

TRABAJO FIN DE GRADO

Título: Desarrollo de una Plataforma de Automatización de Tareas para Smart Homes mediante Beacons

Título (inglés): Development of a Task Automation Platform for Beacon enabled Smart Homes

Autor: Sergio Muñoz López

Tutor: Carlos A. Iglesias Fernández

Departamento: Ingeniería de Sistemas Telemáticos

MIEMBROS DEL TRIBUNAL CALIFICADOR

Presidente: Mercedes Garijo Ayestarán

Vocal: Carlos Ángel Iglesias Fernández

Secretario: Álvaro Carrera Barroso

Suplente: Juan Fernando Sánchez Rada

FECHA DE LECTURA:

CALIFICACIÓN:

UNIVERSIDAD POLITÉCNICA DE MADRID

**ESCUELA TÉCNICA SUPERIOR DE
INGENIEROS DE TELECOMUNICACIÓN**

Departamento de Ingeniería de Sistemas Telemáticos
Grupo de Sistemas Inteligentes



TRABAJO FIN DE GRADO

**DEVELOPMENT OF A TASK
AUTOMATION PLATFORM FOR
BEACON ENABLED SMART HOMES**

Sergio Muñoz López

Enero de 2016

Resumen

El auge del Internet de las Cosas abre un sinfín de posibilidades para el desarrollo de nuevos servicios y aplicaciones. Este concepto se refiere a la interconexión de objetos de uso diario, que son equipados con inteligencia ubicua. Su aplicación al escenario de las casas inteligentes proporciona una solución simple y efectiva, y puede proporcionar la capacidad de monitorizar, controlar y automatizar actividades frecuentes, mejorando la comodidad y accesibilidad de los usuarios.

El objetivo final de este proyecto es el desarrollo de una plataforma inteligente de automatización basada en reglas ECA (Evento-Condición-Acción). El proyecto implementa y define la arquitectura de la plataforma de automatización basada en un motor de reglas y automatizará eventos con distinto origen. Para este propósito, se han desarrollado varios módulos: un servidor de automatización de tareas, una aplicación móvil y una aplicación para Google Glass. Este proyecto se centra en los dos primeros.

El servidor de automatización de tareas está compuesto por varios sub-módulos que proporcionan funciones de gestión de reglas y canales, un motor de reglas donde son evaluadas las reglas y los eventos para generar una acción, y un disparador de acciones.

La aplicación móvil recibe eventos provenientes de los beacons y de otros canales del dispositivo, los procesa y los envía al servidor donde serán evaluados. Además, este módulo también proporciona un disparador de acciones.

El sistema desarrollado en este proyecto ha sido aplicado a un caso real de automatización de tareas en una casa inteligente con beacons.

Para finalizar, se recogen las conclusiones extraídas del proyecto, así como los problemas enfrentados durante el desarrollo y las posibles líneas de trabajos futuros.

Palabras clave: Automatización, Beacons, Tecnologías semánticas, RDF, EYE, Android, PHP, JavaScript, EWE, MongoDB

Abstract

The emerging Internet of Things opens endless possibilities for the development of new services and applications. The Internet of Things refers to the networked interconnection of everyday objects, which are frequently equipped with ubiquitous intelligence. Its application to the smart home scenario is simple yet effective, and can provide the ability to monitor, control and automate frequent activities, improving users comfort and accessibility.

The objective of the final project thesis is the development of an intelligent automation platform based on ECA (Event-Condition-Action) rules. The project defines and implements the architecture of the automation platform based on rule inference engine and will automate internet and contextual event sources. For this purpose, several modules have been developed: Task Automation Server, Mobile App and Google Glass App. This project is focused on the first two.

Task Automation Server is composed of several sub-modules that provide rules and channels management functions, a rule engine where rules and channel events are evaluated in order to generate an action, and an action trigger.

Mobile app receives events from beacons or other devices channels and sends them to the Task Automation Server for be evaluated. In addition, this module is also able to trigger actions generated in response to this events.

The system developed in this project has been integrated in a real case, automating tasks in a beacon enabled smarthome.

Finally, the extracted conclusions from this project are gathered, as well as the problems faced during the development and the possible lines of future work.

Keywords: Automation, Beacons, Semantic technologies, RDF, EYE, Android, PHP, JavaScript, EWE, MongoDB

Agradecimientos

Aunque como autor de este proyecto figure solo una persona, la realidad es que en él han contribuido muchas más. Este proyecto es también suyo, y me gustaría agradecerles el inmensurable apoyo que me han mostrado en todo momento.

A mi madre y a mi padre, que son los principales responsables de este proyecto y de todo lo que pueda conseguir en mi vida; por haberme convertido en todo lo que hoy soy, haberme dado todo lo que hoy tengo y haberme enseñado las cosas más importantes que hasta hoy he aprendido.

A mis hermanas Maricarmen y Rocío, que son el ejemplo en el que se inspira mi vida; por labrar el camino que yo solo he tenido que seguir y aun así preocuparse siempre de impulsarme hacia adelante.

A María, que entró en mi vida y la convirtió en música, y en nuestra; por ser el motor que me empuja con la ilusión de nuestro futuro a seguir mejorando nuestro presente.

A los amigos y amigas que me han acompañado durante todos estos años y los han llenado de experiencias inolvidables.

A Carlos, por su confianza al permitirme formar parte del grupo y por la ayuda prestada en el desarrollo de este proyecto.

A todos los compañeros del Grupo de Sistemas Inteligentes, por la enorme fuente de aprendizaje que están resultando para mí.

Contents

Resumen	V
Abstract	VII
Agradecimientos	IX
Contents	XI
List of Figures	XV
1 Introduction	1
1.1 Motivation	1
1.2 Project goals	3
1.3 Structure of this document	4
2 Enabling Technologies	5
2.1 Beacons	5
2.1.1 Architecture and operation	5
2.1.2 Estimote SDK for Android and iOS	7
2.1.3 Bluetooth Low Energy (BLE)	8
2.1.4 Eddystone	8
2.1.5 Proximity Beacon API	8
2.2 Rule Automation	9
2.2.1 Task Automation Services	9

2.2.2	Notation3	10
2.2.3	EYE	11
2.2.4	EWE Ontology	11
2.2.4.1	Channel	12
2.2.4.2	Event	12
2.2.4.3	Action	13
2.2.4.4	Rule	13
2.3	MongoDB	13
3	Architecture	15
3.1	Overview	15
3.2	Task Automation Server	17
3.2.1	Rule Engine	18
3.2.2	Rule Administration	19
3.2.2.1	Rule Editor	19
3.2.2.2	Rule Manager	22
3.2.2.3	Rule Repository	23
3.2.3	Channel Administration	23
3.2.3.1	Channel Editor	25
3.2.3.2	Channel Manager	27
3.2.3.3	Channel Repository	28
3.2.3.4	Events Manager	28
3.2.4	Action Trigger	31
3.3	Mobile App	31
3.3.1	Contextual Channel	32
3.3.2	Device Channel	33
3.3.3	Rule Administration	34

3.3.4	Action Trigger	34
4	Case study	35
4.1	Login and events configuration	35
4.2	Channel Edition	36
4.3	Rule Edition	38
4.4	Rule Import	40
4.5	Case Study	40
5	Conclusions and future work	43
5.1	Conclusions	43
5.2	Achieved goals	44
5.3	Problems faced	45
5.4	Future work	46
	Bibliography	47
A	Rules and channels creation	49
A.1	Channels definition	49
A.2	Rules definition	52

List of Figures

2.1	Approximate beacon's range	6
2.2	Top of the line beacons	6
2.3	EYE integration	11
2.4	Evented WEb Ontology (EWE) Class Diagram	12
3.1	Architecture	16
3.2	TAS Sub-modules Interconnection	17
3.3	Actions triggering Process	18
3.4	Rules creation and load processes	19
3.5	Rule Editor Interface	20
3.6	Rule Manager Interface	22
3.7	Creation of a rule	24
3.8	Channel Editor Interface	25
3.9	Channel Manager Interface	28
3.10	Events Manager Interconnection	29
3.11	Mobile App Graphic Interface	33
4.1	Login form	36
4.2	Internet channels connection	36
4.3	Channels page	37
4.4	Channels Editor interface	37
4.5	Rules page	38

4.6	Event Selection	39
4.7	Save Rule	39
4.8	Beacons placing	40
4.9	Eye Client Tool	42
4.10	Simulator Tool	42

Introduction

1.1 Motivation

Over the last few years, and with increasing intensity, terms like 'Internet of Things (IoT)' or "smarthome" are permeating society at the pace they start to become a reachable reality.

The idea of implement in everyday objects the necessary technology so they can exchange data with the user or with other objects, would open endless possibilities for humans. To apply this idea to the home scope, is one of the largest and most interesting fields in IoT. The ability to monitor, control and automate some of the simplest acts of daily life such as turning on the light or the air conditioning is interesting not only from the comfort standpoint, but also regarding the tremendous improvement in accessibility that it entails.

This is the fact that we consider in this project, to take advantage of the opportunities created by the data exchange between household objects and to offer the users some of its many applications.

Although the "home automation" concept was conceived nearly a century ago, there had been some technological limitations up to now that made it nothing more than a dream for the average consumer. Nowadays, with the advancement in wireless technologies, this

idea has become an increasingly viable possibility. The integration of technology devices and services through home networking for a better quality of life is called '*smart home technology*'.

A smarthome is a house where comfort, entertainment and aesthetic needs of humans are ruled by technology. For example, environmental parameters such as light or temperature can be regulated smartly to capture the user's needs and act accordingly. In a smarthome, a person can have all necessary services to his disposal with the ease of a click of buttons or gestures.

Smarthomes aim is to make life easier and more convenient to humans. While a person is away from home, the smarthome can alert when something unusual happens, and security systems can be designed to provide lots of help in an emergency such as fire notifications, automatically open doors, communicate the situation to fire department or guide the occupants through the safest exit route.

Continuing the earlier example, related to environmental parameters, little things like lights automatically turn off when a person leaves a room, or automatic temperature control in a room depending on who are inside, would be a significant energy saving.

Smart home technology also offers significant benefits to elderly or disabled people, and can provide them with more independence. Possible examples are to alert the hospital in case of fall, to facilitate daily tasks as simple as making the food, or to give real time information to their relatives.

To fully understand the true potential of this technology, we proceed to describe the different elements that may be part of smart homes:

- **Sensors:** monitor and measure surrounding activity. For example, temperature, movement, or smoke sensors.
- **Actuators:** perform physical actions. Examples: window and door openers, automatic light switches.
- **Drivers:** make decisions based on programmed rules and occurrences. They receive and process sensor's values. For example, a thermometer controller can be programmed to send an order to turn on the heater if the temperature falls below 20 degrees or to open the window if rises 35.
- **Central processing unit:** although not always necessary, sometimes it can be useful for maintenance and system changes. This central unit could be a cloud server.

- **Network:** is the transmitter of the signals in the system. Smarthome's all modern systems have a bus-based network. In such networks, all units of the system can read all messages. These messages include the address of devices that should receive them. For example, in some cases an order for lights can be directed to a particular lamp, while at other times must be received by all system lamps.

To achieve the objectives of this project, we will use an emerging mobile technology based on a small battery-powered devices called beacons, which use Bluetooth as transmission medium. Numerous leisure centers, museums, airports, hospitals and sports complexes are currently experimenting with beacons for a variety of purposes depending on the context. These devices are increasing their popularity thanks to the enormous improvement of the user experience that they offer in spite of its low cost and easy integration, as well as they are supported by most smartphones. Using *BLE*, beacons can be designed to run for years with a button battery.

1.2 Project goals

The main goal of this project is the development of an intelligent automation platform based on ECA (Event-Condition-Action) rules. The project will define and implement the architecture of the automation platform based on rule inference engine and will automate internet and contextual event sources.

This main goal includes some tasks such as:

- Design and build a web server that allows users to create, edit, obtain and remove event driven rules for task automation.
- Develop a mobile application that receives data from beacons (contextual event sources) or device channels and connects with the server.
- Build a software module which is able to connect with web apps like Google Calendar or Twitter (internet event sources).
- Connect with a rule engine which is able to run the event driven drules and handle its response.
- Connect with action triggers in order to run actions generated by the rule engine.

1.3 Structure of this document

In this section we provide a brief overview of the chapters included in this document. The structure is the following:

Chapter 1 is an introduction for the project, here both the context in which it is developed and its main goals are described.

Chapter 2 provides a description of the main standards and technologies on which this project rely.

Chapter 3 explains the complete architecture and all the components and modules of the system.

Chapter 4 offers an overview of the main use case. The running of the modules to offer the detailed functionalities are explained, such as the steps that the user has to follow to automate tasks using this system.

Chapter 5 sums up the conclusions drawn from this project, problems faced and suggestions for a future work.

Enabling Technologies

2.1 Beacons

Beacons are small battery devices able to emit a BLE signal that can be captured by smartphones situated inside its broadcast range devices. This signal is often relayed to a cloud server that processes the information and treated it according to its nature. Information travels by this signal and can be anything from environment data (temperature, air pressure, humidity) to indoor positioning data (asset tracking, retail) or orientation (acceleration, rotation).

The applications of these devices are very numerous, including indoor navigation, location based marketing, location based customer service, clienteling, and personalized assistance.

2.1.1 Architecture and operation

The hardware consists of a microcontroller chip with BLE and a battery. The battery is usually a button battery, but it depends on the manufacturer and the model.

On the other hand, regarding the firmware, we find that each beacon has a specific

one. The main characteristics which controls the firmware, and that can sometimes be configurable, are the transmission power and the transmission interval of the signal. The latter is usually between 350 and 900 ms, while the power is usually about -12 dBm.

Here we can see a graph [1] of the approximate range depending on the transmission power of the beacon.

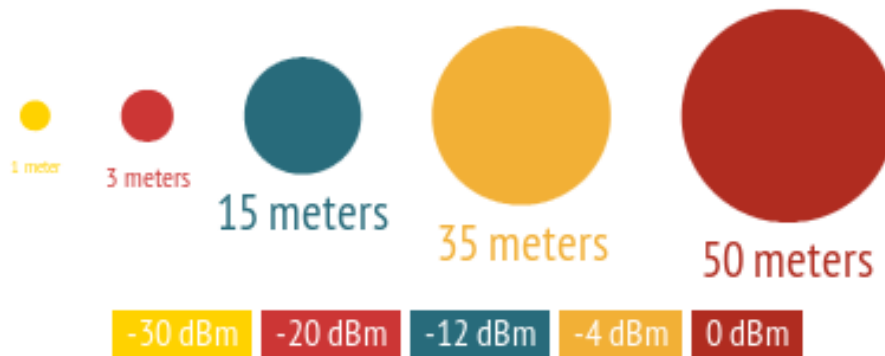


Figure 2.1: Approximate beacon's range

The signals emitted by the beacons can be captured by devices with Bluetooth interface, but only for those that have installed a tracer application. This application is responsible for processing the information contained in the signal or sending it to a cloud server that act accordingly to them.

Actually there is a wide range of beacons available on the market, and in order to find out which is the best suited to our needs we have used the Aislelabs comparative study [1] of some of the biggest manufacturers. Here are the highlights of this report:



Figure 2.2: Top of the line beacons

Having seen the results of this study, we have opted for the use of *Estimote Beacons*. The primary reasons for choosing this manufacturer, were these:

- They offer both Android and iOS SDK.
- Battery lasts more than two years.
- Easy integration.
- Indoor SDK.
- Proximity API and Eddystone support.
- Updatable firmware.
- Configurable output power and advertising interval.

With regard to technical specifications, they have a 32-bit ARM® Cortex M0 CPU that is accompanied by accelerometer, temperature sensor, and 2.4 GHz radio using BLE. Estimote Beacons stand out from the rest in design, integration (incorporating SDK for Android and iOS) and autonomy, so we consider that they are perfect for this project.

Furthermore, they are compatibles with all of the below explained technologies that facilitate the integration and application development for beacons.

2.1.2 Estimote SDK for Android and iOS

The Estimote SDK is a library that allows interaction with Estimote beacons and stickers. The SDK system works on Android 4.3 and iOS 7 or above, and requires device with Bluetooth Low Energy (SDK's min Android SDK version is 9).

It allows for:

- Scanning beacons and filtering them by their properties.
- Monitoring regions for those devices that have entered/exited a region.
- Nearables (aka stickers) discovery.
- Eddystone scanning.
- Easy way to meet all requirements for beacon detection (runtime permissions, acquiring all rights).
- Beacon characteristic reading and writing (proximity UUID, major and minor values, broadcasting power, advertising interval).

2.1.3 BLE

Bluetooth Smart or *Bluetooth Low Energy* is recognized as the key in the development of IoT, because of its ability to open the door to a much more intelligent environment with many devices connected to our smartphones.

This technology is based on reducing consumption and, therefore, the major expansion of autonomy that this entails. This enhanced autonomy positions it as the key standard for the development of all type of sensors, including beacons.

In terms of technical features, BLE provides 1 Mbps connection speed, encrypted connections using AES 128-bit and redundancy codes to minimize erroneous transmissions.

2.1.4 Eddystone

Eddystone [2] is a BLE protocol developed by Google with the purpose of defining a message format for beacons. The most important feature of this protocol is that, unlike its counterpart developed by Apple (iBeacon), it is able to run on both Android and iOS applications. Its other great feature is that it supports different frame types that can be used for a wide variety of applications.

2.1.5 Proximity Beacon API

Proximity Beacon API [3] is a part of the BLE platform for beacons, including Eddystone. It is based on a cloud service that allows the management of data associated with the beacons using a REST interface.

This API allows you to associate data to the registered beacons as attachments. The attached are data blobs that are returned as messages to the Android or iOS application through the API Nearby Messages. You can remotely update these attachments, eliminating the need to physically visit each beacon

In short, this API enables to register beacons in the cloud, associate data with registered beacons as attachments, and maintain beacon registration data.

2.2 Rule Automation

Rules are representations of knowledge with conditions in some domains of logic, such as first-order logic. A rule is basically defined in form of If-then clauses containing logical functions and operations, and can be expressed in rule languages or formats. Generally, a rule consists of antecedence and consequence containing clauses, and logical quantifiers used to quantify possible domains of variables. The antecedence contains conditions combined using logical operators, while the consequence part contains conclusions (or actions). If all the conditions are matched, the conclusions are operated. A clause is in the form of subject-relation-object, where subject and object can be variables, individuals, literal values or other data structures [4].

A number of now prominent web sites, mobile and desktop applications feature rule-based task automation. Typically, these services provide users the ability to define which action should be executed when some event is triggered. Some examples of this simple task automation could be “When I am mentioned in Twitter, send me an email” or “When I reach 500 meters of this place, check in in Foursquare”. We call them Task Automation Services (TASs).

2.2.1 Task Automation Services

These services provide a visual programming environment, where non-technical users can seamlessly create, manage or even share their own personal automations. The automation in these services takes the form of Event-Condition-Action rules that execute an action upon a certain triggering event i.e. “when triggering-event then do action”. In the former examples, the Twitter mention and the reach of 500 meters of a place would be the triggers, whereas sending an email and checking in in Foursquare are the respective actions. The use of TASs has grown substantially, own to three main factors: usability (TASs provide a simple-yet-powerful intuitive interface for programming task automations), customisability (TASs allow their users to program the automations they need) and integration with existing Internet services (users can automate tasks that access the Internet services they already use and are familiar with) [5].

Since multiple TASs exist, the execution strategy defined as rules is completely independent of the actual reasoning platform. Moreover, as both decision trees and rule-based reasoning follow the same kind of thinking (i.e., a sequence of ‘when this then that’ steps), writing rules to specify decision trees comes very natural. The used engine should be very

expressive, to maximize configurability, and there's a language that stands out from the rest in this point: Notation3.

2.2.2 Notation3

Notation3 (N3) is a Semantic Web logic. Being a superset of Turtle – a serialization format of RDF data – it is capable of describing everything using triples, but also capable of describing rules to be executed on those triples. The aims of N3 are:

- To optimize expression of data and logic in the same language.
- To allow RDF to be expressed.
- To allow rules to be integrated smoothly with RDF.
- To allow quoting so that statements about statements can be made.
- To be as readable, natural, and symmetrical as possible.

The language achieves these with the following features:

- URI abbreviation using prefixes which are bound to a namespace (using @prefix) a bit like in XML.
- Repetition of another object for the same subject and predicate using a comma “,”.
- Repetition of another predicate for the same subject using a semicolon “;”.
- Bnode syntax with a certain properties just put the properties between [and].
- Formulae allowing N3 graphs to be quoted within N3 graphs using { and }.
- Variables and quantification to allow rules, etc to be expressed.
- A simple and consistent grammar.

N3 differentiates itself from other rule languages because of its expressiveness. For example, in N3 it is possible to create rules in the consequence, and to use built-ins. The N3 logic has monotonicity of entailment, which means that the hypotheses of any derived fact may be freely extended with additional assumptions, which is an important property when reasoning about a changing knowledge base [6]. The expressiveness of rule-based reasoning engines depends on which logic the underlying programming language supports, and on the inherent expressiveness of the rule language. Some engines like RDFox, FuXi7 or EYE support N3 syntax, but EYE reasoner is the only which support all N3's expressivity.

2.2.3 EYE

The EYE (Euler YAP Engine) [7] reasoner is a high-performance reasoning engine that uses an optimized resolution principle, supporting forward and backward reasoning and Euler path detection to avoid loops in an inference graph. It is written in Prolog and supports, among others, all built-in predicates defined in the Prolog ISO standard. Backward reasoning with new variables in the head of a rule and list predicates are a useful plus when dealing with OWL¹ ontologies, so is more expressive than RDFox or FuXi, whilst being more performant than other N3 reasoners.

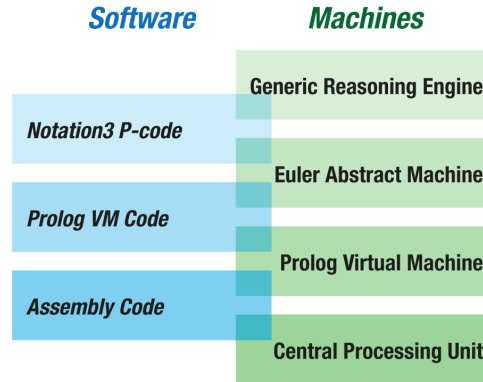


Figure 2.3: EYE integration

Internally, EYE translates the supported rule language, Notation3, to Prolog Coherent Logic intermediate code and runs it on YAP (Yet Another Prolog) engine, a high-performance Prolog compiler for demand-driven indexing. The inference engine supports monotonic abduction-deduction-induction reasoning cycle. EYE can be configured with many options of reasoning, such as not proving false model, output filtering, and can also provide useful information of reasoning, for example, proof explanation, debugging logs, and warning logs. The inference engine can be added new features by using user-defined plugins.

2.2.4 EWE Ontology

EWE is a standardized data schema (also referred as “ontology” or “vocabulary”) designed to describe elements within TASs enabling rule interoperability. Referring to the EWE definition [8], the goals of the EWE ontology to achieve are:

¹Web Ontology Language: the ontology description standard as defined by the World Wide Web Consortium (W3C)



definitions are not bound to certain Channels since different services may generate the same events [9].

2.2.4.3 Action

This class defines an operation or process provided by a Channel. Actions provides effects whose nature depend on itself. These include producing logs, modifying states on a server or even switching on a light in a physical location. By means of input parameters actions can be configured to react according to the data collected from an Event. These data are the output parameters.

2.2.4.4 Rule

The class Rule defines an “Event-Condition-Action” (ECA) rule. This rule is triggered, and means the execution of an Action. Rules defines particular inter-connections between instances of the Event and Action classes; those include the configuration parameters set for both of them: output from Events to input of Actions.

2.3 MongoDB

MongoDB² is an open source database that uses a document-oriented data model. Instead of using tables and rows as in relational databases, MongoDB is built on an architecture of collections and documents. Documents comprise sets of key-value pairs and are the basic unit of data in this database. Collections contain sets of documents and function as the equivalent of relational database tables.

Like other NoSQL databases, MongoDB supports dynamic schema design, allowing the documents in a collection to have different fields and structures. The database uses a document storage and data interchange format called BSON, which provides a binary representation of JSON-like documents. Automatic sharding enables data in a collection to be distributed across multiple systems for horizontal scalability as data volumes increase.

MongoDB provides high performance, high availability, and easy scalability.

²<http://www.mongodb.org/>

Architecture

3.1 Overview

In this chapter, the architecture of this project will be explained, including the design phase and implementation details. First of all, the system is divided into several modules to help its understanding, and later every module will be clarified in detail.

The global project is composed of the following three main modules, that are split into several sub-modules:

- **Task Automation Server (TAS)**: its main role is to handle internet and contextual events and to trigger accordingly an action generated by the rule engine, but also includes several functions for managing rules. This module is split into four sub-modules:
 - Rule Engine
 - Rule Administration
 - Channel Administration
 - Action Trigger

- **Mobile App:** this module handles events coming from beacons or from the own device, and send them to the TAS. It is also able to connect with the server for managing rules.
- **Google Glass App:** this module works as an intermediary between beacons and the server. It connects with beacons and sends its data (contextual events) to the TAS.

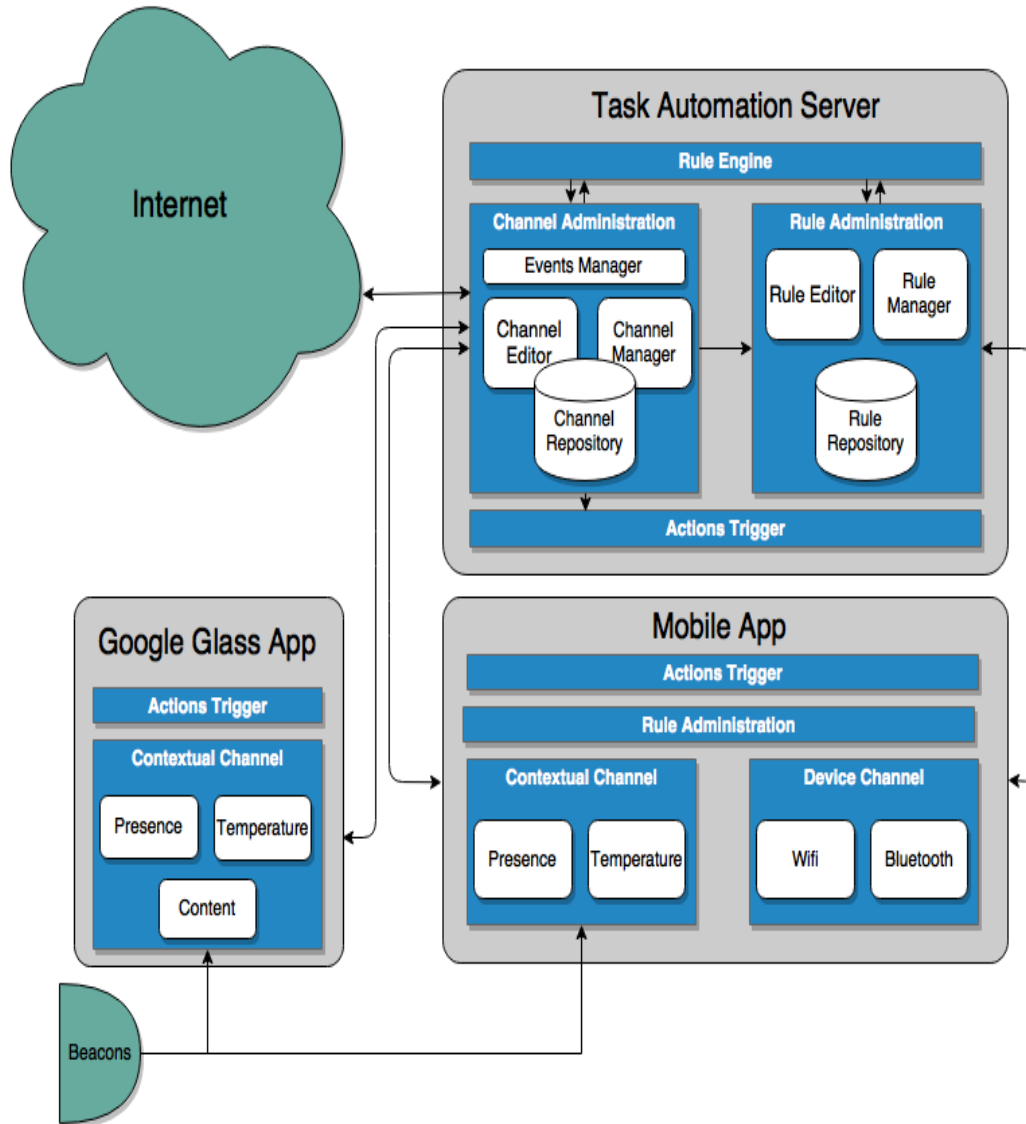


Figure 3.1: Architecture

Beacons and Internet are sources of events. These events are passed to the Channel Administration of the TAS via the Mobile and Google Glass Apps in the case of events coming from beacons and the own device, or directly in the case of Internet. Once the

TAS receives the events, generates actions that will be triggered by the Trigger Actions sub-modules. These actions are generated by the evaluation of rules along with events. Rules can be defined via the TAS or via the Mobile App.

This project is mainly focused on the TAS and the Mobile App.

3.2 Task Automation Server

The main purpose of this project is to handle all events and to generate an action in response. This action is generated by the **Rule Engine** sub-module and can be triggered either on the mobile application or on the server, depending on its nature. The Rule Engine is the core of the TAS, as it provides the main goal of the module. However, a sub-module to manage the rules and to supply several functions such as creation or erasing is also needed. **Rules Administration** is the sub-module that has this role, and it's a web application that has been implemented with both client and server side technology. As explained above, the events and actions of the rules are generated by channels; so is required also a **Channel Administration** sub-module which handles them. Furthermore, this module handle the events and is responsible for sending the response with the actions to the **Action Trigger**, the module that triggers them.

Once the sub-modules that compose the Server have been introduced, and before going deeper into them, it's appropriate to explain the interconnection between them. For this purpose is intended the following diagram:

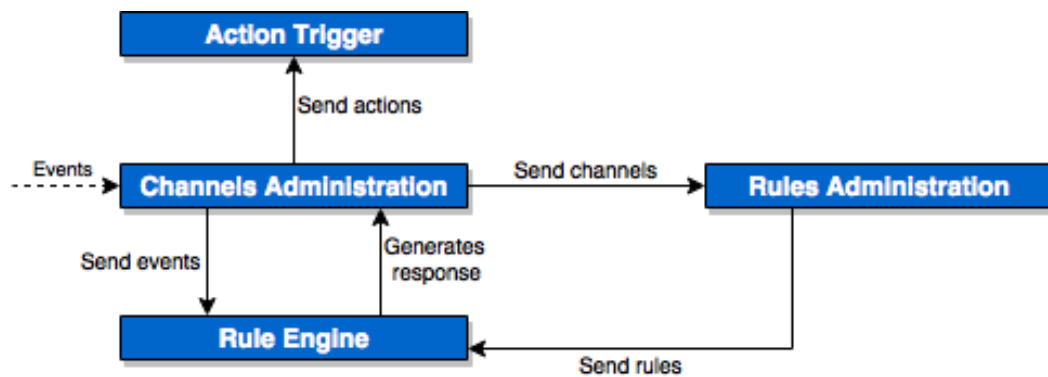


Figure 3.2: TAS Sub-modules Interconnection

In the diagram the interconnection between the sub-modules can be seen. Rule Engine receives events from the Channels Manager and rules from Rules Administration, and generates a response that contains the actions which will be triggered by the Action Trigger. Rules Administration receives channels with their respective events and actions definitions

from Channel Administration.

3.2.1 Rule Engine

Rule Engine is one of the most important modules in this project, and is based on ontology model, which uses the EWE ontology [5]. It is divided into two parts: EYE Server and EYE Helper.

- EYE Server is implemented in Javascript. It is an Euler Yap Engine [7] reasoner. It processes the events and rules written in Notation3 and generates accordingly a response with an action. This response is generated in text format.
- EYE Helper, implemented in PHP, is responsible for the reception of the events from the Channel Administration and the load of rules that are stored in the repository available on the Rules Administration module. Once it retrieves the events and the rules, it sends them to the EYE Server for being processed and receives the response. This response contains the actions that must be triggered, so EYE Helper sends them to Channel Administration.

The process of handling events and rules and triggering actions, is shown in the next figure:

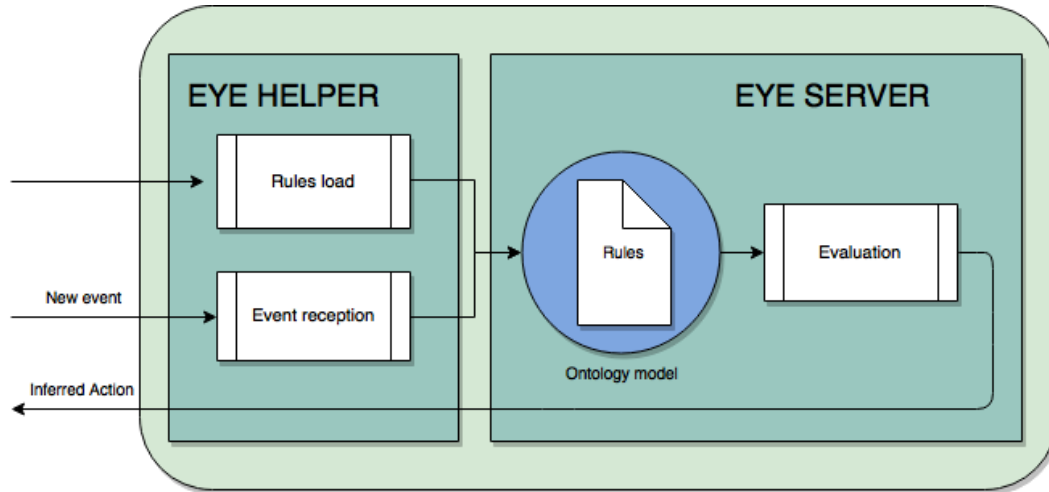


Figure 3.3: Actions triggering Process

When a new event is received, EYE Helper captures it and loads the available rules. Then, events and rules are sent to the EYE Server, that runs the ontology model inferences. This process combines the loaded rules and the inserted events, and draws conclusions from

them. These conclusions from the inserted events and loaded rules of the ontology model, represented by triples, are the actions. The response is generated using Notation3 in a text format, so it will have to be parsed later.

It's important to describe the management of the resources inside the ontology model. EYE Server is stateless [10], so all the events and rules will have to be load before a new inference. Once an event has been inserted into the ontology model and the inferences have been made, this event and the rules are removed from the model. In the same way, the action triples are resources of the ontology model, and have no temporal reasoning neither: are inferred and then removed from the ontology model.

3.2.2 Rule Administration

The main purpose of this module is to provide an automation rule editor in which users can configure and adapt their preferences about Internet and contextual events, and it is also the module that provides the stored rules to the Rule Engine. It is divided into three main parts: Rule Editor, Rule Manager and Rule Repository.

The processes of rules creation and load, are represented by the following diagram:

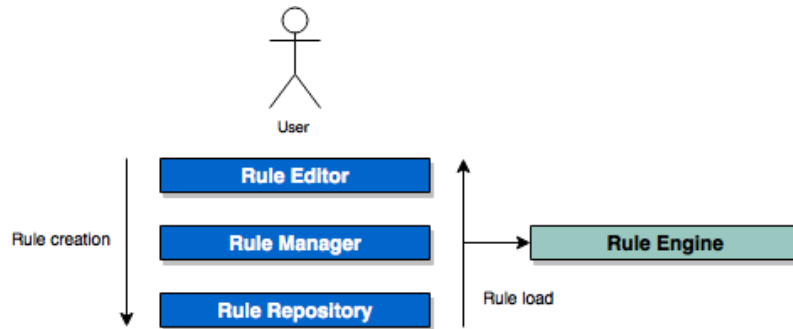


Figure 3.4: Rules creation and load processes

3.2.2.1 Rule Editor

This module, implemented with client side technologies like Javascript, HTML5, and CSS; provides a graphical interface, based on icons and “drag and drop” actions, for automation rules edition. The main purpose has been the creation of an interface that makes easy and fast the process of creating, removing or editing an event driven automation rule.

As explained before, a channel is a source of events and actions. For example, a door can be a contextual channel, and can provide events such as “Door has been opened”, that

can trigger actions. Another example of contextual channel is a Smart TV, that can provide actions such as “Display an image”. The connection between events and actions through the structure “If this then that”, results in an automation rule. Referring to the example above, the rule resulting from the action and event is “If the door is opened, then display an image on Smart TV.”. The creation, edition and deletion of this type of rules is the main role of Rule Editor.

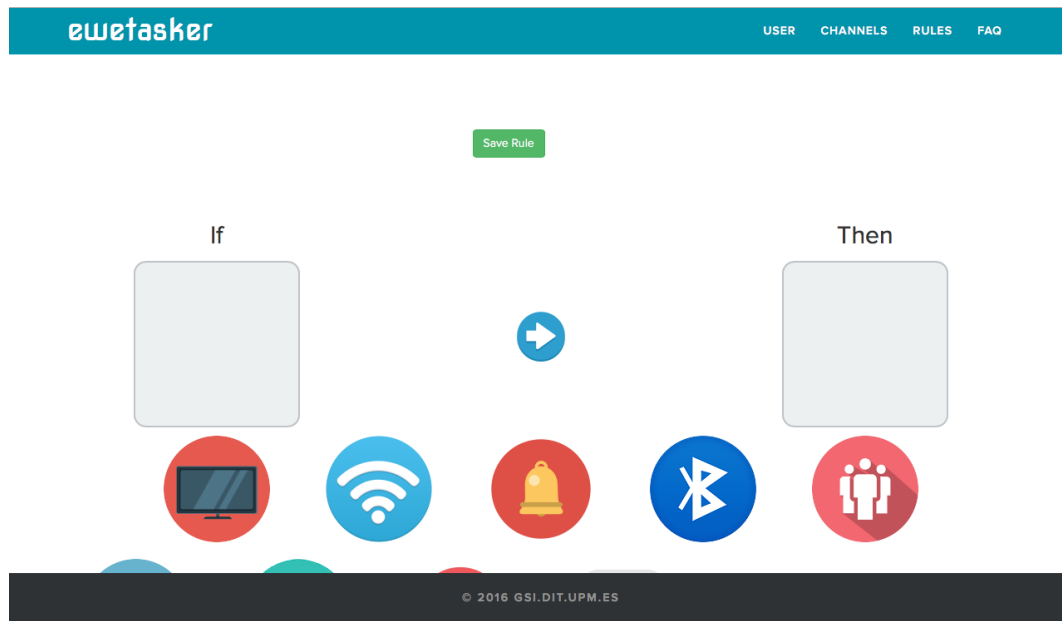


Figure 3.5: Rule Editor Interface

The above figure shows the graphical interface for the rules creation. The user can select the channel, drag it and drop onto one of the two containers. The left container is for the channel which provides events, and the right container is for the channel which provides actions. Once the user has dropped the channel onto the container, will be shown a dialog for choosing the event or the action (depending on the container). After having chosen the event and the action, when the button Save Rule is clicked, will be shown a dialog with several fields that have to be filled:

- **Title:** an identifier for the rule.
- **Place:** where the events of the rule are captured. For example, a certain room in a smart home. This field is important because the user is able to configure which rules wants to evaluate, depending on its place.
- **Creator:** the user that has created the rule.
- **Description:** a brief explanation of the rule.

- **Params fields:** as will be explained below, each event and action can have certain params that have to be setted. Here appears as much fields as params has the rule.

Once the user has filled these fields, the rule can be defined properly, and Rule Editor has to store all the information relating this process. For this purpose, it uses a structure of objects that will be sent to the server, in order of communicate the created rule to Rule Manager. The following schema shows this structure.

Listing 3.1: "Rule structure"

```
{
  "title": "Example Rule",
  "place": "GSI",
  "description": "If presence is detected, then switch on TV.",
  "created_by": "Sergio",
  "event_channel": "PresenceSensor",
  "action_channel": "SmartTV",
  "event":{
    "title": "PresenceDetectedAtDistance",
    "prefix": "@prefix ewe-presence: <http://gsi.dit.upm.es/ontologies/ewe-
      connected-home-presence/ns/#> .",
    "eye_fragment": "?event rdf:type ewe-presence:
      PresenceDetectedAtDistance."
  },
  "action":{
    "title": "SwitchON",
    "prefix": "@prefix ewe-smarttv: <http://gsi.dit.upm.es/ontologies/ewe-
      connected-home-smarttv/ns/#> .",
    "eye_fragment": "ewe-smarttv:SmartTv rdf:type ewe-smarttv:SwitchOn ."
  },
  "eye_rule":"
  {
    ?event rdf:type ewe-presence:PresenceDetectedAtDistance.
  }
  =>
  {
    ewe-smarttv:SmartTv rdf:type ewe-smarttv:SwitchOn . # Switch On the
      Smart TV.
  }.",
  "created_at":"12:13 12-01-2016"
}
```

This module connects with the Rule Manager for storing the rules created by the user,

and with the Channels Manager module for getting the available channels with their events and actions.

3.2.2.2 Rule Manager

This module aims to act as a controller into its parent module, Rule Administration. The functions that Rule Administration provides, such as creating, editing, or removing automation rules; are, in the end, the same: management of rules. Rule Manager serves for this purpose, and provides persisting and managing functionalities. The module allows the user to manage the rules that have been created or edited through a graphic web interface. So the user can, via this interface, obtain a list of rules available on the repository, and check their characteristics.

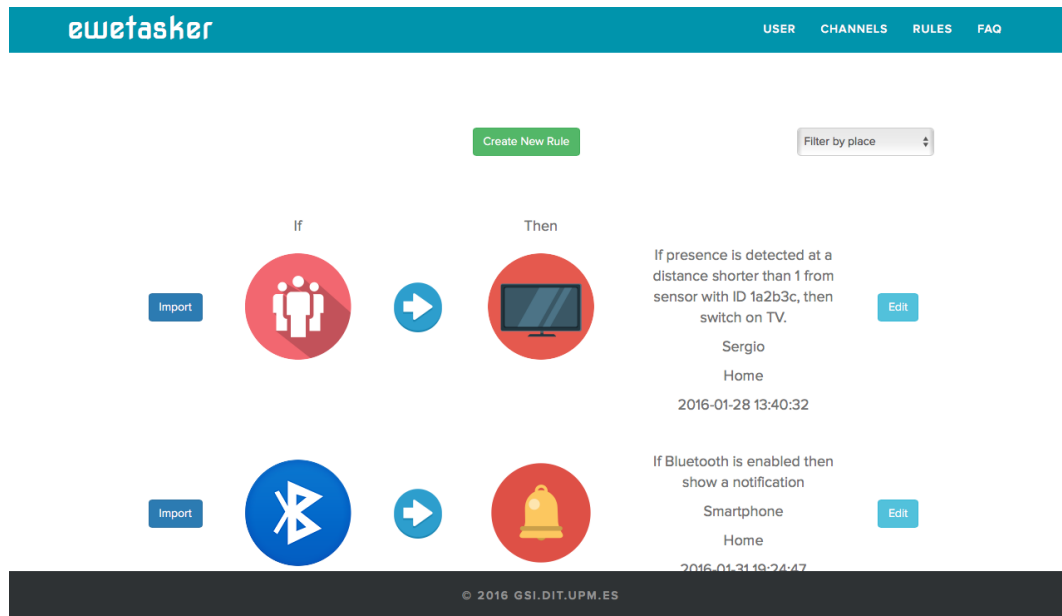


Figure 3.6: Rule Manager Interface

Rule Manager has been implemented in PHP following the MVC¹ pattern. This pattern has a specific architecture, understood as a three layered architecture: Model, View, Controller. The user accesses the pages belonging to the View. These pages have all the content that the user needs to access all the functionalities. All the pages have dynamic content which is processed by the Controller. The Controller processes manages all the data coming from the View and the Model. Besides, this layer processes the requests sent from the user: as much from requests made to the REST interface as from the View pages requests. The Model is an abstract information layer, that accesses the information under

¹<https://en.wikipedia.org/wiki/Model-view-controller>

the Controller request, creating a connection to the corresponding database, extracting the information and returning it to the Controller.

The user can also create rules via the Mobile App, due to it is directly connected with the Mobile App module. The Mobile App has a similar interface to Rule Editor module, and sends to the Rule Manager the rule attributes in a POST request. This module is also connected with the Rule Engine. When the Rule Engine module requires the rules, Rule Manager extracts them from the Rule Repository and are sent to the engine. In addition, it can be configured to extract certain rules that match with a query (like rules from a certain place or user). So, in order to provide persisting functionalities, a Rule Repository is needed.

3.2.2.3 Rule Repository

The Rule Repository is a MongoDB² database where all rules are stored. MongoDB persists the data using JSON format documents, which allows us to store rules using JSON. This is precisely the format that uses the structure of data, as explained in Rule Editor, so Rule Manager makes the management of this data in a fast and natural way.

The Mongo database is organized into collections, each one serving for one purpose. This repository has only one collection: rules. This collection stores the rules which have been created. Each document inside the collection includes a rule, and each rule has references to the channels it comes from.

To sum up, the Rule automation module is responsible for providing rules creating, editing, removing and loading functions. As explained above, channels are needed for the definition of rules, so we need also a Channel Administration module.

3.2.3 Channel Administration

This module is similar to Rule Administration, and provides an automation channel editor. Nevertheless, it also handles Internet and contextual events and pass them to the Rule Engine, where will be evaluated with the rules and will generate an action as result. This action is received by the Channel Administration too, and after being parsed, it's sent to the Actions Trigger modules, that will trigger it depending on its nature. It is divided into four main parts: Channel Editor, Channel Manager, Channel Repository and Events Manager.

²<http://www.mongodb.org/>

The process of creating a channel is very similar to the process of creating a rule, but it's interesting to show the complete process of creating a rule including the creation of the channels. The following flow chart is intended to this purpose.

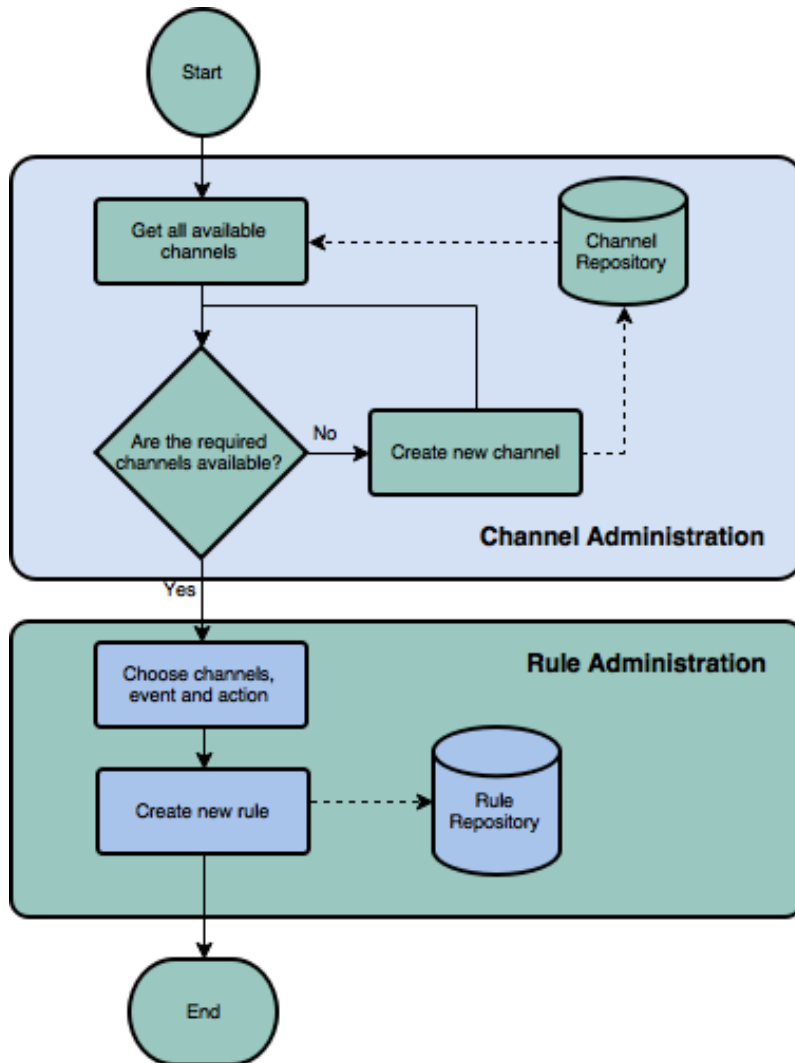


Figure 3.7: Creation of a rule

As shown, both Channel Administration and Rule Administration modules interfere in this process. When the user is going to create a rule, a list of channels (provided by the Channel Manager) is displayed. The user can select the channels with the related events and actions as explained in the Rule Administration section, but if he doesn't find the required channel, he is able to create it. Once the required channels have been defined, the process of rule creation continues in the Rule Administration module. Channels are stored in the Channel Repository.

3.2.3.1 Channel Editor

This module, as its counterpart Rule Editor, is implemented with client side technologies like Javascript, HTML5, and CSS; and provides a graphical interface for channels edition.

Figure 3.8: Channel Editor Interface

The figure 3.8 shows the graphical interface for the channel creation. The user can define the channel filling several fields such as title, description and nickname. In addition, it should upload an image that will be the channel icon in the Rule Editor interface. For the definition of each event and action, the Channel Editor interface provides a form with three fields:

- **Title:** an identifier for the action or event.
- **Rule:** this field has to be filled with the rule fragment written in Notation3 that represents the event or the action. For example, the following rule represents the event of presence detected at a certain distance:

Listing 3.2: "Rule fragment example"

```
?event rdf:type ewe-presence:PresenceDetectedAtDistance.  
?event ewe:sensorID ?sensorID.  
?sensorID string:equalIgnoringCase #PARAM_1#.  
?event!ewe:distance math:lessThan #PARAM_2#.
```

It's important to mention the inclusion of params in the rule fragment. Some events and actions can have params that must be setted when the rule is created. For making this possible, it is necessary to include the expression `#PARAM.X#` in the rule fragment. That indicates to the Rule Manager that this event or action has some params that must be configured, and the expression will be replaced with the param selected by the user.

- **Prefix:** in order to simplify the rule fragment, some large expressions are replaced with shorter prefixes. These prefixes must be included in the rule definition for its proper functioning. For example, the previous event rule fragment needs the following prefixes:

Listing 3.3: "Prefix example"

```
@prefix string: <http://www.w3.org/2000/10/swap/string#>.  
@prefix math: <http://www.w3.org/2000/10/swap/math#>.  
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .  
@prefix ewe: <http://gsi.dit.upm.es/ontologies/ewe/ns/#> .  
@prefix ewe-presence: <http://gsi.dit.upm.es/ontologies/ewe-  
connected-home-presence/ns/#> .
```

The user can add as much events and actions as he wants, by clicking on the corresponding button. Once the user has filled these fields, the channel can be defined properly, and in a similar way as Rule Editor, Channel Editor has to store all the information relating this process. For this purpose, it uses a structure of objects that will be sent to the server, in order of communicate the created channel to Channel Manager. The following schema shows this structure.

Listing 3.4: "Channel structure"

```

{
  "title": "tv",
  "description": "This channel represents a simplified Smart TV with simple
    capabilities.",
  "nickname": "Smart TV",
  "created_by": "Sergio",
  "events": {
    "event": {
      "title": "",
      "prefix": "",
      "num_of_params": "",
      "eye_fragment": ""
    }
  },
  "actions": {
    "action": {
      "title": "Switch on",
      "prefix": "@prefix ewe-smarttv: <http://gsi.dit.upm.es/ontologies/ewe-
        connected-home-smarttv/ns/#> . @prefix rdf: <http://www.w3.org
        /1999/02/22-rdf-syntax-ns#> .",
      "num_of_params": "0",
      "eye_fragment": "ewe-smarttv:SmartTv rdf:type ewe-smarttv:SwitchOn."
    }
  },
  "created_at": "12:23 15-01-2016"
}

```

In order to store the channels created by the user, this module connects with the Channel Manager.

3.2.3.2 Channel Manager

Channel Manager acts as a controller, providing channels management and persistence. It's very similar to the Rule Manager module, and also allows the user to manage the channels that have been created or edited through a graphic web interface. So the user can, via this interface, obtain a list of channels available on the repository, and check their characteristics. This module has been implemented in PHP too, following the MVC pattern, in the same way of Rule Manager.

In the creation of a rule via Mobile App, Channel Manager plays a fundamental role,

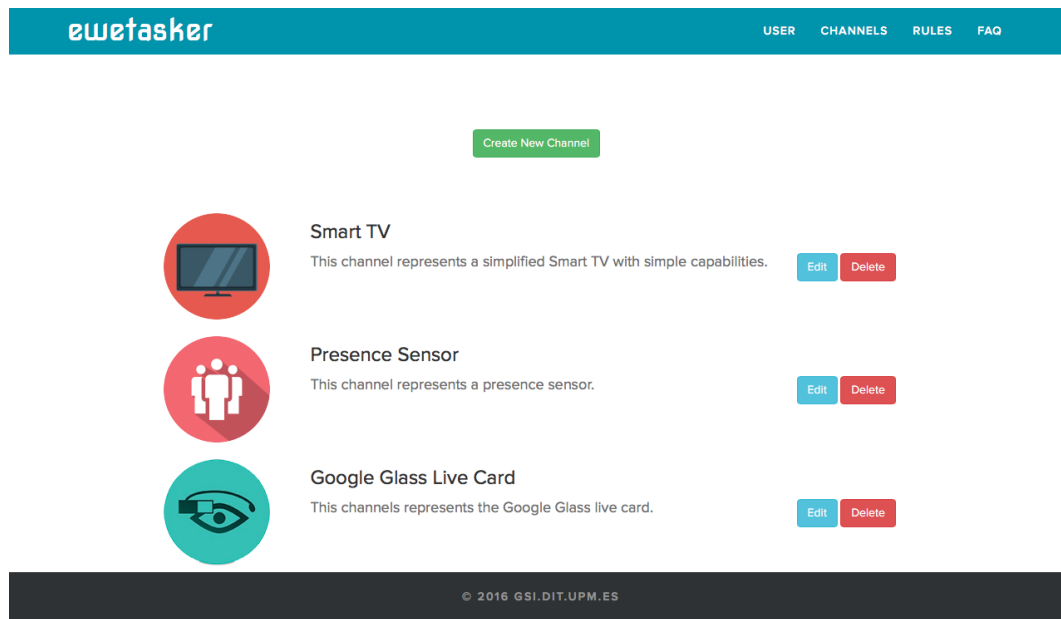


Figure 3.9: Channel Manager Interface

since it's the module which sends all channels, events and actions to the mobile, in JSON format. However, the creation of channels is limited to the web interface, and can't be carried out by the Mobile App module.

When a user is creating a rule, this module extracts channels from the Channel Repository, and pass them with all events and actions available to the Rule Administration.

3.2.3.3 Channel Repository

All the channels, with their respective events and actions, are stored in a MongoDB database. As explained in Rule Repository section, the Mongo database uses JSON format, and is organized into collections. This repository has three collections: channels, events and actions. Events and actions have their own collection separated from channels for making easier the dealing with them.

3.2.3.4 Events Manager

The Channel Administration module has one extra part that doesn't have its counterpart Rules Administration: the Events Manager sub-module. Events Manager is one of the most important parts of this project, because all the events and actions go through it.

This module has two main functions: capture events and send actions. Events from

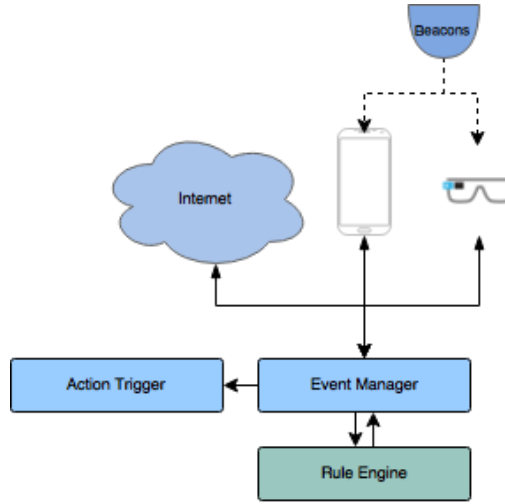


Figure 3.10: Events Manager Interconnection

Internet, Google Glass App and Mobile App are received by Events Manager, and then are sent to the Rule Engine. Once the Rule Engine has generated the consequential actions, they are received by this module, and after being parsed, each action is passed to all Action Trigger modules, and will be triggered depending on its nature. There are Action Trigger modules in the three main modules of this project (TAS, Mobile App and Google Glass App).

One of the EYE limitation is the format in which it generates the response. The EYE output is a string written in Notation3 [6] format that contains the prefixes used in the rule, the input and the response. Handle this output isn't easy, so it has to be parsed. The parsing of the response provided by the Rule Engine is carried out by the Events Manager. It's worthwhile to describe this process in detail, as it's one of the most important process:

1. First, it takes the response and the input, and removes from both the prefix included.
2. The next step is to delete all comments from the response and the input. For making easier this task, all the comments in rules must be between `#C` and `C#` tags.
3. Having removed prefixes and comments, the next fragments to erase are the break-lines. This is done with the use of regular expressions.
4. Now, both the response and the input are composed of triple patterns separated by a dot. The next step is to obtain the sentences that conform the response and the input, so both are split in sentences.
5. Once the sentences are separated, it's time to remove the input from the response, in order to have only the response sentences that corresponds with the actions generated.

6. The sentences of the response are the actions generated, but may not always be an action each sentence, because some of the actions have parameters that forms a new sentence. So, in this step, those parameters are found.
7. Finally, each sentence (that is a triple pattern that represents an action) is split into three fragments: subject, object and predicate. Subject and predicate correspond with the channel and the action respectively. Each parameter found before is added to its action.

After the parsing process, the response has been converted from a simple string to an array of arrays. Each array inside the parent array contains one of the three fragments of the triple pattern that represents the action and the parameters (if there are), so now we can handle each action properly. In order to make easy the deal with the actions in the Action Trigger module, this array of arrays is converted to a JSON with the following structure:

Listing 3.5: "Response"

```
{
  "success": "1",
  "actions": [{
    "channel": "Bluetooth",
    "action": "turnON",
    "parameter": ""
  },
  {
    "channel": "Notification",
    "action": "show",
    "parameter": "Battery level too low!"
  }]
}
```

This JSON is passed to all Action Trigger modules. Each Action Trigger module has a list of channels whose actions is able to trigger. Therefore, when the response arrives to it, it handles the JSON and triggers the actions which channel is contained in the list. For example, the action of “turn on Bluetooth” will be triggered by the Action Trigger module of the Mobile App and the Google Glass App.

In short, the Channel Administration module is responsible for providing channels creating, editing and removing functions; but has also one of the main roles of this project: the events manager. The events received are converted to actions and passed to the three

Action Trigger modules.

3.2.4 Action Trigger

Each main module has an Action Trigger sub-module, whose role is to trigger some of the actions generated by the Rule Engine. The Action Trigger has a channel list with all the channels whose actions is able to trigger. So, when the JSON arrives, this module finds those actions and runs them.

The Action Trigger of the TAS is able to run actions that come from several channels like: Twitter, Connected Door or Google Calendar. For this purpose, the Action Trigger module must be connected with those channels. So some examples of actions triggered by this module may be: open the door, post a tweet, send an email or switch on TV.

As shown, TAS is the core of this project. It's a very complex module with a variety of sub-modules that conforms it. However, the whole system is composed of another two modules.

3.3 Mobile App

The Mobile App module is divided into two main parts: the first for managing rules connecting with the Rules Administrator of the TAS, an a second part intended to work as an intermediary between beacons and the server. This project is focused on the second one.

Events are received by **Contextual Channel** and **Android Channel** sub-modules, depending on its nature. The former is connected with Estimote Beacons and receives events coming from presence and temperature channels; and the latter handle events coming from Wifi, Bluetooth and other Android channels. These events are sent to the TAS, that generates actions. These actions are sent to the Mobile App and triggered by the **Actions Trigger** sub-module. Finally, **Rule Administration** is a sub-module directly connected with the TAS, that provides rules managing functions.

This module is implemented in Java (the Android version) and in Objective-C (the iOS version), using the respectives SDKs.

3.3.1 Contextual Channel

The Contextual Channel sub-module is the core of the Mobile App module, and one of the most important parts in the whole system. Its main goal is to handle the data provided by beacons, and to send accordingly events to the server.

The mobile receives from each beacon via Bluetooth the ambient temperature and the distance. This sub-module is responsible for converting this data to Notation3 events, that may be evaluated by the Rule Engine. For this purpose, beacons are considered presence and temperature channels. Each beacon has an identifier, that is crucial in the event generation process.

All the beacons are associated with a certain place by their identifier. So the distance to the beacon can be processed as the distance to a certain place. For example, if the beacon with identifier “A1B2C3” is asociated to the GSI lab and the distance from the user to that beacon is 5 meters, can be said that the user is 5 meters away from the GSI lab. With this information it is possible to define a rule of the type “If distance to beacon A1B2C3 is less than 10 meters, then post a tweet”.

The event generation process is quite simple, thanks to input templates. Input templates are structures written in Notation3 that can be used as base to create events. An example of this templates for the generation of a presence detected event is the following:

Listing 3.6: "Input template"

```
ewe-presence:PresenceSensor rdf:type ewe-presence:
  PresenceDetectedAtDistance.
ewe-presence:PresenceSensor ewe:sensorID #SENSOR_ID#.
ewe-presence:PresenceSensor ewe:distance #DISTANCE#.
```

The Contextual Channel takes this template an replaces the params (`#SENSOR_ID#` and `#DISTANCE#`) with the data that has obtained from the beacon. Once the event has been created, it's passed to the TAS along with a query, where will be evaluated. The server responses with a JSON format result, that is handled by the Action Trigger sub-module.

However, the user could want that only some rules were evaluated. For this reason, the user is able to configure which rules wants to be considered by the Rule Engine, and here is where come into play the query mentioned above. This selection can be done by a certain place or a certain user creator. And the user can also block some rules by title. In addition, it's possible to configure the Mobile App to select the rules that are evaluated

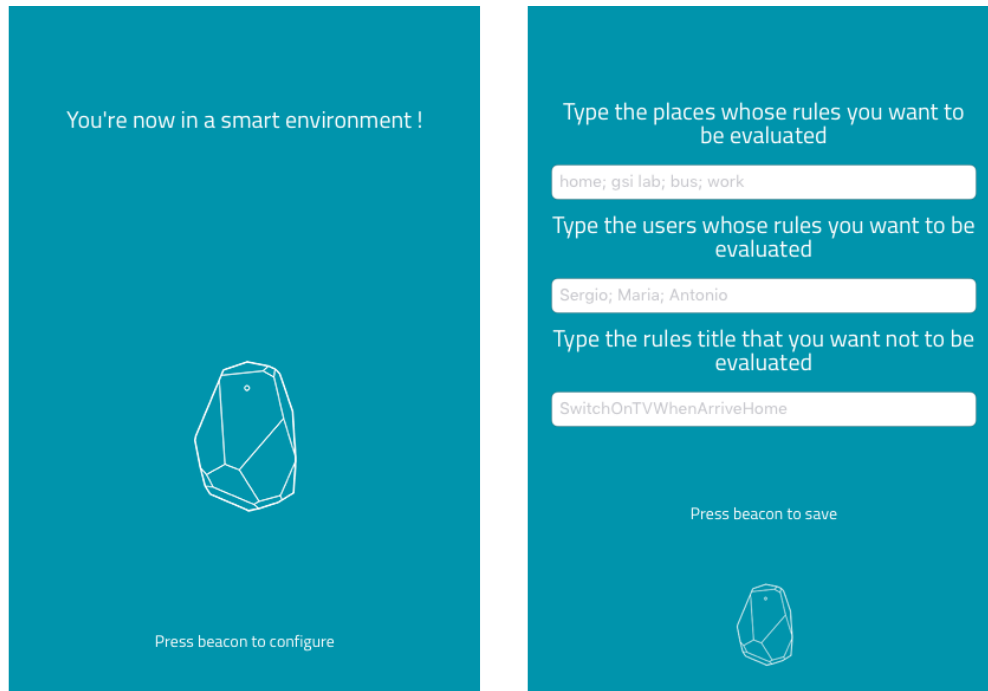


Figure 3.11: Mobile App Graphic Interface

depending on the place where the user is. For example, if the mobile app is receiving data from the beacon with id “A1B2C3”, that is asociated to the GSI Lab, it makes no sense to evaluate rules that are assigned to the place Home. So this module serves the user a graphic interface to configure this parameters, and will generate a query that will be send with each event. The Channel Manager, sub-module of TAS explained above, will handle this query to obtain the wanted rules and to send them to the Rule Engine.

3.3.2 Device Channel

This module is very similar to its counterpart, Contextual Channel, with the difference that handles the events coming from the own device [11]. Some examples of this channels are Bluetooth, Wifi, Battery or Screen.

It works in the same manner as the Contextual Channel, receiving events from this channels and converting them to Notation3 format. It also uses input templates for this purpose. Once the event is converted to Notation3 and the input has been generated, it’s passed to the TAS to be evaluated. Some examples of events captured by this module are: wifi is on, battery level is low, data network is off.

In addition to handle events, Mobile App module also allows the user to manage rules.

These functions are provided by the Rule Administration module.

3.3.3 Rule Administration

This module works in a similar way to the Rule Editor sub-module of the TAS. It provides functions for managing rules, such as creating, editing and removing.

It's connected with the Channel Manager sub-module of the TAS, which provides a channel list that is used by Rule Administration to get the available channels with their events and actions. This list is passed in JSON format. Once the available channels are received, the rules can be defined via the graphic interface that provides this module. When the rule is defined, is sent to the TAS via POST request, where will be stored.

3.3.4 Action Trigger

This sub-module has the same goal that its counterpart in TAS: to trigger actions. It has a list with all the channels whose actions can trigger. So when the module receives the JSON with all the actions from the server, it looks for those whose channel is contained within its list.

Once the actions that the module is able to trigger have been selected, it triggers it. Some examples of actions that may be triggered by this module are: show notification, mute the mobile or call someone.

In this chapter the features and modules that are involved in this project have explained, along with their connection and relationship.

To sum up, this project counts with a TAS module that provides rules and channels management functions, evaluate events and rules and trigger actions; and a Mobile App module that connects with beacons and other devices channels and generates events that are sent to the TAS. This module is also able to trigger some actions. In the following chapter, the case study will be explained in detail.

Case study

This chapter aims to aid in the understanding of the project functionalities. For this purpose, the main use case is described, covering the main features of the system.

The main actor of this case is the user, whose goal is to automate a task. As will be explained, this is a sophisticated process where several modules are involved.

4.1 Login and events configuration

Before being able to create channels and rules, the user has to sign into the system. For this purpose, he may navigate to User page, where a login form will be displayed. Furthermore, in order to enable the platform to receive Internet events, it's necessary to connect with some channels. This can be done in the same page once the user has logged in, and only has to press the corresponding button. The button redirects to the corresponding official page of the channel where will be shown a form to allow our system to connect with it.

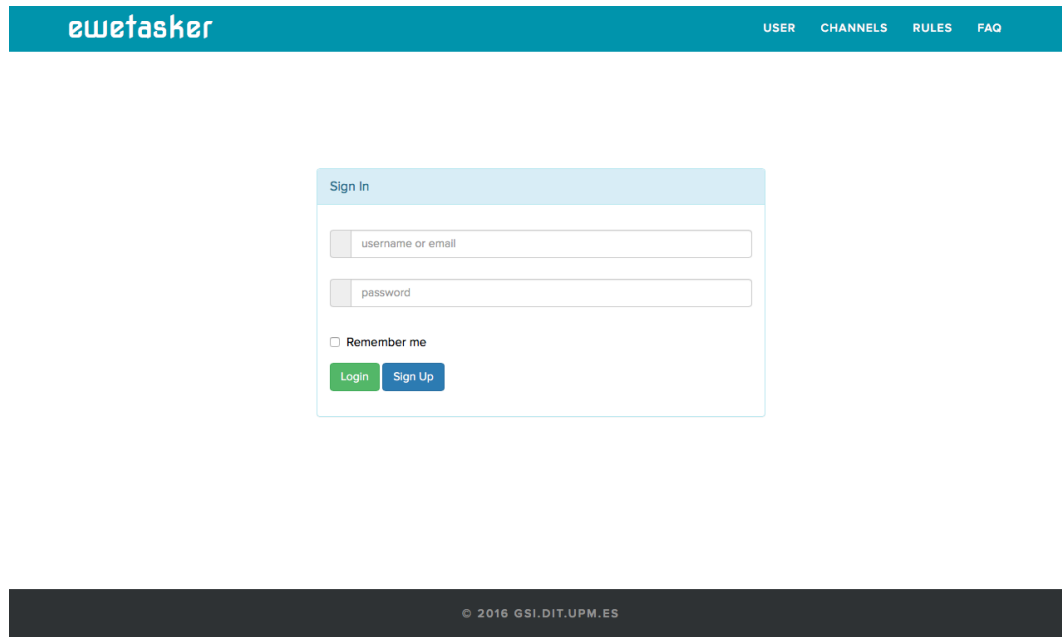
The image shows the login interface of the 'ewetasker' application. At the top, there is a teal header bar with the 'ewetasker' logo on the left and navigation links 'USER', 'CHANNELS', 'RULES', and 'FAQ' on the right. Below the header, a light blue box contains the 'Sign In' form. The form has two input fields: 'username or email' and 'password'. Below these fields is a checkbox labeled 'Remember me'. At the bottom of the form are two buttons: a green 'Login' button and a blue 'Sign Up' button. At the very bottom of the page, a dark grey footer bar contains the copyright text '© 2016 GSI.DIT.UPM.ES'.

Figure 4.1: Login form

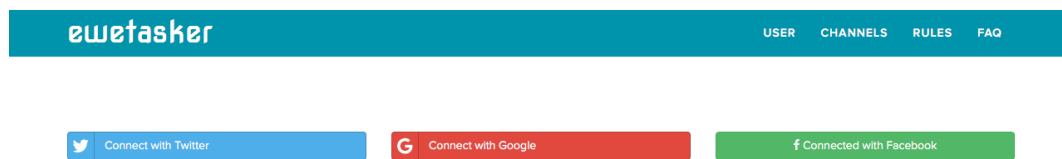


Figure 4.2: Internet channels connection

4.2 Channel Edition

The first step to automate a task, after the user has logged in and the events have been configured, is to create a rule. As explained, a rule is composed of an event and an action, that come from channels. So the user must navigate to the Channels page to check if the required channels are available. This page shows a list with all the channels that have been created by him, and also those which have been created by other users. The user can edit a created channel or create a new one. In this case, the user must press the “Create new Channel” button, and will be redirected to the Channel Editor interface.

In the Channel Editor interface, the user is able to create a new channel by filling some fields. Those fields are divided into the groups. The first group represents channel attributes such as title, description, nickname and image. The other two groups represent events and actions respectively. The fields to fill for each event or action are title, rule and prefix. User

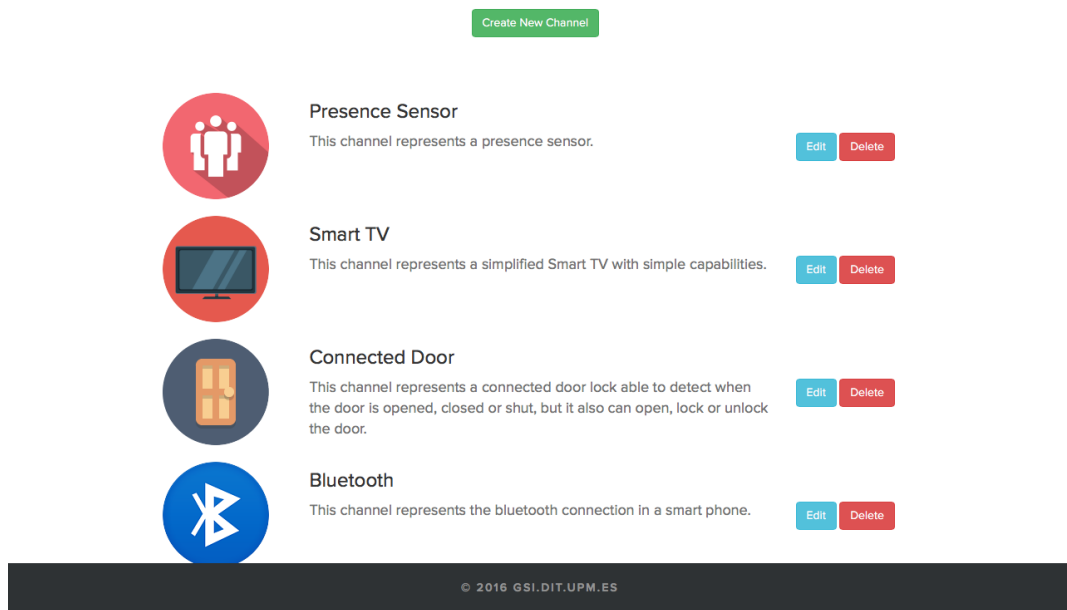


Figure 4.3: Channels page

can add as much events and actions as he wants, by pressing the “Add event” or “Add action” button. In this case, the event or action fields group will be duplicated.

The screenshot shows the "ewetasker" Channels Editor interface. The top navigation bar includes "USER", "CHANNELS", "RULES", and "FAQ". The main form is titled "New Channel" and contains the following fields:

- Title**: Input field with the value "door".
- Description**: Text area with the value "This channel represents a connected door lock able to detect when the door is opened, closed or shut, but it also can open, lock or unlock the door".
- Nicename**: Input field with the value "Connected Door".
- Image**: A button labeled "Seleccionar archivo" and the text "nada seleccionado".
- Event**: A section containing three input fields:
 - Title**: Input field with the value "New tweet".
 - Rule**: Text area with the value "?a <knows ?b.?a:age math:lessThan #PARAM_1#".
 - Prefix**: Text area with the value "@prefix : <ppl#>. @prefix math: <http://www.w3.org/2000/10/swap/math#>.".
- Action**: A section containing one input field:
 - Title**: Input field with the value "Turn on".

At the bottom right of the form, there are two green buttons labeled "Add event" and "Add action". At the bottom of the page, there is a dark grey footer with the text "© 2016 GSI.DIT.UPM.ES".

Figure 4.4: Channels Editor interface

Once all the fields have been filled (if the user doesn't want to add events or actions, he must leave empty those fields), the channel with its events and actions will be stored by pressing the "Send" button. The user may create all the needed channels to its rules following this process (except those which are already available). When all the needed channels are available, the user may navigate to the Rules page.

It's important to mention that, for security purposes, only Admin User is able to edit and remove channels, and he is responsible for managing them.

4.3 Rule Edition

Rules page shows a list containing all rules that have been created. Each rule item shows relevant information such as the channels involved (represented by their image), the rule description, the creator, the place where the rule works, and the date when was created. Rules can be filtered by place. For creating a new rule, the user may press the "Create new rule" button, and will be redirected to the Rule Editor interface.

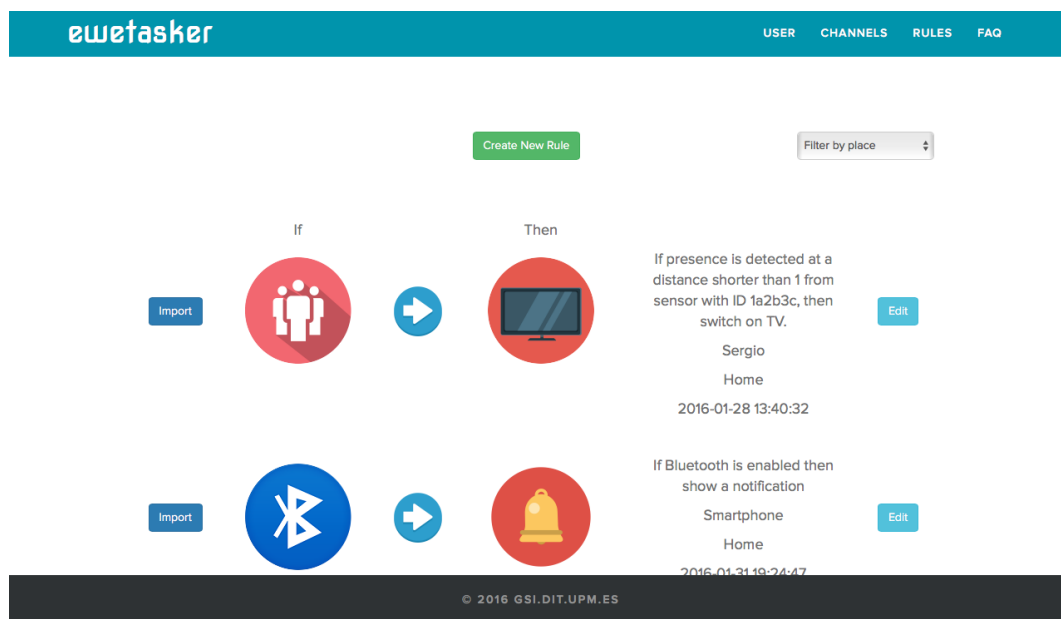


Figure 4.5: Rules page

This interface has been designed to be easy to use and friendly to the user. The user selects the channel that will represent the event. This selection is made by dragging the icon representing the corresponding channel and dropping it into the left container. After this, a modal view appears containing all the events available for this channel. The user must select the wanted event and press the "OK" button.

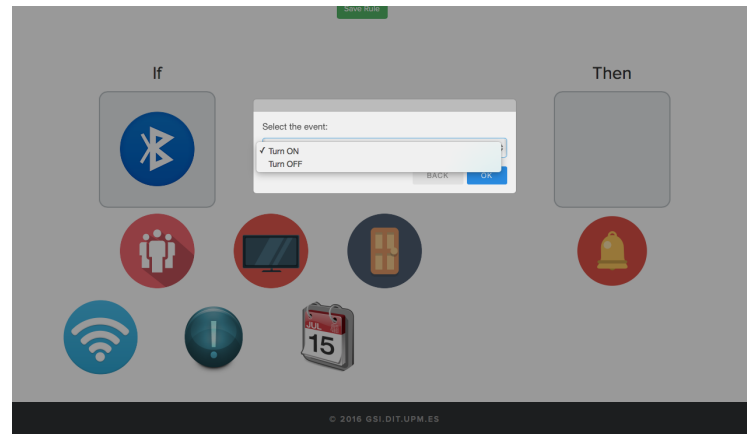


Figure 4.6: Event Selection

Having configured the event of the rule, the user proceeds with the action. The action selection is very similar to the event selection, also made by dragging the icon representing the corresponding channel and dropping it into the right container. In the same way that the event, a modal view appears. In this case, it offers the actions available for the selected channel. The user selects the action which he wants to be triggered and press “OK”. Once these configurations have been made, the rule can be saved by pressing the “Save Rule” button.

Now, a new modal view appears. This modal view contains several fields that must be filled. These fields represents rules attributes such as title, description, creator, place and all the params to configure for the events and actions (if any).

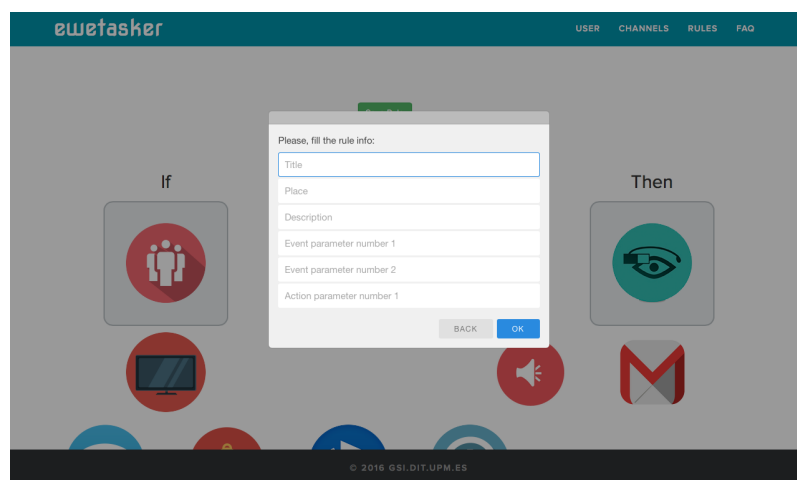


Figure 4.7: Save Rule

Once these fields have been filled, the user may press “OK” and the rule will be stored.

4.4 Rule Import

Rules are stored in the rule repository when they are created. However, the user has to import to his own rule repository for being evaluated when an event is received. This can be done from the Rules page. Each rule has a button called “Import”. Once the user presses that button, the rule is imported to his own rule repository, and will be evaluated when an event received.

All in all, we have presented the steps a user has to follow to automate a task. As an example, a real case study is presented in the following section.

4.5 Case Study

In this case study, we are going to evaluate how the system previously described has been evaluated for automating a Smart House. The project has been evaluated in the laboratory of the Intelligent Systems Group (GSI). In this laboratory, the following equipment has been used:

- Proximity sensors: 3 estimote beacons installed as follows:
 - Purple beacon with ID “G7H8I9”: main door.
 - Green beacon with ID “D4E5F6”: workroom.
 - Blue beacon with ID “A1B2C3”: rest room.

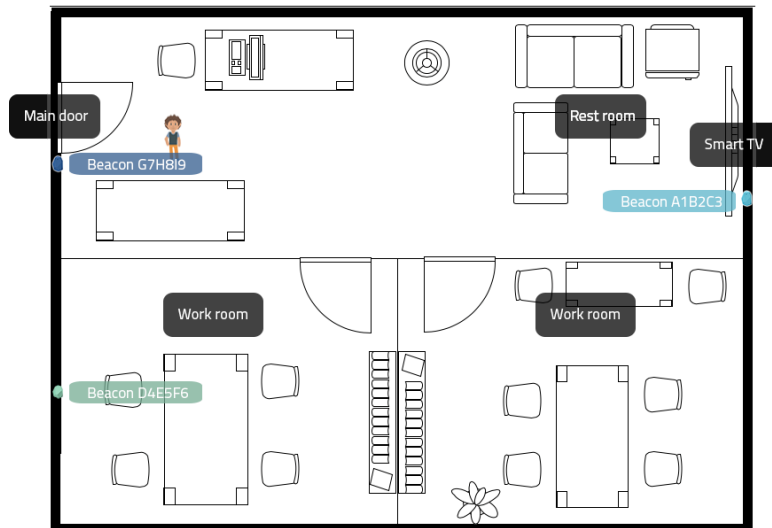


Figure 4.8: Beacons placing

- Internet channels: Twitter.
- Devices: Connected Door, Smart TV.
- Mobile Phone and/or GGlass.

For this example, the following conditions are assumed:

- The user is already logged in the system.
- The channels PresenceSensor, Twitter, Smart TV, Connected Door, and Notification are already created with params explained in appendix A.
- Rules aren't already created.
- The user brings his Mobile Phone or GGlass with Bluetooth activated.
- Beacon with id "A1B2C3" (blue beacon) assigned to the lab rest room, beacon with id "D4E5F6" (green beacon) assigned to the lab workroom, and beacon with id "G7H8I9" (purple beacon) assigned to the lab main door.

With these conditions, several tasks such as opening the main door when the user is near and posting a tweet when he goes into lab, switching on TV when the user is in the rest room, and showing a notification when he goes into the work room could be automated. The steps to realize this automation process are the following:

1. Connect with Twitter pressing the corresponding button in User page.
2. Create several rules from Rule Editor graphic interface with params described in appendix A.
3. Import the rules pressing "Import" button in Rules page.

Once these actions are done, the task automation process has been completed. When the user is next to the main door, it will be opened. Everytime the user goes into the workroom, a tweet with the message "I am working" will be posted. And when the user is in the rest room, the smart TV will be switched on and a notification with the message "Take a coffee" will be shown on user's device (Mobile or GGlass).

In addition, to help the user understand and become familiar with rules and Notation3, two tools have been added:

- EYE Client [12], that can be accessed pressing the “TEST EYE” button on the main page. With this tool the user is able to write rules and inputs in Notation3 format and see the results.

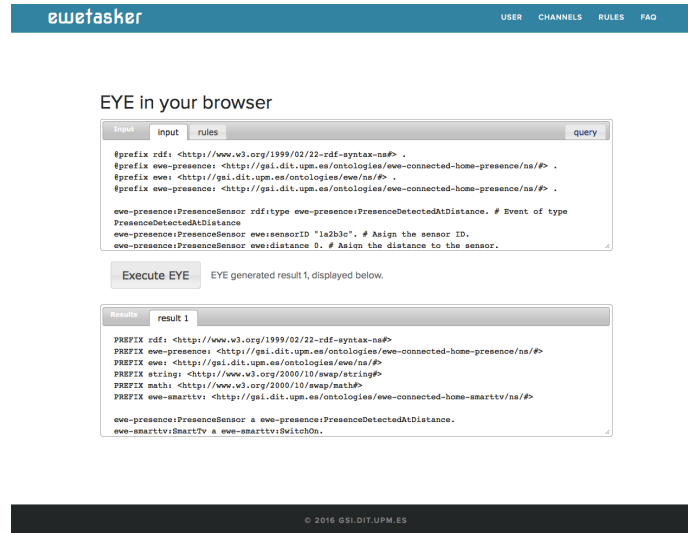


Figure 4.9: Eye Client Tool

- A Simulator where the user can experiment how rules would be executed in a smart environment. The user can place three beacons with different ids in a map of the GSI laboratory, and move its avatar around the three rooms. This simulator can be used for testing rules that involve events coming from the Presence Sensor channel. When a rule is triggered, a modal view is shown.

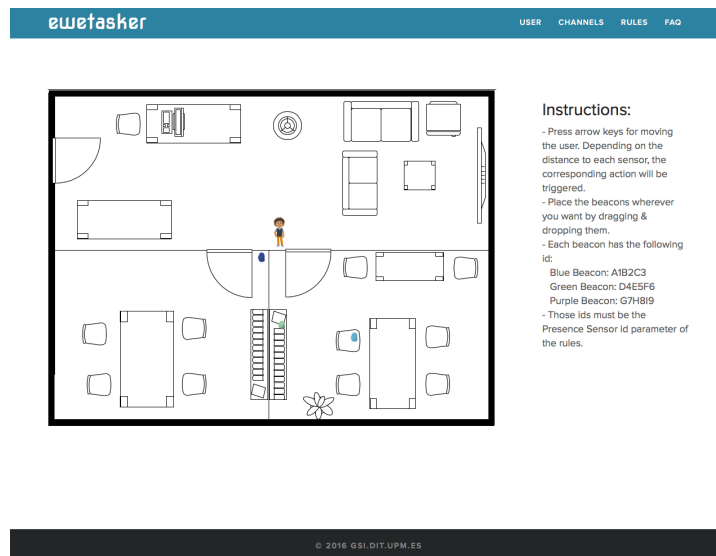


Figure 4.10: Simulator Tool

Conclusions and future work

In this chapter we describe the conclusions extracted from this project, problems encountered, achievements and thoughts about future work.

5.1 Conclusions

In this project, a task automation platform for beacon enabled smarthomes has been developed, to make the task automation process easier. In addition, the project allows the user to manage rules and channels, and store them in a rules repository. Users can also import in their own repository rules created by other users. With this, users are able to automate tasks in a smart environment. In addition, two tools for simulating and testing have been included, with the purpose of helping the user to become familiar with Notation3.

The system uses several different technologies, thus we have learnt these varied technologies, and how to coordinate all of them. We have used web technologies: PHP, Javascript, HTML5; database technologies: MongoDB; semantic technologies: Notation3, EYE, RDF; and mobile development technologies: Java, Objective-C.

The software architecture developed in this project is used in other two projects: Design and implementation of a Semantic Task Automation Rule Framework for Android

Devices [11] and Development of a Beacon enabled content delivery System for Alzheimer's patients equipped with Google Glass [13].

This project can be used in countless applications, such as helping Alzheimer's patients (developed in the project mentioned above); sending and receiving customized info in museums or shopping centers; automating tasks in a smart home for increasing user's convenience; or helping disabled people in daily tasks.

In the following sections, the achieved goals will be described in depth, as the problems faced and some suggestions for a future work.

5.2 Achieved goals

In this section, the achieved features and goals in this project will be described.

Develop a task automation platform for beacon enabled smarthomes This was the main goal of the project, the development of a system that allows the user to automate tasks. The process involves several modules and functions, from a Task Automation Service to a Mobile App.

Create a graphical interface that allows users to manage channels A graphical interface to create and edit channels has been developed in this project. By filling some fields, the user is able to create and edit channels.

Create a graphical interface that allows users to manage rules In this project a graphical interface that allows the user to create or edit automation rules in an easy way has been included. This process is based on a user-friendly "drag and drop" technique.

Persist and manager rules and channels Users need to store and manage their channels and rules, so this was a mandatory requirement.

Create EYE rules In order to evaluate rules with EYE rule engine, they have to be written in a compatible format. This format is Notation3.

Handle EYE response To allow Action Triggers to execute actions, the EYE response has to be parsed and converted to JSON format, to make easier its handle.

Run a semantic motor The created EWE rules are loaded into Rule Engine, that is based on EYE motor. This allows users to run actions that are triggered by events according to these rules. Events are generated by different contextual, device and Internet channels, while actions are triggered by Action Triggers modules.

Connect with Internet channels To receive events from Internet sources and to be able to run actions in Internet channels, it's needed to connect with these channels such as Twitter, Google and Facebook.

Connect with Beacons The system is able to receive contextual events coming from Estimote Beacons, and to generate accordingly actions. In order to connect with beacons and handle this events, the development of a mobile application has been necessary. These sensors are used as Presence Sensor and Temperature Sensor channels.

Connect with Google Glass App The system developed in this project is able to connect with a Google Glass App to receive events and to trigger actions on it.

5.3 Problems faced

During the development of this project was needed to face some problems, that are listed below:

- **EYE motor:** The rule engine used in this project is still under development, the documentation about it is very small and we have encountered some limitations.
- **Rules storage:** At first, rules were going to be stored using Linked Data Fragments [14]. However, EYE rules are written in Notation3, that adds formulas and quantification to RDF¹. LDF servers don't support formulas nor quantification. After this, we tried it using Levelgraph ² but we had the same problem, so we decided to use MongoDB.
- **EYE response:** EYE engine generates as response a raw text in Notation3 format. We have tried to parse this response with some libraries, but we had the same problem that we had storing rules: formulas and quantification not supported. So we had to develop an EYE response parser to deal with this.
- **Beacons accuracy:** The precision of distance measures from mobile to beacon is sometimes quite accurate. We have faced this problem taking average values for the generation of events.

¹<https://www.w3.org/RDF/>

²<https://github.com/mcollina/levelgraph-n3>

5.4 Future work

In this section, possible new features or improvements that could be done to the project will be explained.

Triggers integration This project could be integrated with some triggers and sensors in order to have a more sophisticated smart environment.

Private rules and channels Enable or disable channels and rules depending on the user permissions. Now, every user is allowed to use all the channels and rules available.

Internet channels integration In the current version, only Twitter, Google Calendar and Facebook can work as channels in the system. In the future, would be interesting to add more channels.

Software engineering tasks integration Include channels such as Github, Bitbucket or Trello to automate tasks related to software engineering. For example, to post a tweet when a new release is delivered, or when a Trello task is done.

Machine learning integration Apply machine learning techniques for suggesting new rules to users; based on their locations, rules popularity, most used rules, or personal interest.

Bibliography

- [1] Aislelabs, “The hitchhikers guide to ibeacon hardware: A comprehensive report,” <http://www.aislelabs.com/reports/beacon-guide/>, 2015.
- [2] Google, “Eddystone,” <https://github.com/google/eddystone>, 2015.
- [3] —, “Proximity beacon api,” <https://developers.google.com/beacons/proximity/guides>, 2015.
- [4] T. Rattanasawad, K. Saikaew, M. Buranarach, and T. Supnithi, “A review and comparison of rule languages and rule-based inference engines for the semantic web,” in *Computer Science and Engineering Conference (ICSEC), 2013 International*, Sept 2013, pp. 1–6.
- [5] M. Coronado, C. A. Iglesias, and E. Serrano, “Modelling rules for automating the Evented WEB by semantic technologies,” *Expert Systems with Applications*, vol. 42, no. 21, pp. 7979 – 7990, 2015. [Online]. Available: <http://www.sciencedirect.com/science/article/pii/S0957417415004339>
- [6] B. De Meester, D. Arndt, P. Bonte, J. Bhatti, W. Dereuddre, R. Verborgh, F. Ongenae, F. De Turck, E. Mannens, and R. Van de Walle, “Event-driven rule-based reasoning using EYE,” in *Joint Proceedings of the 1st Joint International Workshop on Semantic Sensor Networks and Terra Cognita and the 4th International Workshop on Ordering and Reasoning*, Oct. 2015. [Online]. Available: <http://ceur-ws.org/Vol-1488/paper-08.pdf>
- [7] J.DeRoo, “Euler yet another proof engine,” <http://eulerssharp.sourceforge.net>, 2013.
- [8] M. Coronado and C. A. Iglesias, “Task Automation Services: Automation for the masses,” *Internet Computing, IEEE*, vol. PP, no. 99, pp. 1–1, 2015.
- [9] O. Araque, “Design and implementation of an event rules web editor,” ETSI Telecomunicación, July 2014.
- [10] R. Verborgh and J. D. Roo, “Drawing conclusions from linked data on the web: The EYE reasoner,” *IEEE Software*, vol. 32, no. 3, pp. 23–27, 2015. [Online]. Available: <http://dx.doi.org/10.1109/MS.2015.63>
- [11] A. F. Llamas, “Design and implementation of a semantic task automation rule framework for android devices,” 2016.
- [12] B. De Meester, D. Arndt, P. Bonte, J. Bhatti, W. Dereuddre, R. Verborgh, F. Ongenae, F. De Turck, E. Mannens, and R. Van de Walle, “Event-driven rule-based reasoning using EYE,” in *Joint Proceedings of the 1st Joint International Workshop on Semantic Sensor*

- Networks and Terra Cognita and the 4th International Workshop on Ordering and Reasoning*, Oct. 2015. [Online]. Available: <http://ceur-ws.org/Vol-1488/paper-08.pdf>
- [13] D. P. Caro, “Development of a beacon enabled content delivery system for alzheimer’s patients equipped with google glass,” 2016.
- [14] R. Verborgh, M. Vander Sande, P. Colpaert, S. Coppens, E. Mannens, and R. Van de Walle, “Web-scale querying through Linked Data Fragments,” in *Proceedings of the 7th Workshop on Linked Data on the Web*, Apr. 2014. [Online]. Available: http://events.linkedata.org/ldow2014/papers/ldow2014_paper_04.pdf

Rules and channels creation

This appendix go through the process of creating channels and rules for the use case explained in section 4.5.

A.1 Channels definition

- The channel PresenceSensor is created with the following params:
 - **Title:** presencesensor.
 - **Description:** This channel represents a presence sensor.
 - **Nicename:** Presence Sensor.
 - **Event:**
 - * Title: PresenceDetectedAtDistance.
 - * Rule:

```
?event rdf:type ewe-presence:PresenceDetectedAtDistance.  
?event ewe:sensorID ?sensorID.  
?sensorID string:equalIgnoringCase '#PARAM_1#'.  
?event!ewe:distance math:lessThan #PARAM_2#.
```

* Prefix:

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
.
@prefix string: <http://www.w3.org/2000/10/swap/string#>.
@prefix math: <http://www.w3.org/2000/10/swap/math#>.
@prefix ewe: <http://gsi.dit.upm.es/ontologies/ewe/ns/#> .
@prefix ewe-presence: <http://gsi.dit.upm.es/ontologies/
ewe-connected-home-presence/ns/#> .
```

- The channel Twitter is created with the following params:

- **Title:** twitter.
- **Description:** This channel represents Twitter social network.
- **Nicename:** Twitter.
- **Action:**
 - * Title: PostTweet.
 - * Rule:

```
ewe-twitter:Twitter rdf:type ewe-twitter:Tweet ;
                    ov:MicroBlogPost "#PARAM_1#" .
```

* Prefix:

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
.
@prefix ewe-twitter: <http://gsi.dit.upm.es/ontologies/ewe-
twitter/ns/#> .
@prefix ov: <http://vocab.org/open/#>.
```

- The channel Smart TV is created with the following params:

- **Title:** smarttv.
- **Description:** This channel represents a simplified Smart TV with simple capabilities.
- **Nicename:** Smart TV.
- **Action:**
 - * Title: Switch on.
 - * Rule:

```
ewe-smarttv:SmartTv rdf:type ewe-smarttv:SwitchOn.
```

* Prefix:

```
@prefix ewe-smarttv: <http://gsi.dit.upm.es/ontologies/ewe-
-connected-home-smarttv/ns/#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
.
```

- The channel Connected Door is created with the following params:

- **Title:** door.
- **Description:** This channel represents a connected door lock able to detect when the door is opened, closed or shut, but it also can open, lock or unlock the door.
- **Nicename:** Connected Door.
- **Action:**

* Title: Open Door.

* Rule:

```
ewe-door:DoorLock rdf:type ewe-door:OpenDoor .
```

* Prefix:

```
@prefix ewe-door: <http://gsi.dit.upm.es/ontologies/ewe-
-connected-home-door/ns/#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
.
```

- The channel Notification is created with the following params:

- **Title:** Notification.
- **Description:** This channel represents a smartphone notification.
- **Nicename:** Notification.
- **Action:**

* Title: Show Notification.

* Rule:

```
ewe-notification:Notification rdf:type ewe-notification:
Show;
                                ov:message "#PARAM_1#";
```

* Prefix:

```
@prefix ewe-notification: <http://gsi.dit.upm.es/
    ontologies/ewe/ns/ewe-notification/ns/#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
.
@prefix ov: <http://vocab.org/open/#>.
```

A.2 Rules definition

- “If I’m near the door, then open it”:
 - **Event channel:** Presence Sensor.
 - **Event:** PresenceDetectedAtDistance.
 - **Action channel:** Connected Door.
 - **Action:** Open Door.
 - **Title:** Open the door when I’m near.
 - **Description:** If I’m near the main door, then open it.
 - **Event param 1:** G7H8I9.
 - **Event param 2:** 1.

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix string: <http://www.w3.org/2000/10/swap/string#>.
@prefix math: <http://www.w3.org/2000/10/swap/math#>.
@prefix ewe: <http://gsi.dit.upm.es/ontologies/ewe/ns/#> .
@prefix ewe-presence: <http://gsi.dit.upm.es/ontologies/ewe-
    connected-home-presence/ns/#> .
@prefix ewe-door: <http://gsi.dit.upm.es/ontologies/ewe-connected-
    home-door/ns/#> .

{
    ?event rdf:type ewe-presence:PresenceDetectedAtDistance.
    ?event ewe:sensorID ?sensorID.
    ?sensorID string:equalIgnoringCase 'G7H8I9'.
    ?event!ewe:distance math:lessThan 1.
}
=>
{
    ewe-door:DoorLock rdf:type ewe-door:OpenDoor .
}
```

- “If I’m in the rest room, then switch on TV”:
 - **Event channel:** Presence Sensor.
 - **Event:** PresenceDetectedAtDistance.
 - **Action channel:** Smart TV.
 - **Action:** Switch ON.
 - **Title:** Switch on TV when I’m in the rest room.
 - **Description:** If I’m in the rest room, then switch on TV.
 - **Event param 1:** A1B2C3.
 - **Event param 2:** 3.

```

@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix string: <http://www.w3.org/2000/10/swap/string#>.
@prefix math: <http://www.w3.org/2000/10/swap/math#>.
@prefix ewe: <http://gsi.dit.upm.es/ontologies/ewe/ns/#> .
@prefix ewe-presence: <http://gsi.dit.upm.es/ontologies/ewe-
connected-home-presence/ns/#> .
@prefix ewe-smarttv: <http://gsi.dit.upm.es/ontologies/ewe-
connected-home-smarttv/ns/#> .

{
  ?event rdf:type ewe-presence:PresenceDetectedAtDistance.
  ?event ewe:sensorID ?sensorID.
  ?sensorID string:equalIgnoringCase 'A1B2C3'.
  ?event!ewe:distance math:lessThan 3.
}
=>
{
  ewe-smarttv:SmartTv rdf:type ewe-smarttv:SwitchOn.
}

```

- “If I’m in the rest room, then show a notification on my device with the message ’Take a coffee!’”:
 - **Event channel:** Presence Sensor.
 - **Event:** PresenceDetectedAtDistance.
 - **Action channel:** Notification.
 - **Action:** Show.
 - **Title:** Show notification when I’m in the rest room.

- **Description:** If I'm in the rest room, then show a notification.
- **Event param 1:** A1B2C3.
- **Event param 2:** 3.
- **Action param 1:** Take a coffee!.

```
@prefix ewe-notification: <http://gsi.dit.upm.es/ontologies/ewe/ns/ewe-
-notification/ns/#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix ov: <http://vocab.org/open/#>. @prefix string: <http://www.w3.
org/2000/10/swap/string#>.
@prefix math: <http://www.w3.org/2000/10/swap/math#>.
@prefix ewe: <http://gsi.dit.upm.es/ontologies/ewe/ns/#> .
@prefix ewe-presence: <http://gsi.dit.upm.es/ontologies/ewe-connected-
home-presence/ns/#> .

{
  ?event rdf:type ewe-presence:PresenceDetectedAtDistance.
  ?event ewe:sensorID ?sensorID.
  ?sensorID string:equalIgnoringCase 'A1B2C3'.
  ?event!ewe:distance math:lessThan 3.
}
=>
{
  ewe-notification:Notification rdf:type ewe-notification:Show;
                                ov:message "Take a coffee!";
}
```

- “If I'm in the workroom, then post a tweet with the message 'I am working.'”:
- **Event channel:** Presence Sensor.
- **Event:** PresenceDetectedAtDistance.
- **Action channel:** Twitter.
- **Action:** Post tweet.
- **Title:** Post tweet when I arrive work.
- **Description:** If I arrive work, then post a tweet with the message 'I am working'.
- **Event param 1:** D4E5F6.
- **Event param 2:** 4.
- **Action param 1:** I am working.

```
@prefix ewe-twitter: <http://gsi.dit.upm.es/ontologies/ewe-twitter/
ns/#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix ov: <http://vocab.org/open/#>. @prefix string: <http://www.w3
.org/2000/10/swap/string#>.
@prefix math: <http://www.w3.org/2000/10/swap/math#>.
@prefix ewe: <http://gsi.dit.upm.es/ontologies/ewe/ns/#> .
@prefix ewe-presence: <http://gsi.dit.upm.es/ontologies/ewe-
connected-home-presence/ns/#> .

{
  ?event rdf:type ewe-presence:PresenceDetectedAtDistance.
  ?event ewe:sensorID ?sensorID.
  ?sensorID string:equalIgnoringCase 'D4E5F6' .
  ?event!ewe:distance math:lessThan 4.
}
=>
{
  ewe-twitter:Twitter rdf:type ewe-twitter:Tweet ;
                      ov:MicroBlogPost "I am working" .
}
```

