Listing 5.13 illustrates the redundancy assertion, which verifies that the response contains a `redundantRules` array where each entry is a pair of rules sharing the same effect.

```
1 var responseBody = pm.response.json();
2 pm.test("Conflict report contains redundantRules", function () {
3     pm.expect(responseBody).to.have.property("redundantRules");
4     pm.expect(responseBody.redundantRules).to.be.an("array");
5     pm.expect(responseBody.redundantRules.length).to.be.
    greaterThan(0);
6 });
7 pm.test("Redundant pair contains both seeded rules with matching
    effects", function () {
8     const pair = responseBody.redundantRules[0];
9     pm.expect(pair).to.be.an("array").with.lengthOf(2);
```

```
10        const ids = pair.map(r => r.db_id);
11        const redundantRuleId = pm.collectionVariables.get("
      redundant_rule_id");
12        pm.expect(ids).to.include(redundantRuleId);
13        pm.expect(pair[0].effect).to.eql(pair[1].effect);
14    });
```

Listing 5.13: Postman assertion verifying that the Petri Net conflict report correctly identifies a redundancy pair with matching effects.

The inconsistency test follows the same structural pattern but asserts on the `inconsistentRules` array and verifies that the paired rules carry *opposing* effects (`Deny` vs. `Permit`).

5.3.5 XACML Translation and Binary Validation

The final positive test in the chain validates the translation engine by requesting the XACML download for the baseline rule. Unlike standard JSON API calls, this endpoint returns a binary stream. The test verifies a `200 OK` status and inspects the `Content-Type` header to assert it equals `application/zip`, confirming that the semantic extraction, XACML serialization, and ZIP packaging pipeline functions correctly.

5.3.6 Error Boundaries

The pipeline concludes with negative tests that exercise the system's resilience against malformed inputs:

- **Missing Home UUID:** Calling the rules endpoint with an empty path segment to verify the Flask router returns a clean `404 Not Found` rather than an unhandled exception.
- **Invalid Rule ID:** Calling the conflict check and download endpoints with a nonexistent numeric identifier (`999999`) to ensure the backend responds with a structured `404` error. A numeric value is required because Flask's `<int:rule_id>` route converter rejects non-numeric segments before the request reaches the application logic.

By combining positive sequential validations with explicitly crafted negative errors, the suite establishes a fully automated, replicable workflow. Running the entire Postman collection exercises the complete PTP pipeline, from rule creation through DHT synchronization, Petri Net conflict analysis, and XACML translation, ensuring that the complex backend logic remains functionally correct and resilient against regressions.

Chapter 6

Conclusion

6.1 Results Achieved

The main objective of this thesis was the migration and integration of the Policy Translation Point into the Privacy Dashboard, transforming a legacy standalone Java prototype into a integrated modern system. Through this work, several results were achieved.

The first major result was the complete redesign of the PTP backend, transitioning from a standalone Spring Boot application that relied on in-memory OWL reasoning via HermiT to a Flask-based RESTful API following the Controller-Service-Repository pattern. The new architecture leverages Owlready2 for ontology parsing, PostgreSQL for persistent storage, and a hybrid hydration-dehydration mechanism that bridges the gap between flat relational records and rich semantic objects. This design ensures that the semantic reasoning capabilities of the original prototype are preserved while gaining the scalability and maintainability of a modern web backend.

A second key result was the successful integration with the Smartotum DHT for bidirectional rule synchronization. The `RuleService` now handles both the ingestion of raw DHT payloads into ontologically validated local database records and the publication of locally created `Deny` rules back to the distributed network. This synchronization pipeline, including the fan-out mechanism for entire-home policies that target multiple devices, ensures that the PTP operates as a fully synchronized node within the SIFIS-Home ecosystem.

Third, the Petri Net conflict detection engine was successfully implemented. The implementation uses Semantic Colored Petri Nets to identify redundancies and inconsistencies across the user's ruleset. By modeling triggers as places, actions as places, and rules as transitions, the system detects convergent transitions that produce either identical or contradictory authorization outcomes, providing the

user with a conflict report before policy enforcement.

Fourth, the XACML translation pipeline was preserved and integrated into the new architecture, enabling users to download their high-level privacy rules as standardized, machine-enforceable XACML 3.0 policy files packaged in a ZIP archive.

Finally, a React-based frontend was built to provide the user interface for the PTP, featuring a multi-step Rule Editor with cascading ontology-driven selections, a card-based rule list with inline actions, and a verification report modal for conflict analysis. The entire pipeline was validated through an end-to-end Postman test suite covering rule creation, DHT synchronization, conflict seeding and detection, XACML generation, and error boundaries.

6.2 Limitations

Despite the results achieved, the current implementation presents limitations that should be acknowledged.

The first limitation concerns the tight coupling between rule persistence and DHT availability. In the current implementation, when a user creates a `Deny` rule, the system publishes it to the Smartotum DHT synchronously if the publication fails, the entire creation is rolled back, and the user receives an error. This means that if the Smart Home hub is temporarily unreachable, the user cannot create or modify any deny rules, even though the local database would be perfectly capable of storing them. A more flexible approach would decouple the local persistence from the remote publication, allowing rules to be saved locally with a pending synchronization flag.

A second limitation is the absence of unit tests for the backend services. While the end-to-end Postman test suite validates the complete pipeline through HTTP requests, the individual service methods such as the hydration logic in `RuleService`, the graph construction in `PetriNetService`, and the XML serialization in `XACMLService` lack dedicated unit-level coverage. This makes it harder to isolate regressions when refactoring internal logic that does not necessarily affect the API arrangement.

A third limitation relates to the scope of the conflict detection engine. The current Petri Net analysis covers direct redundancies and inconsistencies, namely pairs of rules that share identical triggers and either produce the same effect or opposing effects. However, it does not yet detect conflict types such as *shadowing*, where a broader rule implicitly overrides a more specific one (e.g., a rule denying all video actions in the entire home effectively makes a room-specific permit rule unreachable).

A fourth limitation is that the conflict check is currently triggered manually, on

a per-rule basis, after the rule has already been created and saved. This means a user can accumulate conflicting rules in their environment without being warned, unless they explicitly request a verification check. A more proactive approach would surface potential conflicts automatically during rule creation, presenting them as warnings before the rule is persisted.

Finally, the DHT payload format used for rule publication is limited to a simple JSON structure containing temporal fields and device identifiers. This format does not carry the full semantic richness of the original rule, which means that downstream enforcement nodes cannot access the detailed XACML policy logic directly from the DHT; they must either request the XACML file separately or rely on a simplified interpretation of the JSON fields.

6.3 Future Work

Building on the limitations presented above, several directions for future development emerge:

- **Offline rule persistence with deferred synchronization:** Decoupling the local database save from the DHT publication would allow users to create and manage rules even when the Smart Home hub is temporarily unavailable. Rules could be saved locally with a synchronization status flag (e.g., `synced`, `pending`, `failed`), and a background process could retry publication when connectivity is restored, similar to the optimistic update patterns used in modern offline-first applications.

- **Unit and integration testing:** Introducing a comprehensive unit test suite using `pytest` for the backend services would significantly improve confidence during refactoring. Key candidates for unit testing include the hydration and dehydration logic in `RuleService`, the Petri Net graph construction and traversal in `PetriNetService`, and the XML element tree generation in `XACMLService`. Integration tests targeting the database layer with an in-memory PostgreSQL instance would further strengthen the test infrastructure.

- **Expanded conflict detection:** Extending the Petri Net analysis to detect shadowing conflicts would require modeling the ontological hierarchy of actions and locations, so that the system can recognize when a broad rule subsumes a more specific one. This could be achieved by enriching the `ExtendedPetriNet` graph with inheritance arcs derived from the OWL class hierarchy.

- **Proactive conflict warnings:** Integrating the conflict check into the rule creation flow itself, rather than offering it as a separate post-hoc action, would improve the user experience. When a user submits a new rule, the backend

could run the Petri Net analysis before persisting and return any detected conflicts as warnings in the creation response, giving the user the option to proceed or revise their rule.

- **XACML DHT payloads:** An alternative to the current simplified JSON format would be to embed the full XACML policy document directly in the `value` field of the DHT entry. This would enable downstream enforcement nodes to consume standardized, semantically complete policies without requiring a separate translation step. However, this change would require coordination with the Smartotum DHT development team to update the consumer-side parsing logic to handle the new format.

With these improvements in place, the Policy Translation Point could evolve from its current functional prototype into a more robust, production-ready component of the Privacy Dashboard ecosystem, capable of operating reliably across diverse Smart Home environments and providing complete, proactive privacy policy management.

Bibliography

[1] Serap Türkyılmaz and Erkut Altindag. «Analysis of smart home systems in the context of the internet of things in terms of consumer experience». In: *International Review of Management and Marketing* 12.1 (2022), p. 19 (cit. on p. 7).

[2] 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).

[3] 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).

[4] 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).

[5] 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).

[6] 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).

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

[8] *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).

[9] *La domotica del futuro passa da Smartotum, spin-off del Politecnico*. Politecnico di Torino. Mar. 27, 2024. URL: https://www.polito.it/ateneo/comunicazione-e-ufficio-stampa/poliflash/la-domotica-del-futuro-passa-da-smartotum-spin-off-del (visited on 03/01/2026) (cit. on p. 10).

[10] OASIS. *eXtensible Access Control Markup Language (XACML) Version 3.0*. OASIS Standard. OASIS, Jan. 2013. URL: http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.html (cit. on pp. 13, 14).

[11] Pascal Hitzler, Markus Krötzsch, and Sebastian Rudolph. *Foundations of Semantic Web Technologies*. Boca Raton, FL: CRC Press, 2009. ISBN: 9781420090505. DOI: 10.1201/9781420090512 (cit. on p. 15).

[12] Boris Motik, Peter F. Patel-Schneider, and Bijan Parsia. *OWL 2 Web Ontology Language Structural Specification and Functional-Style Syntax (Second Edition)*. W3C Recommendation. W3C, Dec. 2012. URL: https://www.w3.org/TR/owl2-syntax/ (cit. on p. 15).

[13] Jean-Baptiste Lamy. *Ontologies with Python: Programming OWL 2.0 Ontologies with Owlready2*. Berkeley, CA: Apress, 2021. ISBN: 978-1-4842-6551-2. DOI: 10.1007/978-1-4842-6552-9 (cit. on pp. 15, 16).

[14] Jean-Baptiste Lamy. «Owlready: Ontology-oriented programming in Python with automatic classification and high level constructs for biomedical ontologies». In: *Artificial Intelligence in Medicine* 80 (July 2017). ISSN: 0933-3657. DOI: 10.1016/j.artmed.2017.07.002 (cit. on p. 16).

[15] Tadao Murata. «Petri nets: Properties, analysis and applications». In: *Proceedings of the IEEE* 77.4 (Apr. 1989). ISSN: 0018-9219. DOI: 10.1109/5.24143 (cit. on pp. 17, 18).

[16] James L. Peterson. *Petri Net Theory and the Modeling of Systems*. Englewood Cliffs, NJ: Prentice Hall, 1981. ISBN: 0136619835 (cit. on p. 17).

[17] Fulvio Corno, Luigi De Russis, and Alberto Monge Roffarello. «Empowering End Users in Debugging Trigger-Action Rules». In: *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. CHI '19. New York, NY, USA: ACM, 2019, pp. 1–13. DOI: 10.1145/3290605.3300618 (cit. on p. 18).

[18] SIFIS-Home Consortium. *D4.2 - Initial Design and Development of Privacy Aware Analytics for Secure Services*. Project Deliverable. Secure Interoperable Full-Stack Internet of Things for Smart Home. European Commission, Horizon 2020 Project GA 952652, 2021. URL: https://www.sifis-home.eu/deliverables/ (cit. on pp. 18, 19).

[19] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures*. University of California, Irvine, 2000 (cit. on pp. 33, 34, 41).

[20] R Fielding, M Nottingham, and J Reschke. *RFC 9110 HTTP Semantics*. 2022 (cit. on p. 34).

[21] 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. 36, 37).

[22] Thomas M Connolly and Carolyn E Begg. *Database systems: a practical approach to design, implementation, and management*. Pearson Education, 2005 (cit. on p. 38).

[23] Jim Gray and Andreas Reuter. *Transaction processing: concepts and techniques*. Elsevier, 1992 (cit. on p. 38).

[24] Michael Stonebraker and Lawrence A Rowe. «The design of Postgres». In: *ACM Sigmod Record* 15.2 (1986), pp. 340–355 (cit. on p. 39).

[25] *2025 Stack Overflow Developer Survey*. Stack Overflow. 2025. URL: `https://survey.stackoverflow.co/2025/technology#1` (visited on 03/01/2026) (cit. on pp. 39, 41, 43).

[26] Lloyd Shirley Jerry Welch James McDonald. *FROM MONOLITH TO MICROSERVICES: A SPRING BOOT–DRIVEN APPROACH TO MODULAR BACK-END SYSTEMS* (cit. on p. 39).

[27] 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. 41).

[28] Miguel Grinberg. *Flask web development*. " O'Reilly Media, Inc.", 2018 (cit. on p. 42).

[29] 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. 42).

[30] Pete Hunt. *Why did we build React?* Accessed: 2026-03-01. June 2013. URL: `https://legacy.reactjs.org/blog/2013/06/05/why-react.html` (cit. on p. 43).

[31] 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. 44).