UNIVERSIDAD POLITÉCNICA DE MADRID

ESCUELA TÉCNICA SUPERIOR
DE INGENIEROS DE TELECOMUNICACIÓN


GRADO EN INGENIERÍA DE TECNOLOGÍAS Y SERVICIOS DE TELECOMUNICACIÓN

TRABAJO FIN DE GRADO

# DEVELOPMENT OF A TASK AUTOMATION PLATFORM BASED ON SEMANTIC MULTIEVENT-DRIVEN RULES

CARLOS MORO GARCÍA

2017

## TRABAJO FIN DE GRADO

| | |
|---|---|
| **Título:** | Desarrollo de una Plataforma de Automatización de Tareas basada en Reglas Semánticas Multi-evento |
| **Título (inglés):** | Development of a Task Automation Platform based on Semantic Multievent-driven rules |
| **Autor:** | Carlos Moro García |
| **Tutor:** | Sergio Muñoz López |
| **Departamento:** | Ingeniería de Sistemas Telemáticos |

## MIEMBROS DEL TRIBUNAL CALIFICADOR

**Presidente:**

**Vocal:**

**Secretario:**

**Suplente:**

## 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 BASED ON SEMANTIC MULTIEVENT-DRIVEN RULES

**Carlos Moro García**

Junio de 2017

# Resumen

En los últimos años se ha vivido un gran auge de la Internet de las Cosas. Esto abre muchas posibilidades en el desarrollo de nuevas aplicaciones y servicios. La Internet de las Cosas se refiere a la red de dispositivos provistos de inteligencia ubicua que son capaces de interconectarse entre ellos, permitiendo la monitorización, el control y la automatización de actividades diarias.

El proyecto define una plataforma de automatización basada en reglas ECA (Evento-Condición-Acción) que permite el soporte de reglas multi-evento y multi-acción. Las reglas se definen usando la ontología EWE que a su vez está basada en tecnologías semánticas. El objetivo principal es crear la plataforma para automatizar esas reglas. Para ello se han añadidos nuevas características a un servidor de automatización de tareas y se han desarrollado nuevos módulos: un servidor de contexto asociado a un lugar inteligente y un bot de Telegram.

El servidor se compone de varios submódulos que permiten gestionar las reglas multi-evento, un motor que evalúa las reglas multi-evento y multi-acción, y un módulo que ejecuta las acciones.

El bot de Telegram es una conversación en la aplicación donde el usuario puede gestionar las reglas que tiene importadas, borrarlas e importar reglas nuevas.

El servidor de contexto hace de intermediario entre la plataforma y los distintos lugares inteligentes para ejecutar algunas reglas que la plataforma no puede ejecutar por no estar en la misma subred que los actuadores.

Se han creado dos nuevos módulos que sirven como generadores de eventos y como actuadores. Estos módulos son Telegram y Chromecast.

Por último, se presentan las conclusiones extraídas tras la realización de este proyecto, los obstáculos y los posibles aspectos a mejorar en un futuro.

**Palabras clave:** Tecnologías semánticas, RDF, EWE, PHP, Telegram, EYE, MongoDB, Chromecast, Automatización

# Abstract

In the last years, there has been a peak of the Internet of the Things. It opens many possibilities for the development of new applications and services. IoT is the net of devices that are provided with ubiquitous intelligence and are capable of interconnecting among them. It enables monitoring, controlling and automating some daily activities.

The project defines an automation platform based on ECA (Event-Condition-Action) rules and enables the support of multi-event and multi-action rules. The rules can be defined using EWE ontology which is based on semantic technologies. The main aim is to create the platform that will automate the rules. For this purpose, it has been added new characteristics to a Task Automation Server and new modules has been developed: a Context Server and a Telegram Bot.

Task Automation Server is composed of various submodules that allow handling the multi-event rules, an engine that evaluates the multi-event and multi-action rules, and a module that triggers the actions.

Telegram Bot is a chat where the user can manage the imported rules, remove them and import new rules.

The Context Server is the intermediary between the Task Automation Server and some channels. The TAS can't trigger some actions if the channels aren't in the same subnet.

Two new channels have been created. They work as receivers and as performers. These channels are Telegram and Chromecast.

Finally, it is presented the conclusions drawn from this project, the obstacles that have been found during the development and possible future lines of work.

**Keywords:** Semantic Technologies, RDF, EWE, PHP, Telegram, EYE, MongoDB, Chromecast, Automation

# Agradecimientos

Gracias a mis padres por apoyarme en todo momento, ayudarme a tomar las decisiones adecuadas y aceptar siempre la decisión que yo tomara al final.

A mis hermanos por estar siempre ahí.

A los amigos que me han acompañado durante toda la carrera. Gracias a ellos todo ha sido mucho más llevadero.

A Carlos por ofrecerme este proyecto que he disfrutado desde el primer día y me ha permitido formar parte del GSI.

Y por supuesto, a Sergio, mi tutor, gracias a él desarrollar todo este proyecto ha sido mucho más fácil. Su ayuda me ha servido para aprender muchísimo.

# Contents

# List of Figures

CHAPTER 1

# Introduction

## 1.1 Motivation

Nowadays, Internet of the Things (IoT) has become in one of the most important development fields. The improvement of the communication networks has facilitated its evolution and 5G will be a significant impulse for IoT. In addition, the implementation of IPv6 has been an essential improvement because we will need a lot of IP directions.

The evolution of mobile phones into smartphones has caused an increase in their usage, increasing their impact in society and offering a wide ensemble of possibilities. Nowadays, everybody has a smartphone and uses it most of the day.

IoT offers us the ability to interconnect everyday objects with the objective of they can exchange data among them and with humans. As a result, we can monitor, control and automate some daily activities such as turn off the TV in a particular moment or send a message when we arrive at home. It has numberless possibilities for improving the life of the people, above all, for the dependent people.

It is important to take advantage of these interconnection facilities and to exploit its applications.

Currently, this technology research is focused on the development of Smart Environments [1]. Following that line of thought, several Smart Rooms projects were developed and their appliances are very diversified. Such "intelligent" or "smart" environments and systems interact with human beings in a helpful, adaptive, active and unobtrusive way. They may still be proactive, acting autonomously and anticipating the users' needs. In this project, we will focus on smart offices.

To build intelligent environments it is vital to design and make effective use of physical components such as sensors, controllers, and smart devices. Then, using the information collected by these sensors, the software can reason about the environment and trigger actions in order to change the state of the environment, by means of actuators. Such sensors/actuators networks need to be robust and self-organized in order to create an ubiquitous/pervasive computing platform. The design of these sensors/actuators leads us to some pervasive computing and middleware issues. Here we can find challenges as invisibility, service discovery, interoperability and heterogeneity, pro-activity, mobility, privacy, security and trust.

Smart offices aim is to make life easier and more convenient to humans. Security systems can be designed to provide lots of help in an emergency such as notify a fire, open doors automatically, communicate the situation to fire department or guide the occupants through the safest exit route.

Other examples, related to environmental parameters, are lights that turn off automatically when a person leaves a room or a temperature control that changes the temperature depending on who is inside. This example would result in a significant energy saving.

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

- In order to monitor and measure surrounding activity we use **sensors**. For example, luminosity, movement, or smoke sensors.
- The element in charge of make decisions based on programmed rules and occurrences are the **drivers**. They receive and process sensor's values. For example, a Chromecast device can be programmed to play a relaxing video if a user is nervous.
- Performing physical actions are needed. For instance, window openers or automatic light switches. This assignment is performed by **actuators**.
- Sometimes, the smart offices have a **central processing unit** which can be useful for maintenance and system changes. This central unit could be a private server in the subnet.

- The network is really important, as it transmits the signals in the system. Smart Office'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 windows can be directed to a particular window, but other times the order go to the system of windows.

## 1.2 Project Goals

The main aim of this project is to develop an intelligent automation platform based on ECA (Event-Condition-Action) rules, but it will have an advantage: the rules will accept multi-events and multi-actions. The rules will be executed in a rule engine (EYE) and they will be described using the EWE ontology.

Among the main goals of this project, we can highlight:

- Design and build a web server which enables users to create, import and remove multi-event driven rules.
- Develop a feature that allows the automate import of rules when an user arrives in a new smart place.
- Connect with a semantic rule engine that can run the multi-event driven rules and can manage its response.
- Develop a module that enables the connection with a Chromecast device.
- Develop a chatbot that allows an user to manage their rules.

## 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 where the motivation, the main goals, and the structure are described.
***Chapter 2*** explains the enabling technologies related to this project. Here some standards and technologies will be analyzed, in order to provide a technological background of the technologies used in the final project, giving a context to the main idea.
***Chapter 3*** presents the architecture we have adopted to implement the project, starting

with a global overview followed by the explanation of all the modules of the system.
***Chapter 4*** offers an overview of a selected use case, explaining the main functionalities in order to help the user to understand the overall concept.
***Chapter 5*** sums up the conclusions drawn from this project, problems faced and suggestions for a future work.

CHAPTER 2

# Enabling Technologies

## 2.1 Introduction

In this chapter, the main technologies and resources which have been used in this project are going to be presented. Mainly, we will explain the rule automation, which includes the explanation of the ontology that describes them (EWE), the engine on which rules are executed (EYE) and other existing applications. The technologies that will help to develop the Telgram and Chromecast modules will be explained too.

## 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. A rule allows you to automatically perform actions based on various triggers. Sometimes, a condition is required to perform the action. 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 [2].

A number of now prominent websites, mobile applications, 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 instances of this simple task automation are "When I run a voice command, create a reminder" or "When I am in a precise place, turn on the lights". They receive the name of Task Automation Services (TASs).

### 2.2.1 Task Automation Service

The TASs present a visual programming environment, where any user, though they don't have technical knowledge, can easily create and manage their own personal automations. In these services, the automation takes the form of Event-Condition-Action rules that execute an action when an event is triggered. In the above examples, the voice command and the presence in a precise place would be the triggers, whereas creating a reminder and turning on the lights are the actions. Their growth can be explained using three factors. Firstly, usability, the TASs' interface is very intuitive. Secondly, customizability, each user can program the automation that they need. Finally, integration with existing Internet services [3].

There are multiple TAS platforms, and each platform has its execution strategy. However, all platforms follow the same line of thinking (if this then that), which cause that writing rules become very natural. The used engine should be very expressive to maximize the configure ability, and there's a language that stands out from the rest in this point: Notation3.

### 2.2.2 Notation3

Notation3 (N3) is a shorthand non-XML serialization of Resource Description Framework models. N3 is much more compact and readable than XML RDF notation. It is being developed by the Semantic Web community and has support for RDF-based rules.

The aims of the language 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, and

- 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 property 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 is capable of describing everything using triples, but also capable of describing rules to be executed on those triples. The expressiveness of N3 differentiates it from other rule languages [4]. For instance, in N3 it is possible to create rules in the consequence and to use built-ins. There are some rule engines like RDFox, FuXi or EYE that support N3 syntax. Nevertheless, EYE is more expressive than RDFox or FuXi, whilst being more performant than other N3 reasoners.

### 2.2.3 EYE

EYE (Euler YAP Engine) [5, 6] is a reasoning engine supporting the Semantic Web layers. It performs semi-backward reasoning and it supports Euler paths. The Euler path detection is roughly "don't step in your own steps" to avoid vicious circles. EYE interprets each logical rule P $\Rightarrow$ C (where P is a precondition and C is a consequent) as P AND NOT(C) $\Rightarrow$ C, so that rules execute only when they can generate new triples. The semi-backward reasoning is a backward reasoning for rules using $\Leftarrow$ in N3 and forwards reasoning for rules using $\Rightarrow$ in N3.

This reasoner has built-in knowledge; it would know what some concepts mean and how to apply them. The drawback, of course, is that only built-in concepts can create new knowledge. So, EYE was designed to have the least amount of inherent knowledge; it's extensible through rules so that new knowledge can be created. For example, SymetricProperty definition works this way:

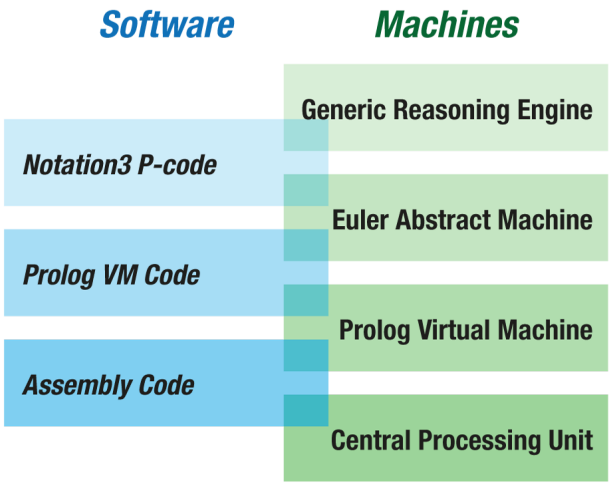


Figure 2.1: Main building blocks of EYE

```
{
  ?predicate a owl:SymmetricProperty.
  ?subject ?predicate ?object.
}
=>
{
  ?object ?predicate ?subject.
}.
```

The reasoning that EYE is performing is grounded in FOL (First Order Logic). Keeping a language less powerful than FOL is quite reasonable within an application, but not for the Web.

A key characteristic of EYE's architecture is portability because it is crucial to the Semantic Web. EYE runs in a Prolog virtual machine, which runs directly on the CPU. The core of EYE is compatible with two major Prolog engines (YAP and SWI-Prolog). This machine accepts N3 P-code, which is a Prolog representation resulting from parsing RDF triples and N3 rules.

### 2.2.4 EWE Ontology

Evented WEb Ontology (EWE) [7] is a standardized data schema (also referred as "ontology" or "vocabulary") designed to describe elements within Task Automation Services enabling rule interoperability. EWE models the most important aspects of task automation

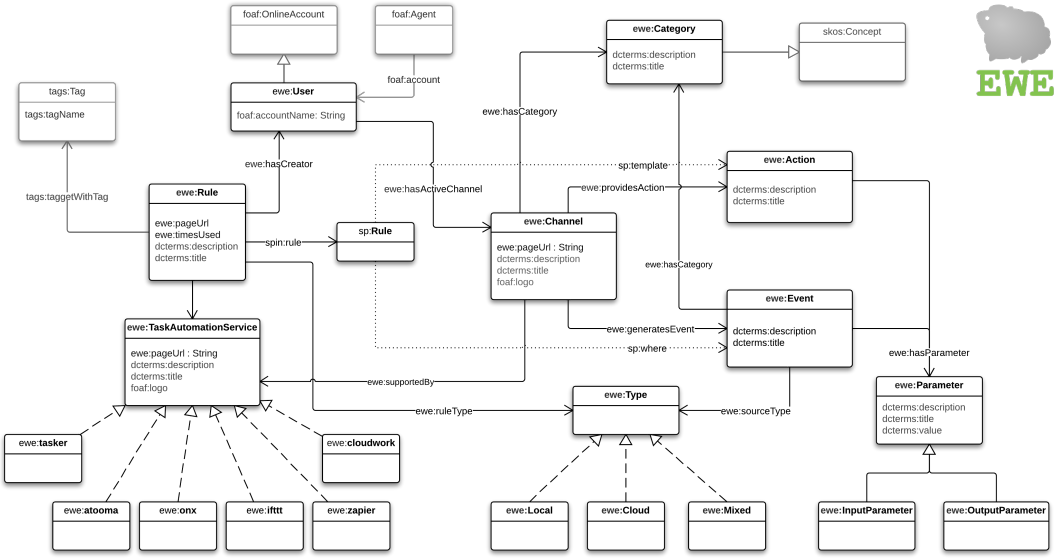


Figure 2.2: EWE Class Diagram

services from a descriptive approach. The goals of the EWE ontology to achieve are:

- Enable to publish raw data from Task Automation Services (Rules and Channels) online and in compliance with current and future Internet trends.
- Rule interoperability.
- Provide a base vocabulary for building domain specific vocabularies e.g. Twitter Task Ontology or Evernote Task Ontology.

The ontology has four main classes: Channel, Event, Action and Rule. Each class can have attached several properties or attributes. The important properties and the purpose of including them within the overall EWE ontology are hasParameter, hasCategory, activeChannel, hasCreator and spin:rule.

The property hasParameter presents the parameters of an Action or an Event. For instance, an event triggered when a particular user post a tweet must be declared the name of the user, i.e. the parameter that defines the particular user.

The property hasCategory indicates that a Channel, Event or Action belongs to a certain category. The EWE Ontology does not provide a taxonomy of channels, events, and action; but it facilitates building that classification. Some example could be the mobile category, which consists of all internal mobile resources (GPS, Bluetooth, WiFi, Calendar...), or apps category formed by third-party applications like Twitter, Facebook, LinkedIn etc. The Channel categorization is important for channel discovery and recommendation. This

happens not only with discovery and recommendation methods based on profiling, but also with methods based on semantic similarity. Besides, expert systems may use the channel categorization and help a user to find alternatives to other channels.

The hasActiveChannel property links users to Channel on which they have an account.

The property hasCreator links instances of Rule to its creator, an onlineAccount from the FOAF ontology. The authors of the rules are significant information for data analysis purposes and recommendation systems.

The spin:rule property links rule instances with the SPARQL execution logic described using the spin vocabulary. In addition, ewe:triggeredByEvent and ewe:firesAction properties are defined to link the rule with the event that triggered it and the consequent action that was executed. These two properties simplify the query required to select the events and actions that are related to a rule instance, removing the extra triple patterns required to navigate from the Rule instance to the Event and Action instances through the spin:rule.

#### 2.2.4.1 Channel

It is the element that produces Events, triggers Actions or both. A Channel can be a connected-to-the-internet object, like a TV; can be an app, like Twitter, or almost anything. For example, turning on the TV is an Action and receiving a direct message is an Event.

#### 2.2.4.2 Event

An Event is a process which occurs in a particular moment. They don't have duration over the time, they are instantaneous.

Sometimes, Events have important information about the channel that generates them. These details can be used within Rules to customize Actions: they are modeled as output parameters. The user can describe under which conditions the Event will be triggered. These are the configuration parameters, modeled as input parameters. Different services may generate the same events, so Event definitions are not bound to certain Channels.

#### 2.2.4.3 Action

An Action is a process which occurs when an Event has been triggered and a condition is satisfied. This process is provided by a Channel. It includes switching on a light in a physical location, modifying states on a server or even posting a tweet. 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. A Rule links the events with the actions; those include the configuration parameters set for both of them: output from Events to input of Actions. This project's platform allows multi-events and multi-actions rules. In other words, a user can define rules where various events are necessary to execute an action, where various actions are executed when an event is triggered, or a combination of both.

### 2.2.5 IFTTT

IFTTT[1] ("if this then that") is a web service where the user can create a conditional task. The "this" can be a local event, like turn on Bluetooth, changes in other web services, such as Twitter or GitHub, or a direct order, like a voice command of Google Assistant. The "that" can be a physic actuator, like a door, or another web service, for example, Facebook.

In IFTTT's platform, the users can work with thousand of channels, so they have infinite combinations, but they can't create a multi-event or multi-action rule (it isn't "if this, this and this then that and that").

Currently, it is used and shared by hundred of users. This platform is very complete, but it has some defects: limitation in the creation of rules, it only has default channels, the users can't create them and its channels are not always enough.

## 2.3 EWETasker 1.0

In this section, EWETasker [8] will be explained, the previous version of this project. EWETasker is a web application which does rule-based task automation. The rules have an Event-Condition-Action (ECA) structure, this means that an action will be executed if a condition is met when an event is received. The rules are written in Notation3.

In EWETasker, the users can create channels with events and actions. With these channels, or with channels previously created, the user can create rules and import them. The user can also import existing rules. He can test the rules and simulate the events.

[1] https://ifttt.com/discover

EWETasker has three main modules: Task Automation Server (TAS), Mobile App and Google Glass App.

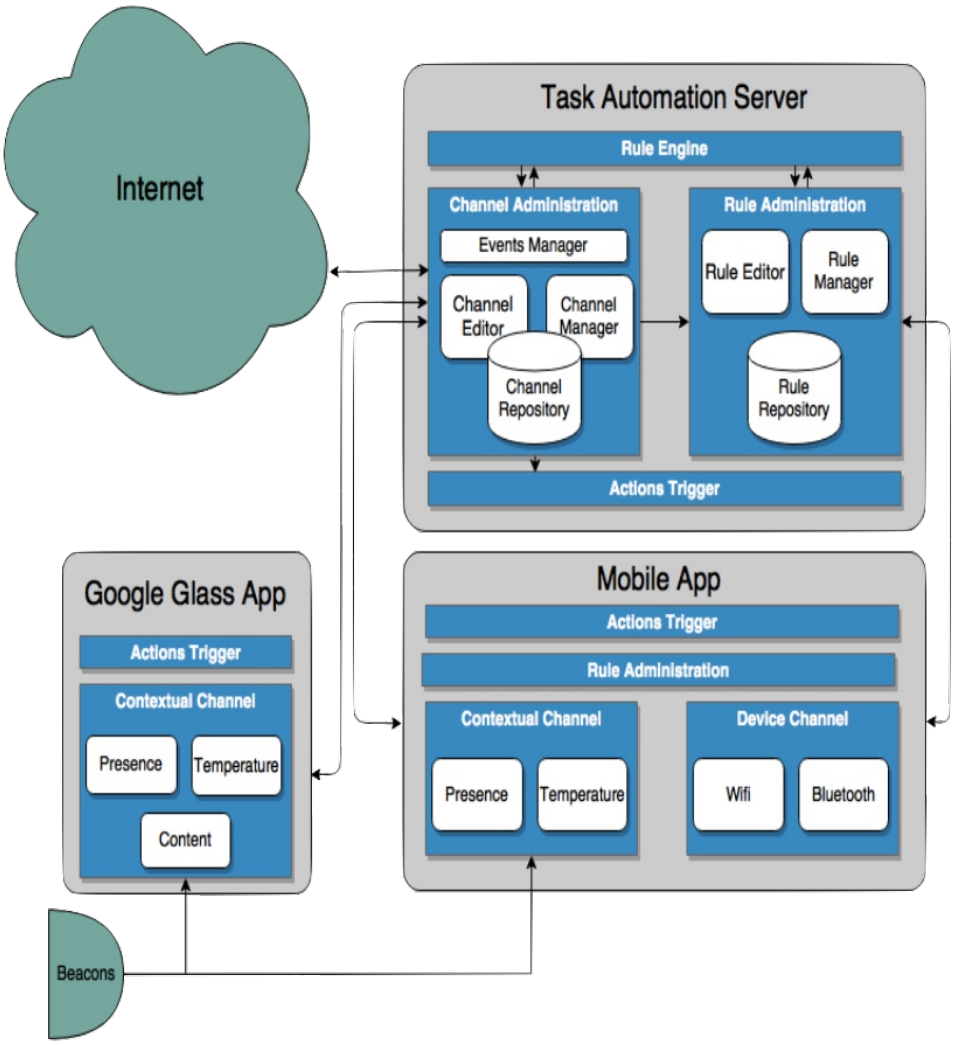


Figure 2.3: Architecture

### 2.3.1 Task Automation Server

This module is in charge of automating certain tasks with the objective of performing an action when an event is received. This module has four submodules:

- Rule Engine: when an event is launched, this submodule evaluates the rules imported by the user. If a condition is met, an action is triggered.
- Channel Administration: that submodule allows to create and manage the channels. It manages events too, and it sends to the Action Trigger the action that it has to

trigger.

- Rule Administration: it allows the user to create and manage the rules. The rules are formed by events and actions, and these events and actions are provided by Channel Administration.

- Action Trigger: submodule that triggers an action when it receives the appropriate response.

Figure 2.4 displays the interconnection among submodules.

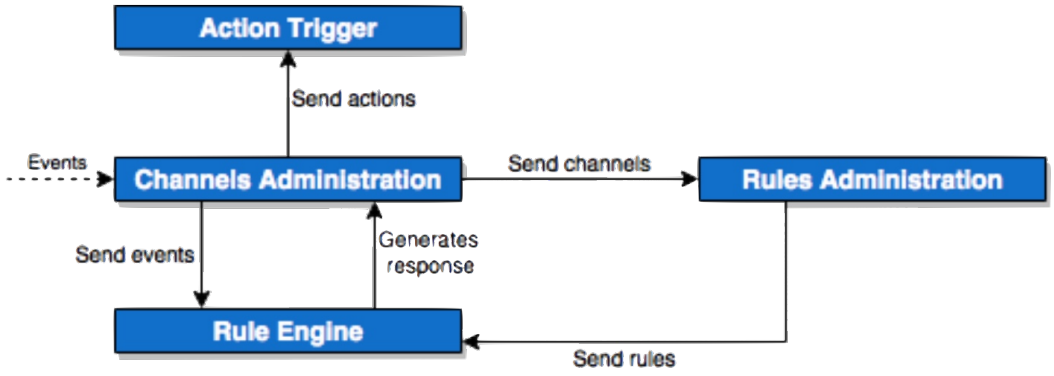


Figure 2.4: TAS Sub-modules Interconnection

#### 2.3.1.1 Rule Engine

Rule Engine is a very important module, and it is based on ontology model, specifically, on EWE ontology [7]. It is composed of the EYE Server and the EYE Helper.

Firstly, EYE Helper captures the new events, loads the rules and sends to EYE Server these events and rules. EYE Server evaluates this data, generates a response and sends it to EYE Helper again. Finally, EYE Helper sends the response to Channel Administration.

EYE Server is stateless, so all the events and rules will have to be load before a new inference. The event and the rule are removed from the model once the event has been inserted into the ontology 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.

#### 2.3.1.2 Channel Administration

The purpose of Channel Administration is to create and to manage the channels and to manage the events. It is formed by Channel Editor, Channel Manager, Channel Repository and Events Manager.

Channel Editor provides a graphic interface to create and to edit the channels. The user can define various events and various actions in the channels. Each event and each action have three fields: title, rule, and prefix.

Channel Manager is a controller that provides channels management and persistence. It has a graphic interface that displays the list of the channels that are in the Channel Repository.

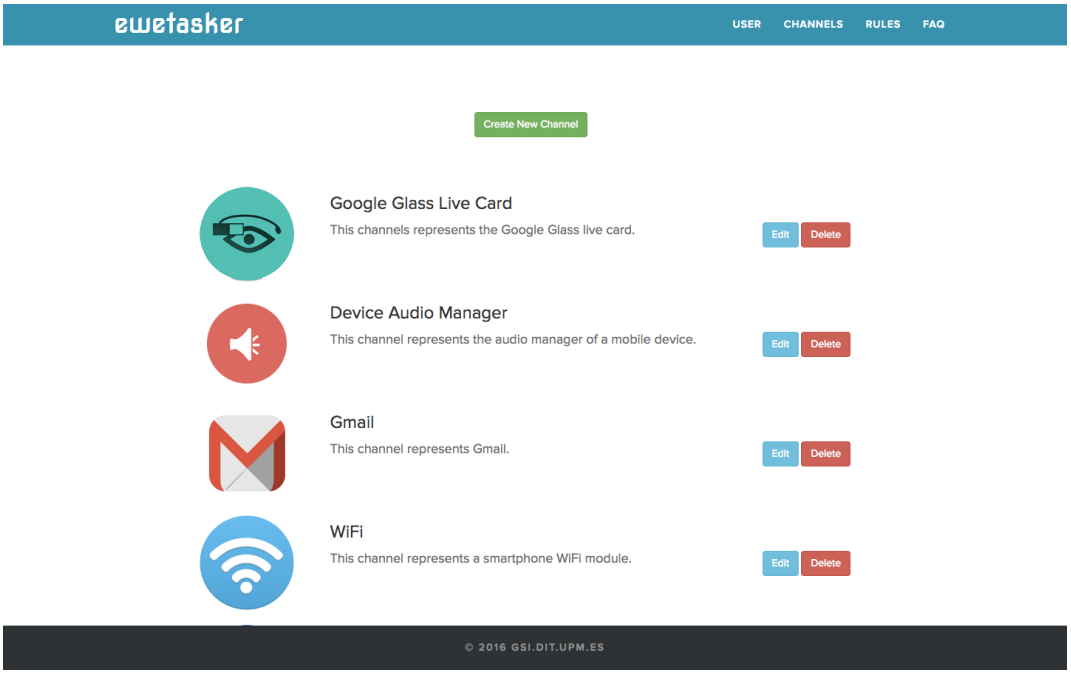


Figure 2.5: Channels Manager Interface

The channels are stored in the Channel Repository, this repository is a MongoDB database. Mongo databases use a JSON format and are organized into collections. This repository has three collections: channels, events, and actions.

Events Manager is an important submodule whose aim is to connect with EYE Helper and translate Notation3 to code that we can work with it. It also sends the action to the Action Trigger.

#### 2.3.1.3 Rule Administration

The objective of Rule Administration is to create and to manage the rules. It has three parts: Rule Editor, Rule Manager, and Rule Repository.

The connection between events and actions through the structure "If this then that", results in an automation rule; so if the condition is matched the action is triggered. For instance, "if the room is dark, increase the brightness of the light".

The rules are created by the Rule Editor with a graphic interface based on icons and "drag and drop" actions.

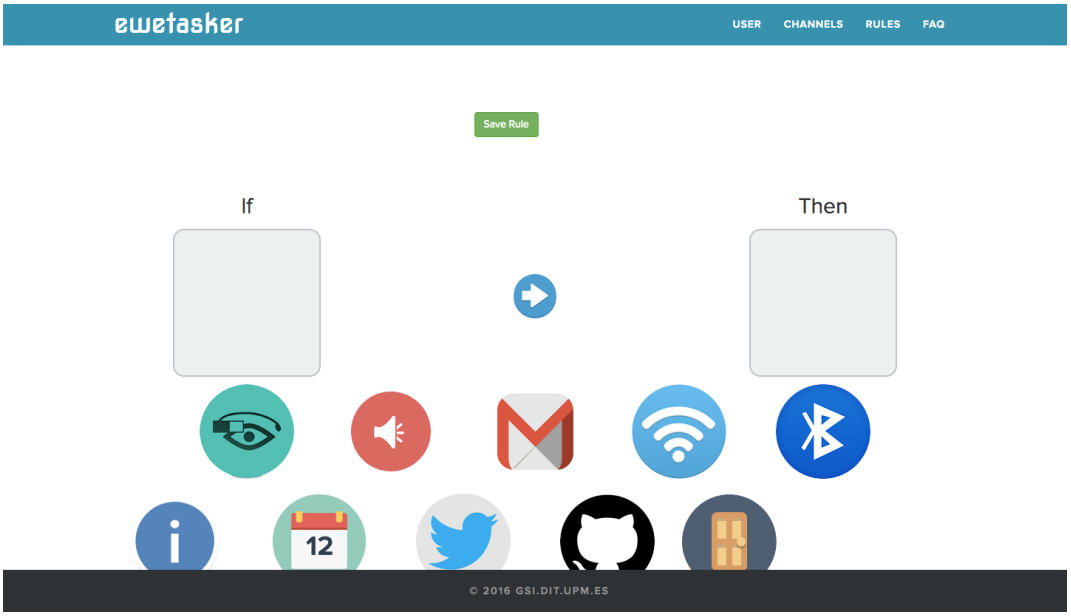


Figure 2.6: Rule Editor Interface

The user can manage them with the Rule Manager, it also has a graphic interface where the user can see them and import them.

Once the rule is created, it is saved in the Rule Repository. The Rule Repository is a MongoDB database where all rules are stored. This repository has only one collection: rules.

#### 2.3.1.4 Action Trigger

Action Trigger is the module in charge of executing the actions. It receives a JSON from the Events Manager with the action it has to execute, the channel that executes the action and, if it is necessary, the parameter of the action.

Not all the actions are executed from the TAS, some actions are executed from the Mobile App and from the Google Glass App, so these modules also have an Action Trigger.

### 2.3.2 Mobile App

The Mobile App [9] is a very useful module that is used in this project. It can been use the modules of the mobile (Bluetooth, WiFi, Calendar...) and others sensors like Beacons, which can connect with the mobile, like an event. The user can also use those modules (WiFi, Audio Manager...) like an action.

#### 2.3.2.1 Rule Administration

Rule Administration of the Mobile App is capable of creating rules and save them in the Rule Repository of the TAS.

Firstly, it connects with the Channel Manager of the TAS. Secondly, Channel Manager sends the list of channels. Then, the user chooses the event and the action, and, if it is necessary, chooses the parameter. Next, the user sets a title and a place. Finally, it sends the rule to the Rule Administration of the TAS and it stores it in the Rule Repository.

#### 2.3.2.2 Action Trigger

As explained earlier, Action Trigger is a submodule that appears in different modules. Mobile App needs an Action Trigger to execute some actions like Turn On Bluetooth or Silence the Mobile.

The mobile uses internal modules like an event. When an event occurs, the Mobile App sends it to the Events Manager of the TAS, and it responds with the actions that the mobile has to execute. These actions are received by the Action Trigger and it executes them.

## 2.4 MongoDB

MongoDB[2] is an open source NoSQL database that is oriented to documents. That is to say, saves the data in documents. The format of the storage is BSON, which is a binary representation of JSON. The documents from the same collection - such as a table in a SQL database - could have a different scheme, so each document of a collection could have

[2]http://www.mongodb.com

different fields. MongoDB documents tend to have all data for a given record in a single document, whereas in a relational database information for a given record is usually spread across many tables. As a result, it is an agile database.

Unlike most NoSQL databases, MongoDB provides comprehensive secondary indexes, including geospatial and text search, as well as extensive security and aggregation capabilities. MongoDB's flexible document data model presents a superset of other database models. It allows data to be represented as simple key-value pairs and flat, table-like structures, through to rich documents and objects with deeply nested arrays and sub-documents.

MongoDB provides strong consistency, great flexibility, high scalability, and high performance.

## 2.5 Telegram

Telegram[3] is an instant messaging app which allows to send and receive messages, photos, gifs, videos, voice, stickers, and files. It is synchronized in the cloud with different platforms, like, smartphone, tablet, computer or web-version. The user can create groups, where all member can write, and channels, where only the admins can write. However, bots are its most interesting feature.

A bot is a software application that runs automated tasks. If the bot's purpose is to offer a service through a conversation, it will be named chatbot. Actually, it is the variety of bot which the user can create in Telegram. The BotFather is a tool that enables the development of new bots for Telegram. It helps the developer to create the basic needs of the bot, i.e. it creates the chat and the means of communication between user and bot. It also enables to set some characteristics, for example, privacy, the option to join chat groups, the commands the user can use, etc. The only thing that The BotFather can't make is the code of the bot.

A user can choose among many programming languages to develop a bot in Telegram, included PHP. Particularly, this project uses the *Longman* library [10].

[3]http://www.telegram.org

## 2.6 Chromecast

Chromecast[4] is a line of digital media players developed by Google. The devices enable users with a mobile device or personal computer to initiate and control playback of Internet-streamed audio/visual content on a television or home audio system through mobile and web apps that support the Google Cast technology. Alternatively, content can be mirrored from the Google Chrome web browser running on a personal computer, as well as from the screen of some Android devices.

Chromecast devices are dongles that are powered by connecting the device's micro-USB port to an external power supply or a USB port. Video-capable Chromecasts plug into the HDMI port of a high-definition television or monitor.

It is really easy to install, the user only has to connect the Chromecast device, download the Home App, search for the device and synchronize the WiFi network from the smartphone to the device.

The Google Cast SDK [11] allows third parties to modify their software to work with Chromecast and other Cast receivers and we will use it in the mobile app.

In addition, there is a python library named *pychromecast* [12] that allows connecting with Chromecast and sending it media. This library allows us connecting with it from the server.

[4]https://www.google.com/chromecast

CHAPTER 3

# Architecture

## 3.1 Introduction

In this chapter, the architecture of this project will be explained, including the design phase and implementation details. First of all, in the overview we will present a global vision of the project architecture, identifying the visualization server and other necessary modules. Secondly, this chapter will focus on each module explaining its purpose in this project. Finally, the current and the previous architecture will be compared.

## 3.2 Overview

In this chapter, the architecture of this project and the design phase will be detailed. There are numerous modules that will be explained in detail.

In general, the architecture is very similar to the architecture of EWETasker 1.0. The two projects have in common many modules. However, most of the modules work differently and the modules of this project are capable of performing more tasks and have better features.

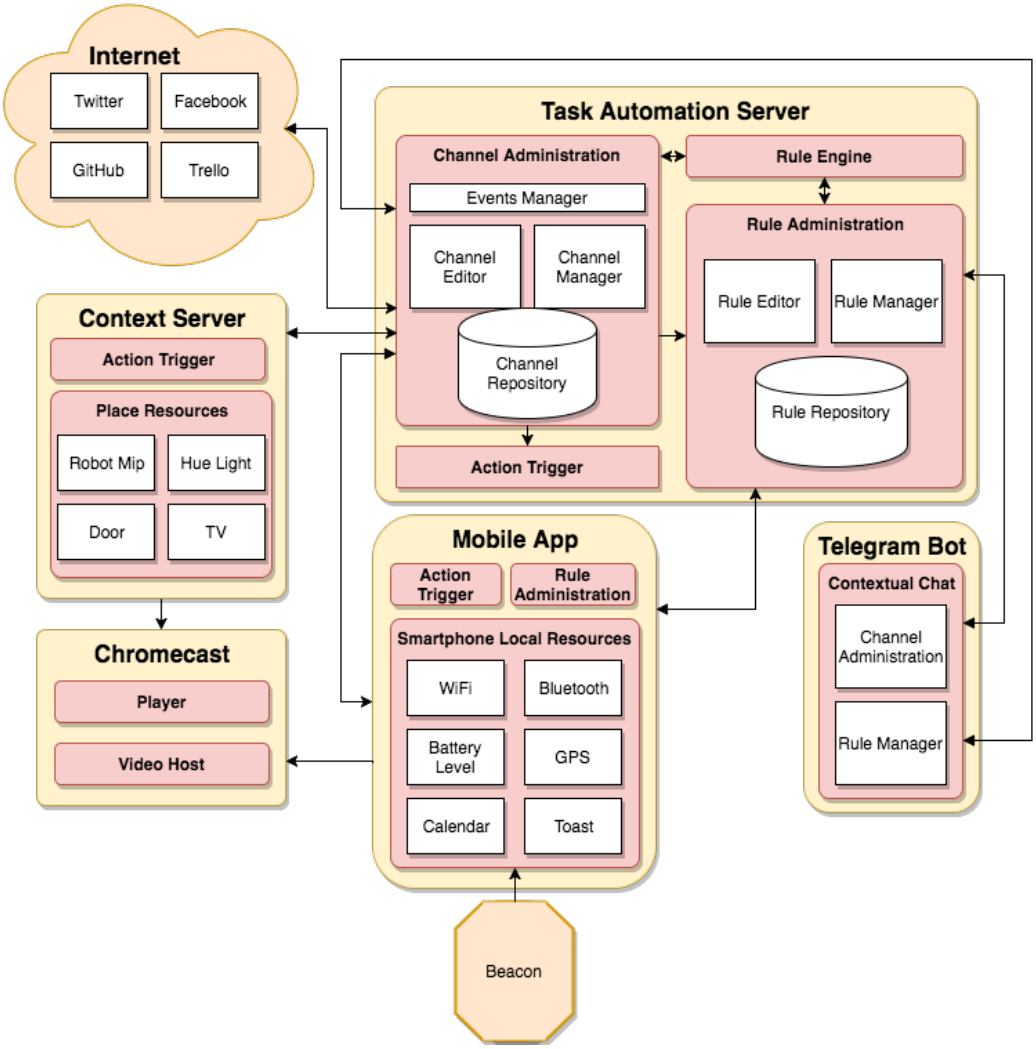


Figure 3.1: Architecture

The project is divided into four main modules and they are divided into several submodules:

- **Task Automation Server (TAS)**: it is the main module of the project. This module connects to the Internet, handles events and triggers the correct action generated by the Rule Engine. It also manages the channels and the rules. The module is distributed in four submodules:
  - Rule Engine.
  - Channel Administration.
  - Rule Administration.
  - Action Trigger.

- **Mobile App**: its principal aim is to handle the events of the device and of the Beacons, and sends them to the TAS. It can create rules, save them in the Rule Repository of the TAS and triggers actions. Mobile App has two important submodules:
  - Rule Administration.
  - Action Trigger.
- **Telegram Bot**: this bot manages the users' rules. It allows the user to import and to remove rules among the platform's rules. It also shows the imported rules of the user. It can send an event to the TAS and trigger actions from the TAS.
- **Context Server**: it is a public server that is located in every smart place. Its job is trigger actions when the channel's physic device and the TAS aren't in the same subnet. It also allows differentiating a channel in different places. It only has a submodule: Action Trigger.
- **Chromecast**: it is a module that enabled the connection between the Mobile App and the Chromecast device, and between the Context Server and the Chromecast device.

The mobile, the bot and the Internet are sources of events. These events are sent to the Channel Administration of the TAS, and Channel Administration sends the response with the action to the correct Action Trigger. The rules can be created by the TAS or the mobile, but only the TAS can create multi-event and multi-action rules. These rules can be managed by the TAS, the mobile, and the bot.

This project is principally focused on the TAS and the Telegram Bot.

## 3.3 Task Automation Server

It is a fundamental module of the project. The basic objective of this module is to handle the events, to evaluate them with the rules and to generate a response with the action. This work is performed by the **Rule Engine** that is the core of the module. Nevertheless, every submodule is important to reach the final objective. There are two submodules that manage the events, actions, and rules: the Channel Administration and the Rule Administration. The **Channel Administration** handles the channels that can generate events and execute actions, the events and the actions are used to create rules. **Rule Administration** handles those rules, and can create, edit and delete them. Finally, the **Action Trigger** executes the actions of the channels that receives from the actions from the Channel Administration.

It works in the same way that the TAS of EWETasker 1.0, so the interconnection that we can see in Figure 2.4 is the same.

### 3.3.1 Channel Administration

A way to manage channels, events, and actions is needed. As a result, we have the Channel Administration which creates and edits the channels that are composed of events and actions. Channel Administration also manage the events that are received and sends them to the Rule Engine. When the Rule Engine sends the response, Channel Administration parses it and sends the result to some Action Trigger depending on its nature. This submodule stores every action that is executed in a database and produces a list with the most-run actions. These tasks are distributed in different parts: Channel Editor, Channel Manager, Channel Repository, and Events Manager.

#### 3.3.1.1 Channel Editor

It is a graphic interface that helps the user during the creation of the channel. The module is designed with HTML, Javascript, and CSS.

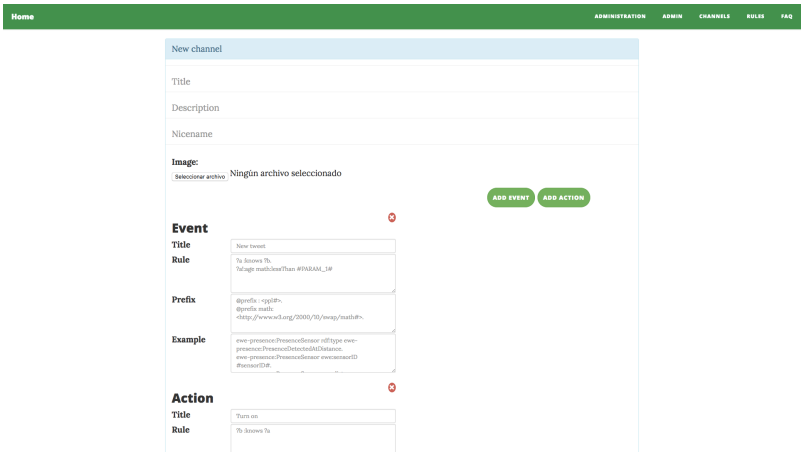


Figure 3.2: Channel Editor Interface

Each channel has a *title*, it is unique and identifies the channel; a *description*, a *"Nice Name"*, it is the name that will be visible; an *image*, it should be representative, and *events* and or *actions*. The events and the actions are the part most important of the channels. They can be differentiated with various fields:

- The **title** identifies the action or the event.

- The **rule** field represents the event or the action and is the text that will form the rule. It is written in Notation3. For instance, the following rule represents the event of Telegram's message with a particular text:

```
?event rdf:type ewe-telegram:EventCommand.
?event!ewe:text string:equalIgnoringCase #text#.
```

  The events and the actions can have parameters. If they have, the parameters will be set when the rule is created. However, the rule field must mark that the event or the action need them. It marks it including the name of the parameter between two "#". It doesn't have to be the name, but it helps the user to know what have to write. That indicates to the Rule Manager that this event or action has some parameters that must be configured, and the expression will be replaced with the parameter selected by the user.

- It is really helpful using a **prefix** because a lot of characters are replaced by an only word. It simplifies the rule. In this field, the user indicates the prefix that will replace the long text. For example, the before rule needs these lines to work:

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

- The **example** field is only in the events, not in the actions. This field is an example of the event, written in Notation3, that will arrive at the Events Manager. This will serve to speed up the tests with the EYE Engine. This is the only field that is not required.

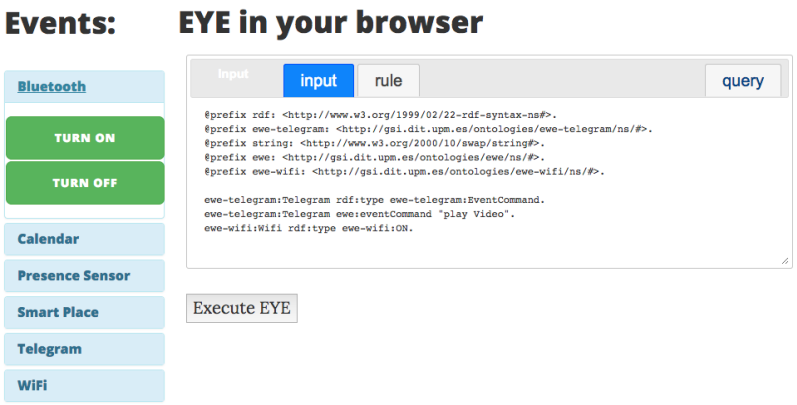


Figure 3.3: Event Example

The user can add to the channel indeterminate events and actions, or even remove them. He only has to push the appropriate button. When all the fields are filled, the user has to click the button "send" and Channel Editor will store the channel. Obviously, the channel has to be correctly created. It stores the channel with a particular structure that will be sent to the server, in order to communicate the created channel to Channel Manager. The structure is as follows:


```
{
    "_id" : ObjectId("5921a8242c86701b590a64e2"),
    "title" : "chromecast",
    "description" : "It is a Chromecast device",
    "nicename" : "Chromecast",
    "image" : "../img/chromecast.png",
    "events" : [ ],
    "actions" : {
        "1" : {
            "title" : "Welcome",
            "rule" : "ewe-chromecast:Chromecast rdf:type ewe-chromecast:
                Welcome.",
            "parameters" : [ ],
            "prefix" : "@prefix rdf: <http://www.w3.org/1999/02/22-rdf-
                syntax-ns#>.
            @prefix ewe-chromecast: <http://gsi.dit.upm.es/ontologies/ewe-
                chromecast/ns/#>."
        },
        "2" : {
            "title" : "Play Video",
            "rule" : "ewe-chromecast:Chromecast ewe:providesAction ewe-
                chromecast:PlayVideo.
            ewe-chromecast:PlayVideo ov:video #video#.",
            "parameters" : [ "video" ],
            "prefix" : "@prefix rdf: <http://www.w3.org/1999/02/22-rdf-
                syntax-ns#>.
            @prefix ewe-chromecast: <http://gsi.dit.upm.es/ontologies/ewe-
                chromecast/ns/#>.
            @prefix ov: <http://vocab.org/open/#>."
        }
    }
    "createdAt" : "2017-05-21 14:45:56"
}
```


This module connects to the Channel Manager in order to store the channels.

#### 3.3.1.2 Channel Manager

Channel Manager controls and manages the channels and provides persistence. For this purpose, it displays a graphic interface with a list of the existing channels where the user can create and edit them. It is implemented in PHP following a Model-View-Controller (MVC) pattern.

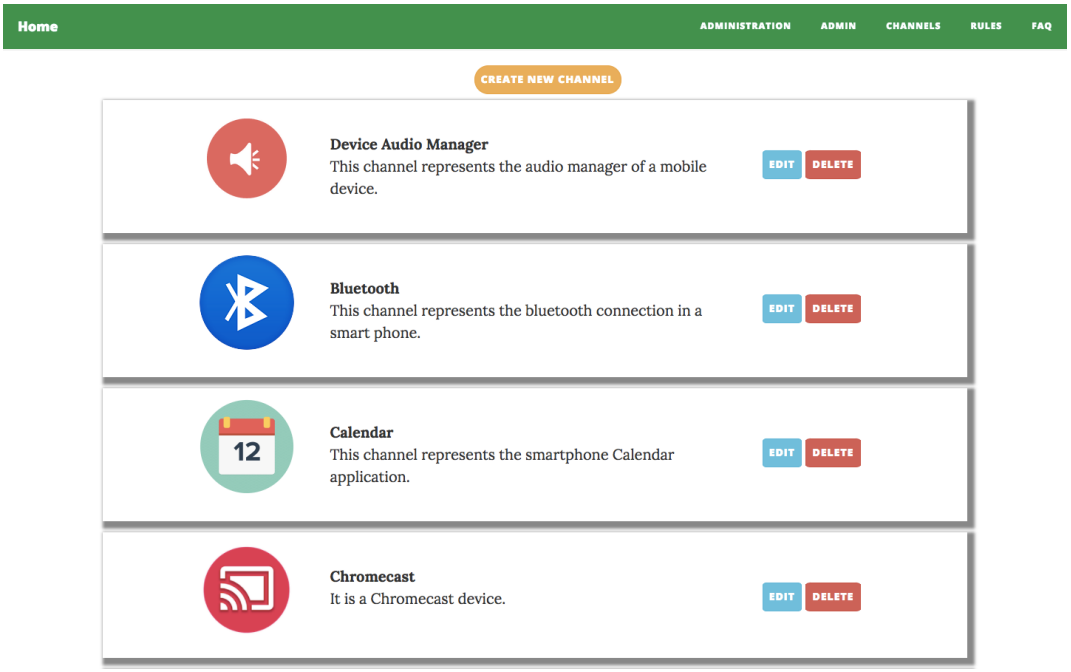


Figure 3.4: Channel Manager Interface

Channel Manager also manages the executed actions and provides an sorted list of the most-run actions. When the Events Manager sends the channels with their actions to the Action Trigger, it also sends them to the Channel Manager. It updates the repository with the number of times that the action has been executed. This module has been implemented in PHP and it displays a graphic interface to the administrator users with the list. It can be used in the future to make recommendations to the user.

This submodule is very important to the Mobile App too. Channel Manager sends to the Mobile App the list with the channels, events, and actions when the mobile wants to create a rule. Nevertheless, the mobile can't create a new channel.

The channels are stored in the Channel Repository and the Rule Editor picks up the channels from the Channel Repository to create the rules.

#### 3.3.1.3 Channel Repository

The channels, with its events and actions, are stored in a MongoDB database. Mongo databases use JSON format and organize the data into collections. This repository has two collections: channels and most-run actions. The events and the actions of the channels are stored with them.

#### 3.3.1.4 Events Manager

It is a very important part of the Channel Administration because all the events go through the Events Manager. It sends the events and the rules to the Rule Engine, the Rule Engine evaluates the rules and sends to the Events Manager the response with the actions. The response includes the input (the events and the prefixes that Events Manager sends) and the actions, and is written in Notation3. Parse the response is needed because the platform only is interested in the actions and it needs to change the Notation3 format. Once the response is parsed, it has only the actions and it has them in a workable format. Now, it can send the actions to the right Action Trigger and to the Channel Manager.

Each input of an event is saved during ten seconds, and each time an input is received, this module sends to the Rule Engine the new input and all the saved inputs. Normally, the triggers only send one event by time, which makes impossible to drive multi-event rules. This way, we correct it.

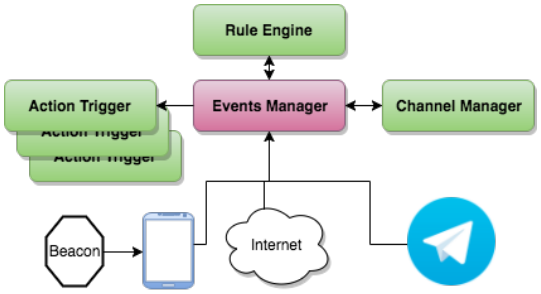


Figure 3.5: Events Manager Interconnection

The proccess of parsing is very important, so it will be explained in detail:

Firstly, it takes the response and the input. Both data is needed.

Secondly, the parts that do not contribute with util information are removed. It removes the prefixes, the comments and the break lines from the response and the input. All comments must be between #C and C#, it eases the job.

As explained above, the response also includes the input. Then, it removes the input from the response. Before, it had split the response and the input into sentences. This allows to compare sentence by sentence and it will help in the future.

If the action had a parameter, the action would use two different sentences. In a line, it is the name of the channel and the name of the action, and in another line, it is the name of the action and the parameter, so the second line is the line that will be used in this step. Each sentence (that is a triple pattern that represents an action) is split into three fragments: subject, object, and predicate. The subject corresponds to the name of the action and the predicate corresponds to the parameter. The name of the action and the parameter have to be saved in an array to the further use.

In the first line, subject and predicate correspond with the channel and the action respectively. We can separate them and save the channel and the action in an array, it checks in the previous array if this action has an associated parameter and add it to the new array. It also adds the name of the user who has generated the event.

Each Action Trigger module has a list of channels whose actions is able to trigger. Therefore, it sends the actions one by one to the Action Trigger that is able to trigger them. For instance, the action "TurnOn" of a Light channel has to be sent to the Action Trigger of a Context Server, but it has to know the correct server. The place of the rule gives us that information.

Before sending it, it converts the array to a JSON. The structure of the sent data is:

```
"channel" : "Chromecast",
"action" : "PlayVideo",
"parameter" : "fire",
"user" : "carlos"
```

In conclusion, the Channel Administration module is responsible for providing channels creating, editing and removing functions, and managing the most-run actions; but has also one of the main roles of this project: the events management. The events received are converted to actions and passed to the Action Trigger modules.

### 3.3.2 Rule Administration

It this platform, it is necessary a module that handles the rules, a module that allows an easy creation of rules and the user can manage them. This module is the Rule Administration. Apart from these tasks, this submodule also allows keeping a list with the most popular rules. The most remarkable feature, as opposed to the previous version and other platforms,

is the creation of multi-event and multi-action rules. It is divided into three parts: Rule Editor, Rule Manager, and Rule Repository.

#### 3.3.2.1 Rule Editor

Rule Editor provides an easy way to create a multi-event and multi-action rule. As a result, it provides a graphical interface based on icons and "drag and drop" actions. The user only has to drag the icon of a channel and drop it in the container. The interface is implemented with HTML, Javascript, and CSS.

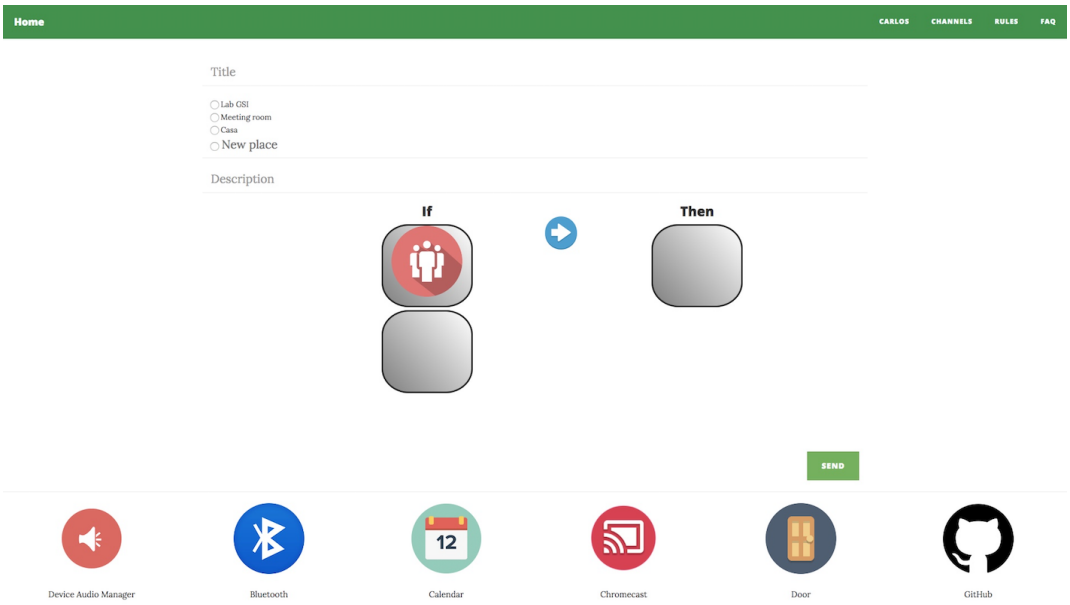


Figure 3.6: Rule Editor Interface

In Figure 3.6 we can see the graphical interface for the rules creation. The user can select the channel, drag it and drop onto one of the two initial containers. 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) and, if the event or the action needs parameters, it will show a dialog with a field to fill them. When the event or action is chosen and the parameters are filled, a new container for events or a new container for actions will appear, like we can see in the Figure. This is the way to generate multi-events and multi-actions rules. The left containers are for the channels which provide events, and the right containers are for the channels which provide actions. The channels without events can't be dropped onto the left containers and the channels without actions can't be dropped onto the right containers. Moreover, there are more fields that have to be filled:

- **Title**: it identifies the rule and it is the name that will be visible.
- **Place**: space where the events of the rule are captured or where the actions will be triggered. For instance, a lab in the university. It is important because the platform has to know if it must turn on the light of your house or the light of your office. The user can select a place where there are rules defined or select a new place. If he selects a new place, it will have to provide an ID to that place and the URL of the place's Context Server.
- **Description**: it is a brief explanation of the rule. It helps other users to know what do the rule.

Once the user has filled these fields, it can store the rule. Rule Editor uses the following structure:

```
{
    "_id" : ObjectId("5916dfae2c8670112c7f5d6b"),
    "title" : "Meeting",
    "description" : "Silence mobile in the meeting room if it is meeting
       time",
    "place" : "Meeting room",
    "author" : "carlos",
    "event_channels" : [ "calendar", "smartPlace" ],
    "event_titles" : [ "Event Start", "Inside Of" ],
    "action_channels" : [ "audio" ],
    "action_titles" : [ "Silence" ],
    "rule" : "
        @prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
        @prefix string: <http://www.w3.org/2000/10/swap/string#>.
        @prefix ewe: <http://gsi.dit.upm.es/ontologies/ewe/ns/#>.
        @prefix ewe-calendar: <http://gsi.dit.upm.es/ontologies/ewe-
            calendar/ns/#>.
        @prefix ewe-place: <http://gsi.dit.upm.es/ontologies/ewe-place/ns
            /#>.
        @prefix ewe-audio: <http://gsi.dit.upm.es/ontologies/ewe-audio/ns
            /#>.
        {
            ?event0 rdf:type ewe-calendar:EventStart.
            ?event0!ewe:eventTitle string:equalIgnoringCase "meeting".
            ?event1 rdf:type ewe-place:Inside.
            ?event1 ewe:placeID ?placeID1.
            ?placeID1 string:equalIgnoringCase "7g8h9i".
        }
        =>
```

```
        {
            ewe-audio:AudioManager rdf:type ewe-audio:Silence.
        }.",
    "createdAt" : "2017-05-13 10:27:58"
}
```

In this snippet, "?event" is a variable which will be filled in when the rule is executed. Variables in N3 can be recognized by a leading question mark "?". In the previous listing, we can see that the part of the events uses variables to generalize the rule for all calendars, all places, etc. If the rule has multi-event, it needs a different variable for each event, so Rule Editor adds automatically a number after the variable.

In the project, there is a special kind of rules that are named Admin Rules. They are a class of rules that only can be created by the administrator user and they have unique features: when an Admin Rule is created, it is imported to every user, and when a user signs up, he will import automatically every Admin Rule and he can't remove them.

This submodule connects to the Rule Manager for storing the rules, and with the Channels Manager module for getting the channels with their events and actions.

#### 3.3.2.2 Rule Manager

Rule Manager is a controller, its aims are to manage the existing rules, to create new rules and to delete rules. 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, check their characteristics and import them. The user can also choose to display only the rules of a particular place.

Each time a user imports a rule or removes a rule, this module updates the repository with the number of users who have the rule imported. This way, it creates a list of the most-imported rules. The submodule displays a graphic interface to the administrator users with the list. This interface shares the space with other important data to the administrator. It can be used it in the future to make recommendations to the user.

This submodule connects with the Mobile App and with the Telegram Bot. Mobile App can create simple rules, i.e. not multi-event or multi-action rules, and Telegram Bot can manage the rules, it can see the imported rules, import rules and remove them. This module connects to the Rule Engine to send it the user's imported rules. In order to provide persistence, it needs a Rule Repository.

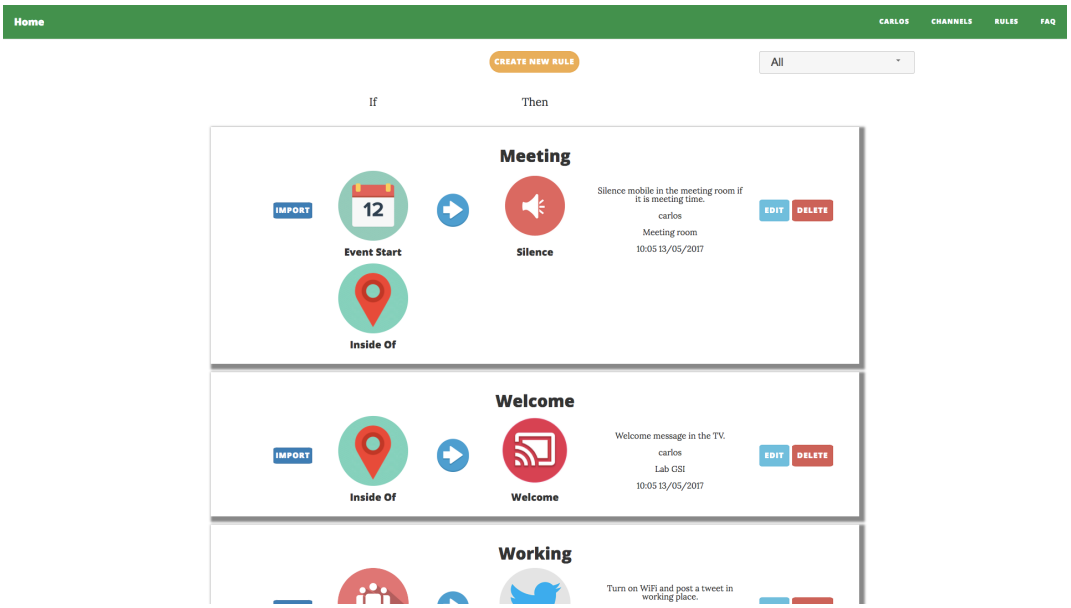


Figure 3.7: Rule Manager Interface

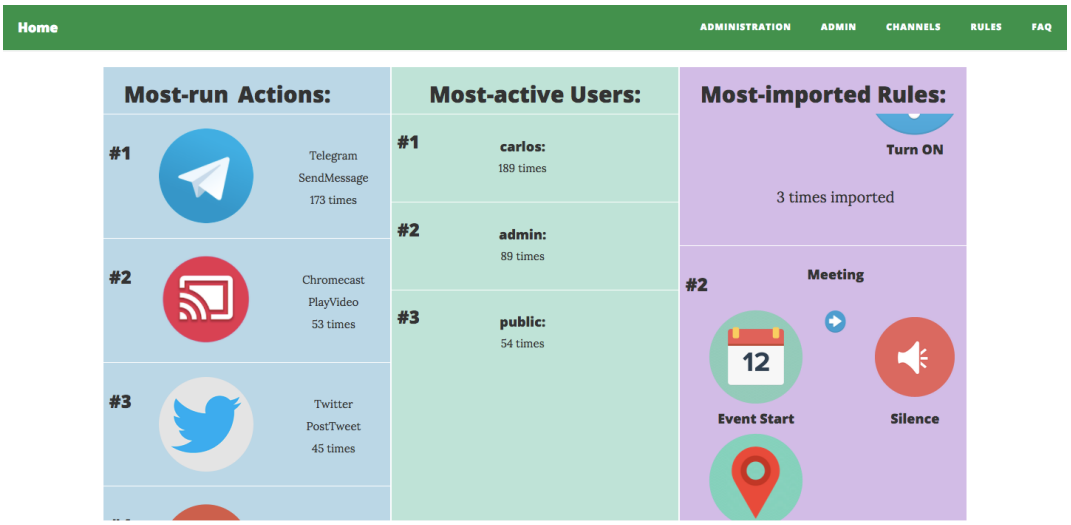


Figure 3.8: Platform Administration Interface

#### 3.3.2.3 Rule Repository

All rules are stored in a MongoDB database. As explained above, the Mongo databases store the data in JSON format and organize it in collections. Rule Repository has two collections: rules and most-imported rules.

### 3.3.3 Action Trigger

The task of this submodule is to execute the actions. It connects with the Events Manager and receives from it a JSON with the actions that has to run. This Action Trigger doesn't receive all actions because it can't execute some actions, so, Events Manager only sends the actions that this Action Trigger can perform. This submodule has limitations like it is not in the same subnet as some channels.

This Action Trigger can perform actions from channels like Twitter, which it was already implemented, and from new channels like Telegram.

As we have seen, TAS is the most important module of the project. It is very complex and it has several submodules. Nevertheless, there are another three modules.

To end this module, the support of **multi-event and multi-action rules** will be detailed. It is the most important feature of this project and deserves a mention aside. In each module, the principal peculiarities that allow the module to work with multi-event and multi-action rules have been explained. However, it has to be explained in detail.

Firstly, it is necessary to explain the operation of these rules:

- **Various events**: if the rule is "if event1, event2, and event3 are met, then executes the action1", then Events Manager has to send event1, event2, and event3 to the Rule Engine in the same request to satisfy the rule. Next, the Action Trigger executes the action1.
- **Various actions**: if the rule is "if event1 are met, then action1 and action2" and Events Manager sends event1 to the Rule Engine, then Rule Engine will response with action1 and action2 and Events Manager will send them to the Action Trigger.
- **Various events and various actions**: it is simply a combination of the previous options.

Next, it is necessary to explain the creation. As explained above, in the previous version, the rules had an event and an action. This project adds the opportunity of using various events and various actions per rule, without a determinate number of them, which complicates the creation of the rule.

- **The part of the events**. The events are defined with the variable "?event" and if all the events have the same variable in the rule, each input event will try to satisfy

the condition of every event, which is impossible because each event has individual conditions. It is necessary to set automatically a different variable per event. That is to say, it can't set a default variable, the variables have to be dynamic.

- **The part of the actions**. If the rule runs two actions that are performed by the same channel, it can generate a conflict with the parameters. Therefore, changing the form that the rules are implemented has been needed. Before, the parameter was only related to the name of the channel, so the parameter could accompany two different actions. Now, it uses the *providesAction* property of the EWE ontology that relates the parameter with the action. The name of the channel and the name of the actions are always related, so the platform can link the name of the channel, the name of the action and the parameter.

Finally, the storage of the rules will be explained. This is the easiest part, it only has to save the name of the rule, its description, its author, the name of all events' channels and actions' channels, the name of all events and actions and the rule in Notation3 format.

## 3.4 Telegram Bot

The Telegram Bot module is used like a manager of rules, like a performer of actions and like a generator of events. The three parts are connected in the **Contextual Chat**.

The server part is implemented in PHP and the Telegram APP part is implemented with The BotFather. The BotFather[1] helps to create Telegram Bots from commands that the user can write. Those commands usually start with "/". When the user runs a command that manages the rules, this module connects with **Rule Administration**. When the user runs a command that sends an event, this module connects with **Channel Administration**, specifically with **Events Manager**. The platform, through **Action Trigger**, can send a message to the chat with the bot.

### 3.4.1 Contextual Chat

The Contextual Chat submodule is the only means of communication between user and bot. The bot shows the information here and the user sends the commands from here.

Contextual Chat has a Channel Administration and a Rule Manager that connect with their counterparts in the TAS.

[1] https://telegram.me/BotFather

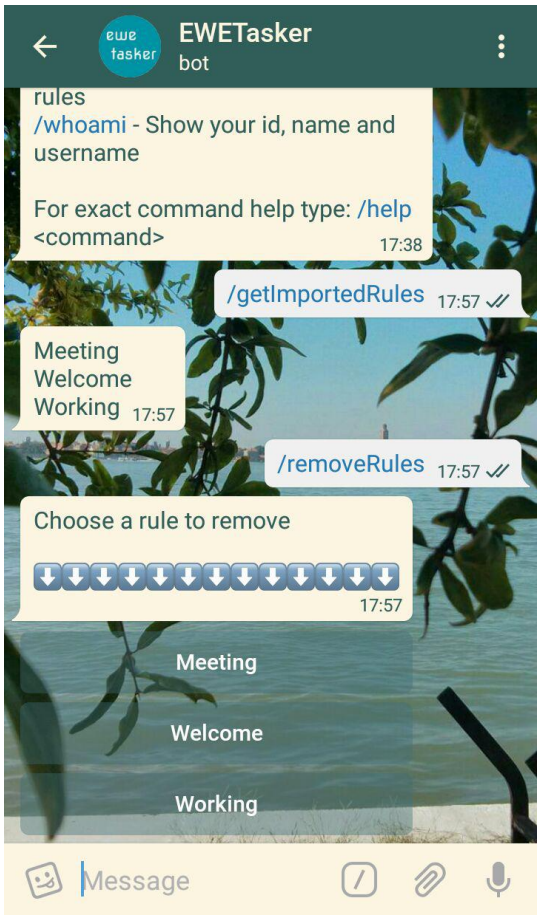


Figure 3.9: Contextual Chat Bot

#### 3.4.1.1 Channel Administration

Telegram Bot can generate an event using commands. There is a special command that is used for this purpose, this command is: "/eventCommand <text>". For example, a rule is: "if I send 'Turn Off Lights' from Telegram, the lights will turn off"; then, the user has to write in Telegram: "/eventCommand Turn Off Lights" to turn off the lights. Channel Administration of the bot reads the <text>, converts it to Notation3 and creates an event that the Rule Engine can understand. Finally, it sends it to the Events Manager. The event has the following structure:

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

ewe-telegram:Telegram rdf:type ewe-telegram:EventCommand.
ewe-telegram:Telegram ewe:text "Turn Off Lights".
```

Telegram Bot also can trigger an action. This action is to send a message to the user in the chat. For this purpose, it uses the command "/sendMessage <text>" that sends a message to the same user whose text is <text>. It is a hidden command, the users can use directly, but it does not appear in the documentation. When a user uses a command, he is sending a POST request to the server with data like the command, the text and the id of the chat. Action Trigger of the TAS sends a POST request with this data as if it was the user, so the user receives the message but actually, he has never sent the command.

#### 3.4.1.2 Rule Manager

Rule Manager can import and remove rules and see the imported rules. It connects with the Rule Manager of the TAS. It manages the rules using the following commands:

- **/help:** Show bot commands help.
- **/whoami:** Show your id, name, username, and your profile photo. It is important to know the chat id. To use this module, the user has to share this id with the platform.
- **/getImportedRules:** Show your imported rules. It is a simple list with the name of the rules that the user has imported.
- **/importRules:** Allow to import rules. The bot shows a list of buttons with the name of the rules that the user doesn't have imported. if the user pushes a button, the pushed rule will be imported.
- **/removeRules:** Allow to remove rules. The bot shows a list of buttons with the name of the rules that the user has imported. if the user pushes a button, the pushed rule will be removed.

## 3.5 Context Server

This module isn't a sole module. That is to say, there should be a Context Server in each smart place. However, all these servers have the same structure, features, and functions. It is a public server where some actions can be performed. Its main function is to be an intermediary between the TAS and some channels. For instance, if the users wants to play a video in a Chromecast device, the order has to be executed from the same subnet. In this case, Events Manager sends the action to the proper server and it runs the action. The submodule that runs the action is the Action Trigger.

### 3.5.1 Action Manager

This submodule has the same task that the Action Trigger of the TAS. However, it can execute different actions. Some of the channels with which it can connect are Chromecast, Hue Light, and Robot MiP.

## 3.6 Chromecast

As explained above, Chromecast is a Google device that enables the reproduction of media in a device with an HDMI input, like a TV.

In the project, it is used like a performer that plays videos in the TV. It has some default videos and the user can add new videos. The default videos are:

- **fire**: It is a relaxing video of a chimney.
- **space**: It is a video from the International Space Station that shows the Earth.
- **info**: It is a video with information about a particular place. This video is different in each place.
- **rain**: It is a relaxing video of rain over a window.
- **welcome**: This video can be used like a welcome when the user goes into a place. When this parameter is used, four different videos can be played and they are chosen randomly.

This action is performed by two Action Trigger modules. The TAS chooses if it sends the action to the Mobile App of to the Context Server, the election is based on the module that generates the event.

In the Mobile App, a new performer has been added and the Rule Execution Module has been changed. It is necessary to connect the mobile and the Chromecast device, so this feature has also been added. In the development of the performer, the official Google Chromecast SDK [11] has been used. Obviously, this has been implemented in Java.

In the Context Server, the performer has been implemented in Python with the *pychromecast* [12] library.

Both Action Triggers execute a similar code, these are the steps that the code follows:

1. Connects with the Chromecast.
2. Reads the parameter with the name of the video.
3. Chooses the video's URL.
4. Sends the URL to the device.
5. The device reads the URL, searches for the media and plays it.

In this chapter, the features and objects that this project has along with their communication and relationship has been explained.

The new features and improvements have been explained in this chapter. The main feature is the support of multi-events and multi-actions rules. For this purpose, several modules of the platform have been changed and new important modules have been added. Telegram Bot allows the user managing easily the rules from a chat, the Context Servers allow using channels that we couldn't use before and the Chromecast module allows that the Mobile App and the Context Server can connect to the Chromecast device. We have new generators of events and new performers of actions like Chromecast and Telegram. In the following chapter, the case study will be explained in detail.

CHAPTER 4

# Case Study

This chapter objective is to help understand the project main functionalities. For this purpose, we will go over the main use case to show the main features.

The actor of this case will be the user. It will be explained the process that the user has to follow to create the rule and the result of an accomplished rule.

## 4.1 Channel Creation

To begin with the creation of a rule, the first step is to choose the channels which will provide the events and the actions. The Creation Rule Interface shows to the user all the available channels. The user also can check them in the Channel Manager Interface. Sometimes, the necessary channel does not exist or it does not satisfy the requirements. In this case, the user can create a new channel pressing "Create New Channel" button of the Channel Manager Interface. Only the administrator users can edit an existing channel.

Channel Editor Interface shows different fields that the user must fill to create the channel. There are fixed fields, like the name, the description, the image and the "Nice Name" and, the fields with the events and actions. These fields don't appear at first, but the user can create them pushing a button, and he can create as fields with actions and

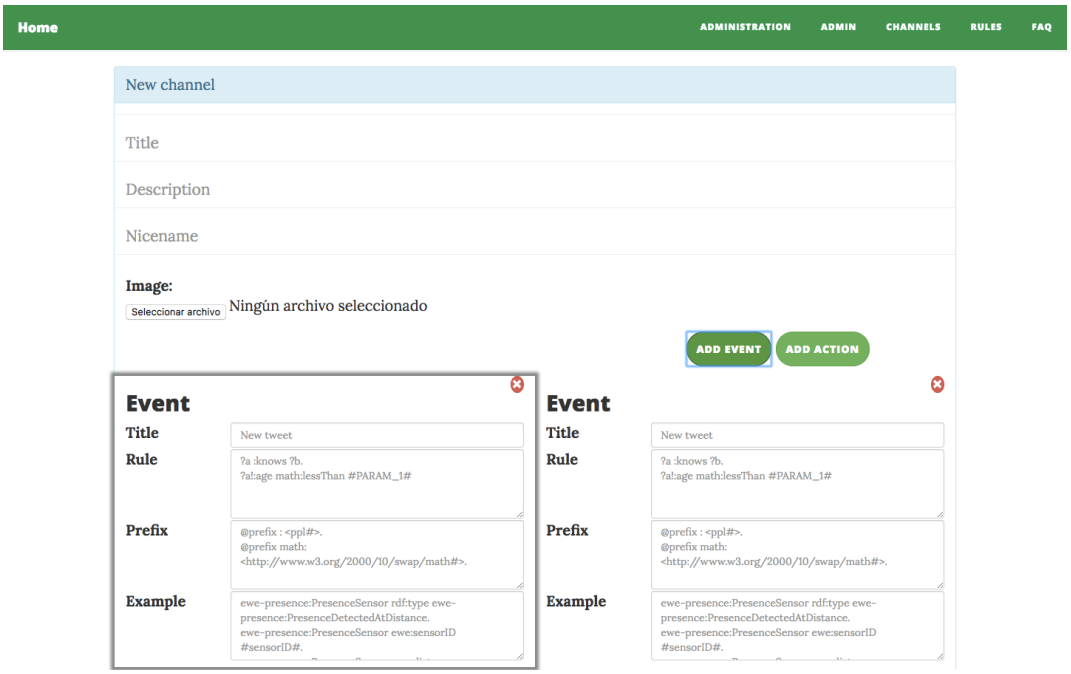


Figure 4.1: Creation of a Channel

events as he wants. In the event fields, the user must fill the title, rule, prefix, and example, and in the action fields, the user must fill the title, rule, and prefix. A channel needs to have all the fields filled, and at least, an event or an action.

Finally, the user has to push the "Send" button. The channel will be stored and any user will be able to use it in a rule.

The user can create the rule now.

## 4.2 Rule Creation

The user can create easily a rule from the rules page. He only has to press the "Create New Rule" button and he will redirect to the Rule Editor Interface.

In the Rule Editor Interface, the user has to fill some fields. Firstly, he must fill the name, description, and place. Secondly, he must fill the events and actions. The procure is to drag the icon of the channel and drop it in the correct container. Once the icon has been dropped, the user has to choose the event or the action of that channel. A dialog will appear with the options. If the event (or action) needs parameters, another dialog with fields to fill them will appear. After setting an event or action, a new event or action container will emerge. A rule can have infinite events and actions. If have various events, every event

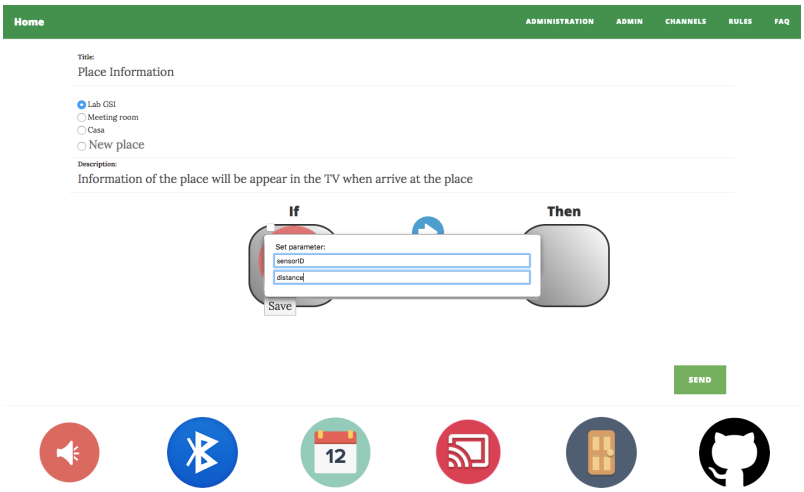


Figure 4.2: Set Parameters

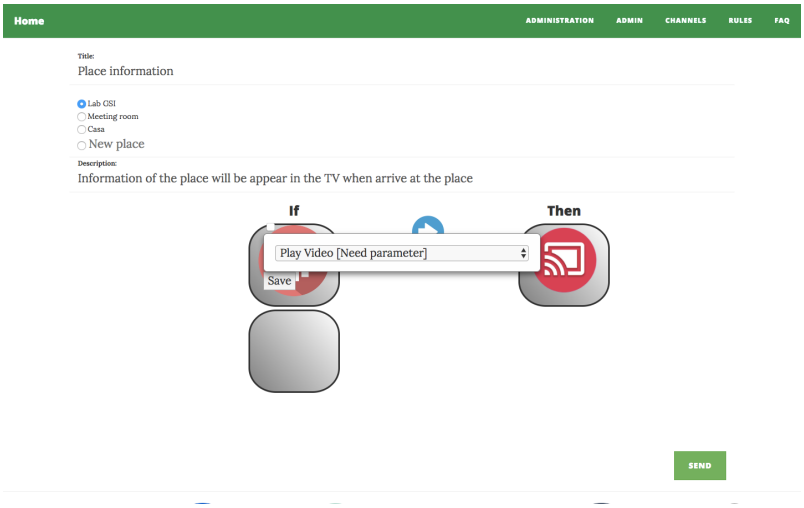


Figure 4.3: Action Selection

has to be generated to execute the actions. If have various actions, all the actions will be executed. Finally, the user must press "Send" button to store the rule.

## 4.3 Rule Import

Rule Engine only will evaluate the rules the user has imported. Therefore, to import a rule is very important. When the user creates a rule, it will be imported automatically, so the user has to do nothing. However, if the rule has been created by a third person, he will have to import it. The user has two ways to import a rule. From the rule page, he has to press

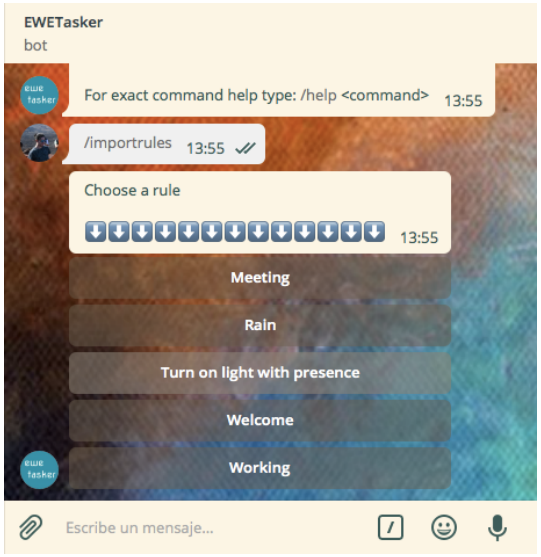


Figure 4.4: Telegram Import Rule

the "import" button. And from the Telegram Bot, he has to run the proper command and to select the rule.

In that moment, the user has two imported rules: *Place Information* (the rule the user has created) and *Turn on light with presence* (the rule the user has imported from Telegram Bot). In the following section, a real case with these rules is presented.

## 4.4 Case Study

In this case study, we are going to evaluate the operation of the rules previously described. The evaluation place will be the laboratory of the Intelligent Systems Group (GSI). In this test, the used items have been:

- Proximity sensor: 2 estimote beacons[1]: a beacon with ID "1a2b3c" in the main door and a beacon with ID "4d5e6f" in the lamp.
- A Raspberry Pi[2]. It is used as a Place Server.
- Place Server channels devices: Chromecast and Hue Light.
- Telegram Chat Bot.
- Smartphone.

[1] https://es.wikipedia.org/wiki/Beacon
[2] https://www.raspberrypi.org

In the example, we have assumed that all the required channels are already created and the user brings his smartphone with Bluetooth activated.

The rules the user is going to evaluate are:

- *Place Information*: It is a simple rule. When the user arrives at the laboratory of the GSI, the TV will play through Chromecast a video with information about this laboratory.
- *Turn on light with presence*: It is a multi-event rule. When the user is near the lamp and it's after nine o'clock in the afternoon, the lamp will turn on.

The steps the user realize to automate this process are the following:

1. Connect with Telegram Bot sharing the chat ID with the platform.
2. Create *Place Information* with Rule Editor Interface. The rule will be imported automatically.
3. Import *Turn on light with presence* with the Telegram Bot. The user has to write the "/importRules" command and select the rule in the list.

After these steps, the task automation process has been completed. When the user is near the main door of the laboratory, a video with information about this laboratory will be played on the TV. If it's after 21:00 and the user passes by the lamp, it will be turned on.

Every channel and every rule used in this case study are created with parameters explained in Appendix A.

In addition, to help the user understand and become familiar with rules and Notation3, a test tool has been added. The user can access it pressing the "Test EYE" button on the index page. The user is able to test the existing rules with the possible events. He also can write inputs and rules in Notation3.

In Figure 4.5, we see the rule *Turn on light with presence* executed. The user has to select the rule in the right column, the events in the left column and set the correct parameters. Then, he presses "Execute EYE" button and it shows the result. The run action appears in upper letters.

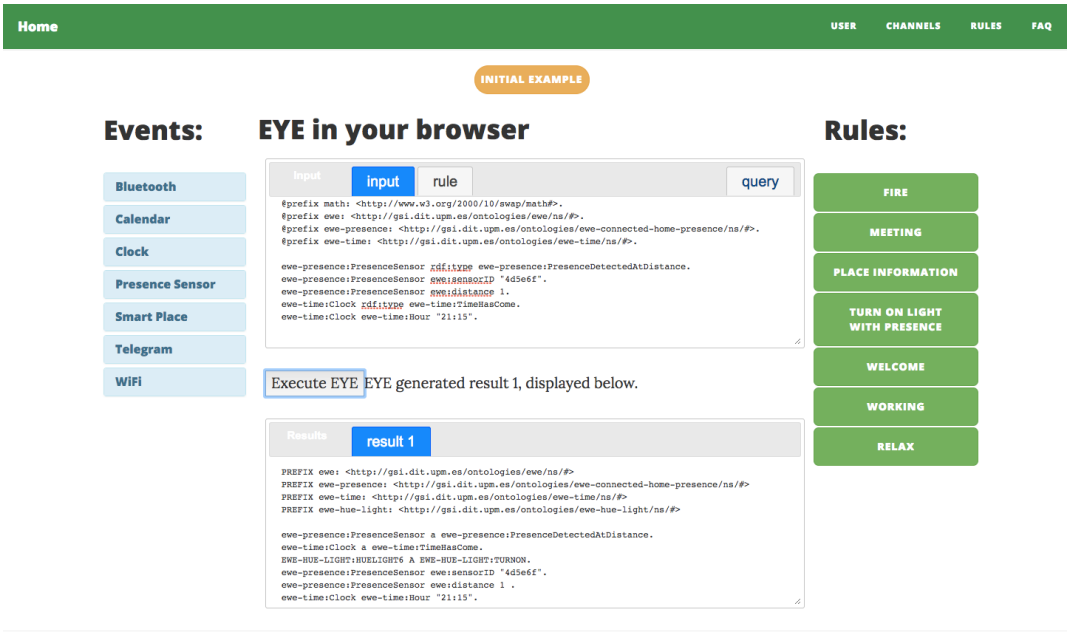


Figure 4.5: EYE Test

CHAPTER 5

# Conclusions and future work

In this chapter, we will gather the conclusions obtained as a result of the project, as well as possible future work that can be done for the further development of this project.

## 5.1 Conclusions

To conclude this project, it will be summarized the principal concepts that are explained in this document. We have developed a task automation platform that connects with other platforms and allows driving multi-event and multi-action rules. These rules are formed by channels that are modeled by the EWE [7] ontology. The channels have events and actions which are declared in Notation3 format.

Receivers and performers implement the events and the actions of the rules. These receivers and performers define the possible events and action that a channel might have. When the TAS receives a generate event, it connects with the EYE [5] reasoner and evaluates the rules.

It has also been developed an external module that allows managing easily the rules, the Telegram Chat Bot. Finally, implemented new receivers and performers have been. Telegram acts as a receiver and a performer, and Chromecast acts as a performer.

The project has been designed and developed with classic technologies as well as innovative technologies that may be uncommon to the standard user. It has allowed us learning new things. The technologies used on this project are listed below:

- Task Automation Manager:
  - Frontend: HTML, Javascript and CSS.
  - Backend: PHP.
  - Database: MongoDB.
  - Semantic Technologies: Notation3, RDF, EYE, EWE Ontology.
- Telegram Bot:
  - PHP.
- Mobile App Improvement:
  - Android.

This project has been based on two different projects:

- Development of a Task Automation Platform for Beacon enabled Smart Homes [8].
- Design and implementation of a Semantic Task Automation Rule Framework for Android Devices [9].

Next sections will describe what goals were achieved by this project, what problems were encountered and also future lines of work.

## 5.2 Achieved Goals

In the following section, I will explain the achieved features and goals that are available in this project.

- **Development of a task automation platform based on semantic multievent-driven rules.** This was the main objective of this project, to develop task automation platform that allows the creation of multi-event and multi-action rules and be able to evaluate them.

- **Development of a Telegram Bot.** The bot allows the user to manage the platform through a chat.
- **Connect with a server located in a smart place.** The platform is capable of connecting with any server that has an Action Trigger module and executes actions through this server.
- **Connect with new channels.** New channels have been defined in order to achieve our goals. The events and actions of these channels are described in Notation3. The Telegram and Chromecast channels are now available on the TAS to use for everyone.
- **Connect with a Chromecast device.** A new module that enables the connection with a Chromecast device has been developed.
- **Connect with Mobile App.** The platform is able to connect with EWETasker Mobile App to receive events and to trigger actions on it. In this project, changing the app has been necessary.

## 5.3 Problems Faced

The list of the problems encountered during the development of this project is shown bellow:

**Receive single events** The TAS receives events from a lot of different places, so the events come one by one. Then, the engine will never evaluate the multi-event rules. We have faced this problem creating a buffer with the input events.

**Subnet Limitations** Some channels only can be executed from the same subnet. The platform is on a server in some place and the devices that represent channels are in a different place. This was solved by launching a server where we need it.

## 5.4 Future work

In the following section, I will explain the possible new features or improvements that could be done to the project:

- **Improve bot**: more features to the Telegram Bot could be added. For example, the creation of rules.

- **Machine learning**: it could be implemented into the platform. The platform could suggest interesting rules to the user based on his rules, his location, the most imported rules or other characteristics.
- **Improve user profile**: The user profile is very simple. It only shows the created rules, the imported rules, and the button to connect with some internet services. It could be more complex.
- **New channels integration**: More channels open more possibilities.
- **New features Admin Administration**: The Admin Administration Interface is very simple, new features can be added. For example, the places where users evaluate more rules.

# Bibliography


[1] Carlos Ramos, Goreti Marreiros, Ricardo Santos, and Carlos Filipe Freitas. *Smart Offices and Intelligent Decision Rooms*, pages 851–855. pringer Science+Business Media, 2009.

[2] 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)*, pages 1–6. International, Sept 2013.

[3] M Coronado, C. A. Iglesias, and E. Serrano. Modelling rules for automating the Evented WEb by semantic technologies. *Expert Systems with Applications*, 42(21):7979 – 7990, 2015. [Online]. Available: `http://www.sciencedirect.com/science/article/pii/S0957417415004339`.

[4] 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. *Joint Proceedings of the 1st Joint International Workshop on Semantic Sensor Networks and Terra Cognita and the 4th International Workshop on Ordering and Reasoning*, 2015. [Online]. Available: `http://ceur-ws.org/Vol-1488/paper-08.pdf`.

[5] J. DeRoo. Euler yet another proof engine, 2013. `http://eulersharp.sourceforge.net`.

[6] R. Verborgh and J. De Roo. Drawing Conclusions from Linked Data on the Web: The EYE Reasoner. *IEEE Software*, 2015. [Online]. Available: `http://online.qmags.com/ISW0515?cid=3244717&eid=19361&pg=25#pg27`.

[7] M Coronado and C. A. Iglesias. Task Automation Services: Automation for the masses. *Internet Computing, IEEE*, PP(99):1 – 1, 2015.

[8] S. Muñoz. Development of a Task Automation Platform for Beacon enabled Smart Homes. Technical report, ETSI Telecomunicación, January 2016.

[9] A. F. Llamas. Design and implementation of a Semantic Task Automation Rule Framework for Android Devices. Technical report, ETSI Telecomunicación, January 2016.

[10] Longman. PHP Telegram Bot. `https://github.com/php-telegram-bot/core`.

[11] Google. Chromecast SDK. `https://developers.google.com/cast/`.

[12] Balloob. Python Chromecast Library. `https://github.com/balloob/pychromecast`.

APPENDIX A

# Rules and Channels Definition

In this appendix, we will present the definition of the rules and the channels that we have used in the case study.

## A.1 Channel Definition

- Channel Presence Sensor:
  - **Title**: presence.
  - **Description**: This channel represents a presence sensor.
  - **Nicename**: Presence Sensor.
  - **Event**:
    - Title: Presence Detected At Distance Less Than
    - Rule:

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

    - 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/#>.
```

* Example:

```
ewe-presence:PresenceSensor rdf:type ewe-presence:
    PresenceDetectedAtDistance.
ewe-presence:PresenceSensor ewe:sensorID "1a2b3c".
ewe-presence:PresenceSensor ewe:distance 1.
```

- Channel Clock:
  - **Title**: clock.
  - **Description**: This channel represents a clock.
  - **Nicename**: Clock.
  - **Event**:
    * Title: Time greater than
    * Rule:

```
?event rdf:type ewe-time:TimeHasCome.
?event ewe-time:Hour ?hour.
?hour string:greater #hour#.
```

    * Prefix:

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

    * Example:

```
ewe-time:Clock rdf:type ewe-time:TimeHasCome.
ewe-time:Clock ewe-time:Hour "15:00".
```

- Channel Chromecast:
  - **Title**: chromecast.
  - **Description**: It is a Chromecast device.
  - **Nicename**: Chromecast.

  - **Action**:
    - Title: Play Video
    - Rule:

```
ewe-chromecast:Chromecast ewe:providesAction ewe-chromecast:
    PlayVideo.
ewe-chromecast:PlayVideo ov:video #video#.
```

    - Prefix:

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

- Channel Hue Light:
  - **Title**: hueLight.
  - **Description**: It is a Philip's bulb.
  - **Nicename**: Hue Light.
  - **Action**:
    - Title: Turn on
    - Rule:

```
ewe-hue-light:HueLight rdf:type ewe-hue-light:TurnOn.
```

    - Prefix:

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

## A.2 Rule Definition

- Place information:
  - **Title**: Place information.
  - **Description**: Information of the place will be appear in the TV when arrive at the place.
  - **Place**: Lab GSI.
  - **Event channels**: ["presence"].
  - **Event titles**: ["Presence Detected At Distance Less Than"].

- **Action channels**: ["chromecast"].
- **Action titles**: ["Play Video"].
- **Rule:**

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

{
    ?event0 rdf:type ewe-presence:PresenceDetectedAtDistance.
    ?event0 ewe:sensorID ?sensorID0.
    ?sensorID0 string:equalIgnoringCase \"1a2b3c\".
    ?event0!ewe:distance math:lessThan \"2\".
}
=>
{
    ewe-chromecast:Chromecast ewe:providesAction ewe-chromecast:
        PlayVideo.
    ewe-chromecast:PlayVideo ov:video #video#.
}.
```

- Turn on light with presence:
  - **Title**: Turn on light with presence.
  - **Description**: It turns on the light when you arrives at Lab GSI after 21:00.
  - **Place**: Lab GSI.
  - **Event channels**: ["presence", "clock"].
  - **Event titles**: ["Presence Detected At Distance Less Than", "Time greater than"].
  - **Action channels**: ["hueLight"].
  - **Action titles**: ["Turn on"].
  - **Rule:**

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

{
    ?event0 rdf:type ewe-presence:PresenceDetectedAtDistance.
    ?event0 ewe:sensorID ?sensorID0.
    ?sensorID0 string:equalIgnoringCase "4d5e6f".
    ?event0!ewe:distance math:lessThan "2".
    ?event1 rdf:type ewe-time:TimeHasCome.
    ?event1 ewe-time:Hour ?hour1.
    ?hour1 string:greaterThan "21:00".
}
=>
{
    ewe-hue-light:HueLight rdf:type ewe-hue-light:TurnOn.
}.
```

APPENDIX B

# New Channels

In this appendix, we will present the definition of the new channels.

- Channel Chromecast:
  - **Title**: chromecast.
  - **Description**: It is a Chromecast device.
  - **Nicename**: Chromecast.
  - **Action**:
    - Title: Play Video
      - Rule:

```
ewe-chromecast:Chromecast ewe:providesAction ewe-
    chromecast:PlayVideo.
ewe-chromecast:PlayVideo ov:video #video#.
```

      - Prefix:

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

- Channel Telegram:
  - **Title**: hueLight.
  - **Description**: It is a Telegram's chat bot.
  - **Nicename**: Telegram.
  - **Event**:
    - Title: Event Command
    - Rule:

```
?event rdf:type ewe-telegram:EventCommand.
?event!ewe:text string:equalIgnoringCase #text#.
```

    - Prefix:

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

    - Example:

```
ewe-telegram:Telegram rdf:type ewe-telegram:EventCommand.
ewe-telegram:Telegram ewe:text "play Video".
```

  - **Action**:
    - Title: Send message
    - Rule:

```
ewe-telegram:Telegram ewe:providesAction ewe-telegram:
    SendMessage.
ewe-telegram:SendMessage ov:message #message#.
```

    - Prefix:

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