TRABAJO FIN DE GRADO

Título:	Diseño e implementación de un Framework de Automatización de Reglas Semánticas para dispositivos Android
Título (inglés):	Design and implementation of a Semantic Task Automation Rule Framework for Android Devices
Autor:	Antonio Fernández Llamas
Tutor:	Carlos A. Iglesias Fernández
Departamento:	Ingeniería de Sistemas Telemáticos

MIEMBROS DEL TRIBUNAL CALIFICADOR

Presidente:	Mercedes Garijo Ayestarán
Vocal:	Luis Bellido Triana
Secretario:	Carlos Ángel Iglesias Fernández
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

DESIGN AND IMPLEMENTATION OF A SEMANTIC TASK AUTOMATION RULE FRAMEWORK FOR ANDROID DEVICES

Antonio Fernández Llamas

Enero de 2016

Resumen

El concepto de Internet de las Cosas se basa en explicar cómo la conexión entre elementos físicos e Internet permite a dispositivos recoger y extraer datos, interactuando con el mundo real desde una perspectiva exterior. Su aplicación a las plataformas móviles puede ofrecer la posibilidad de definir, gestionar y analizar el comportamiento del usuario, automatizando tareas realizadas frecuentemente de forma sencilla e intuitiva.

Este proyecto se centra en diseñar e implementar un *framework* de automatización basado en la utilización de reglas con un patrón del tipo ECA (Evento - Condición - Acción), que permitan a los usuario automatizar acciones desde sus teléfonos móviles utilizando recursos internos o externos (Ej: Cambia el estado de mi teléfono de normal a vibración cuando empiece la reunión). Este *framework* se conectará con un motor de inferencias para ejecutar el proceso.

Este motor esta alojado en un servidor remoto, donde se almacenan y gestionan las reglas creadas. Cuando se lanza un evento, se envía una petición al servidor con los parámetros de entrada. Estos son procesados por el motor de inferencias EYE, el cual devuelve una respuesta. El servidor se encarga de filtrar y extraer la información deseada de esta respuesta y redirigirla de vuelta al smartphone.

Por último, se presentan las conclusiones extraídas tras la realización de este proyecto, así como las tecnologías utilizadas durante el desarrollo y los posibles aspectos a mejorar en un futuro.

Palabras clave: Automatización de Tareas, Android, EWE, SQLite, Motor de Inferencias, Java, EYE.

Abstract

The Internet of Things concept is based on how the connection of physical things to the Internet enable devices to collect and exchange data, interacting with real world from a distance. Its application to mobile platforms can provide the ability to define, manage and track user behaviour, automating daily frequently actions in a simple and intuitive way .

The project will focus on design and implement an automation framework based on the use of ECA (Event - Condition - Action) rules, that allow users to automate tasks in their phone based on internal or external events (i.e. switch ring mode from normal to vibration when I am in a meeting). The framework will connect to a remote rule inference engine for executing the inference process.

This engine will be hosted in a remote server, where the rules will be storaged and managed. When an event is triggered, a request will be sent to the server with the input parameters. This input will be processed by the EYE inference engine which will offer a response. The server will adapt this response filtering and extracting the desirable information generating the action that will be returned to the smartphone.

Finally, we will present the conclusions extracted from this project, the technologies we have used during the development and the possible lines of future work.

Keywords: Task Automation, Android, EWE, SQLite, Inference Engine, Java, EYE.

Agradecimientos

A mis padres por animarme a terminar esta carrera y apoyarme en todo momento y a mi tutor Carlos Ángel Iglesias Fernández por ayudarme con este proyecto.

Contents

List of Figures

CHAPTER 1

Introduction

1.1 Motivation

Since the beginning of the century, the Internet of Things concept and its impact in human daily life has become an innovative and important field of development in several countries. The relation between physical things and the Internet enable devices to collect and exchange data, interacting with the real world from distance. From the analysis of this data, some interesting conclusions about people behaviour and their habits can be extracted.

The evolution of mobile phones into smartphones has caused a *boom* in their usage, increasing their impact in society and offering a wide ensemble of possibilities. Nowadays, everybody has a smartphone and use it most of the day , which results in a huge dependency. Tons of innovative aplications have been developed with multiple perspectives, which have created a new market with new ways of interact and contact with people through different platforms.

The idea of implementing technology in familiar devices has come up recently with a clear objective, choose a technological solution to automate common activities that people accomplish frecuently in their lifes. The smartphone is hardly attached with this idea and might become precursor of it. A device that stays most of the time in the pocket of your

jeans may contribute valuable information to affiliate customers in some enterprises.

Even though people use the smartphone every day, many of them don't know how to optimize it resources. There are many processes like scheduling a meeting everyday at nine o'clock, or turning off the WiFi connection and enabling data network when I leave home that are repeated frecuenly. The automation of these tasks might be a smart solution for the waste of time and battery power consumption. Therefore, the concept of automating tasks depending on position, or specific events triggering sounds really attractive to people familiarized with the Smart Homes sector.

Inside this scenario, some new companies are emerging as task automation platforms on mobile devices. Some examples of these are IFTTT [3] or Atooma [4]. These aplications give the user the opportunity to define rules using local resources or third-party aplications like social networks or calendar, even gadgets or external devices with specific uses like beacons. The rule structure is really simple and the creation process is intuitive and accesible. Some rule examples could be *If I start a meeting sctive silent mode on my smartphone* or *If my battery has low charge level then decrease the screen brightness*. This simplicity makes the user feel comfortable and free to create and customize the rules he actually needs.

The aplication that we present in this project feeds on these ideas. All these innovative projects try to make advantadge of the imminent automated tasks arrival, where every event accomplished through mobile phones (E.x. search on Google, send an email, upload a picture...) can generate an action performance, considering statments with a Event - Condition - Action (ECA) structure for rule automation. This approach opens hundreds of possibilities, where the users can combine different events and actions in multiple ways to solve their problems.

To conclude, the global overview of this project relies in the implementation of a complete independent framework which enables the channel adition, the rule definition and the rule execution from the smartphone, using local or remote resources as channels for create rules. This can be useful for automating processes that the user executes frecuently and setting up rules to some geographic locations, allowing to manage these decisions in order to create an intelligent device capable of acting itself and learning from his owner. For this we will connect to a remote rule inference engine, which hosted the automated server, for executing the inference process.

1.2 Project Goals

The main purpose of this project is to create a scalable Android aplication for creating rules and executing them. This aplication must be able to connect with the remote server, where the channels will be described. This server will act as an intermediary between the aplication and the rule inference engine. The aplication will have to listen to the local resources in case one event is triggered, and communicates with the server, where the response will be generated with the action that has to be performed.

Among the main goals of this project we can highlight:

- Design and implement an Android aplication which enables the rule definition using the list of local and remote channels hosted in the server.
- Implement a local service that handles the communication with the server and allows the channel download and rule management.
- Develop the receivers and performers of the mobile local resources and implement their events and actions.
- Build a rule execution module that uses the EWE ontology to describe every channel and notices the server when an event happens.
- Build a response interpreter for the EYE engine response sent by the server.

1.3 Structure of this document

In this section we provide a brief overview of the chapters included in this document. The project is divided into the following sections:

Chapter 1 is an introduction of the project where the motivation and the main goals and achievements of this project 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 aglobal overview followed by the explanation of all the modules of the system. It finalizes with the description of the Android aplication itself.

Chapter 4 offers an overview of a selected use case, explaining the main functionalities to help the user understand the overall concept.

Chapter 5 sums up the conclusions extracted from the realization of this project with a brief future perspective.

CHAPTER 2

Enabling Technologies

2.1 Introduction

In this chapter we will present in depth the main technologies and resources used in this project. First of all, we will analyze the structure of the IFTTT[3] ecosystem, explaining their different usages and how third party applications can be connected to it. Secondly, we introduce some emerging applications that have come up recently and are related to task automation services on mobile devices. Finally, we present how semantic reasoning works with EYE engine and it behaviour inside the Android SDK framework.

2.2 IFTTT

IFTTT (*"If This Then That"*) [3] is a web-based service that allows users to create conditional statements, which are triggered based on the occurrence of particular local events (E.g. Mute your phone, enable GPS receiver...) or changes to other web services applications such as Facebook, Twitter or Gmail (E.g. Send a tweet, make an appointment in my Google Calendar profile, recieve an email...).

If we analize in detail IFTTT's platform, we can work out thousands of rule combina-

tions that can automate and solve daily situations in a smart way. This allows us working efficiently, training mobile phones to make their own decissions and learn from their bearer.

Nowadays, IFTTT platform is used and shared by hundred of its users, sorting similar rules by categories and assembling packages for specific situations.For example, rules oriented to keep track of your health and fitness activities (*If I enter the gym - Then record my session on a Google Doc*), even using third-party applications with similar purposes (*If I dont meet my Fitbit daily goals - Then send me a notification message to my email addres*). The huge growth that this trendy applications are experiencing comes from three success factors [5].

1.- Usability: These applications provide an intuitive interface for programming task automations, therefore users experiment no learning curve when they begin using them.

2.- Customisability: These task automation services allow users to program the rules they need, which cause a sense of freedom and innovation, providing a total control over rule definition.

3.- Integration: The integration between Internet applications and task automation frameworks makes possible the access to services and platforms where the user feels comfortable and familiar, being accustomed to using them everyday.

2.2.1 Architecture

Every single IFTTT rule follows the same design pattern. This pattern has the following structure: If *EVENT* accomplish *CONDITION*, Then perform *ACTION*. Attending to this structure we can differenciate several components:

- Event: It is the preamble of the rule. It describes the action that should happen to execute the rule. The principal component of the event is the channel, which is the actor who performs the corresponding action. For example, one channel could be the Bluetooth module of a smartphone. Every channel have different triggers and performig actions, like turn on and turn off, find a device entering the Bluetooth starring radio or a device attempting to contact us with a specific SSID.

- Condition: This element indicates what should happen to perform the action. If satisfied or evaluated to true, causes the action to be carried out.

- Action: It is the update or invocation on the receiver agent. Analyzing the previous example, the action will correspond to *send a notification message via email*. Actions could be carried out by external applications, like social networks applications or

Gmail, or internal components like Android toast service or smartphone connectivity modules such as GPS or WiFi.

2.3 Atooma

In the last section we have explained how IFTTT enviroment works and the evolution it has experienced in recent years. Some development groups have take advantage of this emerging technology, building up some applications and projects related to this field. Atooma [4] is one of the most important Android application in the market for task automation service following the IFTTT model.

What makes Atooma different is the simplicity of its user interface. You simply have to program the rule and tell the app what to do in the case of a certain event. This allows to perform automatically any kind of tasks on your smartphone.

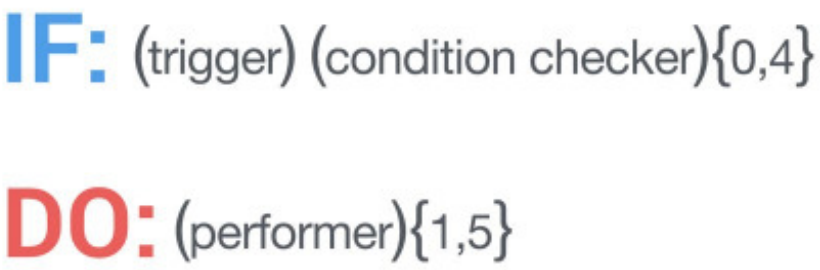

Figure 2.1: Atooma's module structure based on IF - DO aproach

Attending to figure 2.1, we can differenciate several components in module definition structure. In the Atooma framework SDK, the main element is the Module object, which is the component that will perform the Event - Condition - Action (ECA) statement. Some examples could be the SMS controller or the battery level listener, which controls the power consumption. Modules are composed by three different set of components: Triggers, Condition Checkers and Performers. [6]

2.3.1 Modules

Modules are the components that will perform the automated rule statement. Atooma assembles these modules in different categories or subclasses, depending on their architecture and purpose. We can differenciate different modules (Figure 2.2)

- Mobile: In this group we can find local modules physically installed in the smartphone. For example, battery, screen, GPS, WiFi, light sensor, Internet, SMS ...
- Apps: This ensemble is composed by third-party applications such as Twitter, Facebook, Google Drive, Calendar, Dropbox...
- Objects: This section allows the user to write rules using external devices as modules. NFC or Smart Watch are some of the examples that we can find in this mixture.
- Plugins: This section regroups plugins developed by third persons using the Atooma Plugin SDK.
- Files: Finally, this group is formed by files stored inside the smartphone memory card, like photos, videos, audio files o private folders.

Figure 2.2: Atooma modules: Mobile, Apps, Objects, Plugins and Files

2.3.2 Triggers

Triggers are components responsible for rules activation when a specific event occurs. Some examples are *WIFI ENABLED* (Figure 2.3) or *SMS INCOMING*. There are two different types of triggers:

- Intent based triggers: The activation of intent based triggers relies on the reception of an specific intent from Android system. The WiFi activation or SMS reception belong to this section.

- Alarm based triggers: The activation on alarm based triggers relies on periodic checks of specific conditions. The Alarm module or events related to scheduled situations belong to this section.

Figure 2.3: WiFi triggers

2.3.3 Condition Checkers

Condition checkers are used by a module for allowing it to check a condition when they are invoked, and return a boolean outcome. It is possible that some condition checkers require specific parameters to decide the right answer. One condition checker example could be the *WIFI CONNECTED*, which allows us to control if WiFi is enabled and connected (Figure 2.3). This condition checker requires an optional parameter SSID (Unique Identifier), that reports the identifier of the WiFi network to check (Figure 2.4). By default, connection to any network is positively evaluated. Triggers and Condition checkers usually conform de IF part of the rule statement.

Figure 2.4: WiFi Connected condition checker. Requests an optional parameter SSID.

2.3.4 Performers

Performers are components responsible of executing operations on demand, as requested by rules they are declared in. As condition checkers, they can have optional parameters attached and can produce optinal variables in output. The most visible performer is *NOTIFICATION TOAST*, where the performer shows a message to the user via Android toast module. It requires a text parameter with the string sentence to be displayed (Figure 2.5.

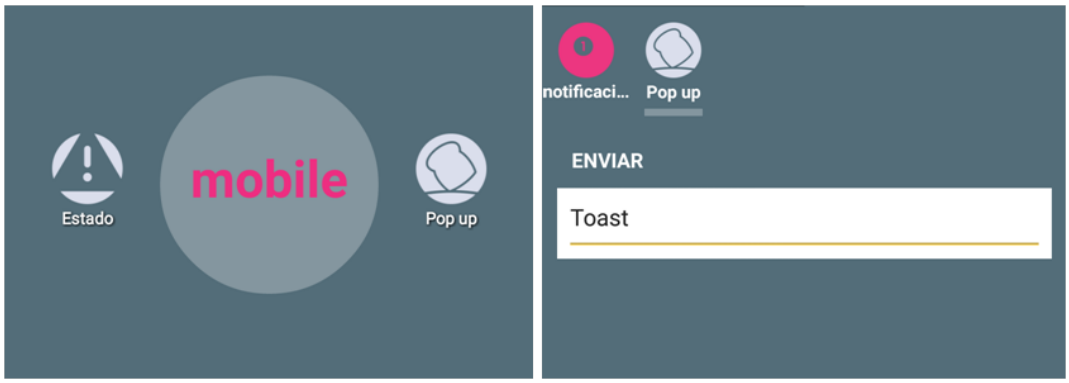

Figure 2.5: Notification Toast performer. Requests an optional String type paramenter.

2.4 EYE

EYE [7] stands for "Euler Yet another proof Engine" and it is a further delevopment of Euler which is an inference engine supporting logic based proofs. EYE is a semibackward reasoner

enhanced with Euler path detection, Semibackward reasoning is backward reasoning for EYE components, i.e. rules using <= in N3 and forward reasoning for rules using => in N3.

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.

During the last years, EYE [8] has been tackling such reasoning challenges on a large scale, thereby forming a part of the bigger Semantic Web vision. In this vision, machines perform tasks on the Web for people, combining linked data with concepts such as reasoning and proof to turn data into knowledge and concrete actions.

Let's see how EYE moves from data to conclusions. We can describe the *knows* property with existing concepts such as *domain*, *range* and *SymetricProperty*. Many reasoners have built-in knowledge; they would know what those concepts mean and how to apply them. The drawback is that only built-in concepts can create new knowledge. So, EYE was designed to have the least amount of inherent knowledge. It is extensible through rules so that new knowledge can be created. Here is an example of how SymetricProperty definition works:

```
{
?property rdf:type owl:SymetricProperty
?A ?property ?B
}
=>
{
?B ?property ?B
}
```

Figure 2.6: Example of how SymetricProperty definition works

2.4.1 Architecture

Algorithmically speaking, EYE is a theorem prover. Users set a goal and EYE tries to reach it by applying logical rules similar to what we mentioned previously, mostly working backward from the final goal. To evade endless loops, the algorithm avoids needlessly repeating previous work through Euler path detection. EYE interpretes each logical rule $P => C$,

where P is a precondition and C is a consequent, as P *and* NOT C => C, so that rules execute only when they can generate new triples.

A key characteristic of EYE's architecture is portability, because interoperability is crucial to the Semantic Web. EYE runs in a Prolog virtual machine, which is compiled to assembly code ans runs directly on the CPU. The core of EYE (the *Euler Abstract Machine*) accepts N3 (*Notation3*) code, which is a Prolog representation resulting from parsing RDF (**Resource Description Framework**) triples and N3 rules. This entire stack becomes a generic reasoning engine, which is extensible with any kind of domain specific rules. Because EYE can output N3 code to a file, you can create reasoning-engine instances that have a certain rule set preloaded for a particular domain.

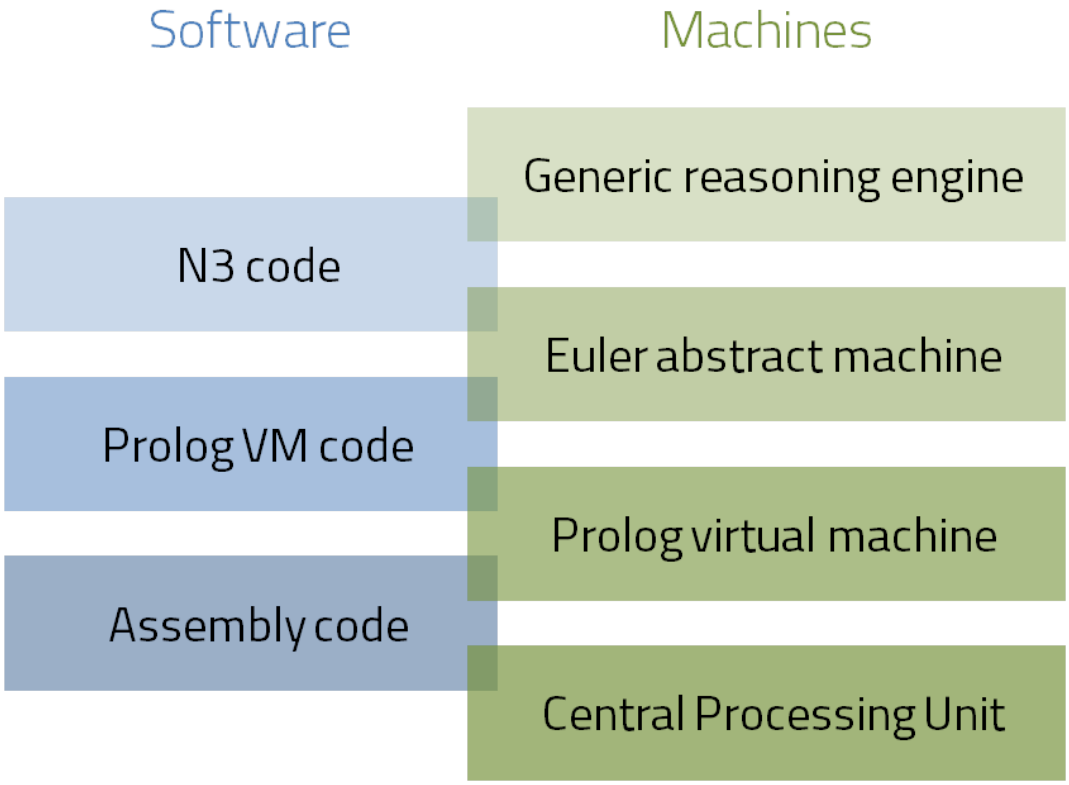

Figure 2.7: The EYE stack, which offers a generic reasoning engine.

EYE also exhibits external compatibility, accepting exchangeable N3 and RDF documents from any source on the Web. Users can ask EYE to generate a proof explaining how the given goal is reached. Such proofs use a publicly available , interoperable vocabulary, so other parties can follow and understand the line of reasoning.

Most important, this mechanism allows independnt proof validation, which contributes to one of the forms of trust on the Semantic Web. If a certain party comes to a conclusion,

any other party can thus verify why taht conclusion is valid.

2.4.2 EWE

In this section, we will present the EWE (Evented WEb) ontology [1] [9], which models the most important aspects os task automation services from a descriptive approach, that enables service discovery and semantic rule definition featuring reasoning over LOD (*Linked Open Data*). The goals of the EWE ontology to achieve are:

- Enable to publish raw data from TASs online, and in compliance with current and future Internet treds.
- Enable rule interoperability
- Provide a base vocabulary for building domain specific vocabularies like Twitter Task Ontology or Bluetooth Task Ontology.

2.4.3 Design methodology

The final model we propose is the result of an iterative development process consisting on three steps. First of all,the analysis of each TAS considered, identifying features and functionalities later we can use to extract the concepts ans properties they address. Secondly, the definition of a model that formally describes those elements, and finally, the evaluation of the model against the different use cases considered.

After each iteration, the proccess is repeated, including some new elements that the results have shown to be relevant or important, and remodelling others in order to improve the domain description.

2.4.4 Elements

2.4.4.1 Main classes

The core of the ontology comprises four major classes: Channel, Event, Action and Rule. The description of particuar TASs or use case scenarios may inherit from them, creating new sub-classes that are more specific to the domain. This ontology model is described in 2.8

- Channel: this class defines individuals that either generate Events, provide Actions, or both. In the context we refer to, a Channel implementation usually defines the behaviour of an specific smartphone component (i.e Buetooth module, Light sensor, Calendar application...). It can also define Web services implementations like Twitter or Facebook. This channels generate Events every time a tweet is posted, every time a scheduled meeting starts through Calendar application etc, and Actions like show a text message pop up, disable GPS connectivity or switch your mobile to Vibrate mode.

- Event: This class defines a particular occurrence of a process, which may trigger rules in the TAS. They are instantaneous, which means it happens without duration over time. For instance, the class *NewEventScheduled* is subclass of Event and defines the type of event that is generated when a new event has been scheduled in the Calendar application. This event can carry information of the event name or when it has been triggered.

- Action: This class defines an operation or process provided by a Channel. Actions produce effects whose nature depends on the action nature. In this category we can include facts like turn off the WiFi receiver, decrease the screen light level in order to reduce power consumption or show a custom notification. These actions can be triggered by an event generation in the same Channel or a different one. In the same way as Events behaviour, the Action class may be subclassed to specific actions. Nevertheless, an Action is not attached to a specific Channel, it can be supported by different ones.

- Rule: Finally, the Rule class defines an *Event-Condition-Action* (ECA) rule, triggered by an Event that produces thee execution of an Action. Rules define particular interconnections between instances of the Event and Action classes transferring information from the event to the action. EWE rules can be fully described as EYE construct queries, being executed in theremote server by the EYE inference engine.

2.4.4.2 Main properties

Each class described in the previous section can have attached several properties or attributes. Coming up next we will list some of this properties and explain the purpouse of including them within the overall EWE ontology [1]. These attributes are *hasParameter*, *hasCategory*, *hasActiveChannel*, *hasCreator* and *spin:rule*.

The property *hasParameter* presents the parameters of an Event or Action. For instance,

an event triggered when a user enters a specific area must be declared the latitude, longitude and radius parameters that defines the scene of action. 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 aplications like Twitter, Facebook, LinkedIn etc. The *hasActiveChannel* property links users to the Channel on which they have an account. The property *hasCreator* links instances of Rule to its creator, allowing to associate an author to every single rule statement. To sum up, the *spin:rule* property which links rule instances with the SPARQL execution logic described using the spin vocabulary. In this thesis we will not use this module because we will focus on the EYE usage.

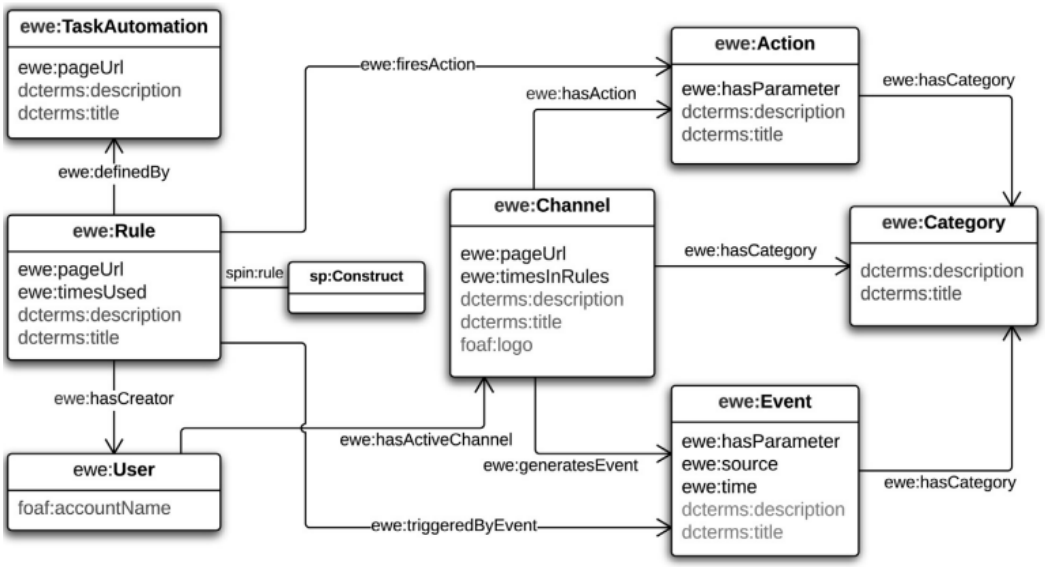

Figure 2.8: Detail of the EWE Ontology model. [1]

CHAPTER 3

Architecture

3.1 Introduction

In this chapter we will explain the architecture of the project, including a global overview about how the different elements interact and communicate each other, and showing how they have been implemented. First of all, we will present a global vision about the project architecture, identifying and explaining the principal server and client modules. Secondly we will focus on the task automation application, starting with the Rule Definition module and his internal activity sequence. To sum up, we will explain the Rule Execution module, explaining how the Android receivers listen when an event is triggered, and how the application communicates itslef with the remote server, analyzing the response and performing the right action.

3.2 Overview

In this section we will present the global architecture of the project, defining the diferent modules that participate in the Rule Task Automation service. We can divide the modules in two sub-groups: Client and Server [2], linked each other between an API service. Attending

to the 3.1, we can notice several parts which build the project architecture.

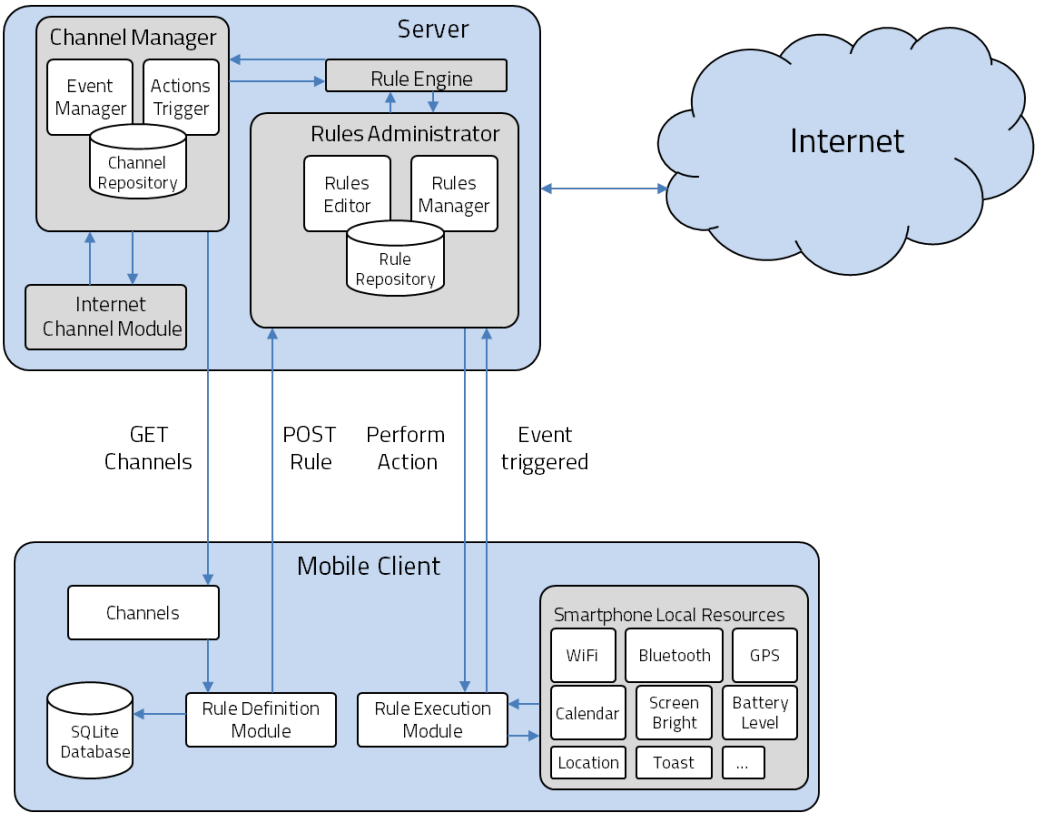

Figure 3.1: Client - Server architecture overview.

3.2.1 Server Modules

The server [2] is formed by different modules that communicate each other and interact with the client application. The main functionalities that the server gives to the user are the possibility of save, edit and delete custom rules, enable direct contact between EYE web reasoner and the final user, and wrap the response (the action that will be performed) from any input (the triggered event),what offers a transparent service for the client part.

- **Channel Manager**: This module provides an automation channel editor. Nevertheless, it also handle 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. 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 module or to the Mobile or Google Glass App, depending on its nature. It is divided into four main parts: Channel Editor, Channel Manager, Channel

Repository and Events Manager.

- **Rules Administrator**: 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, Rules Editor, Rules Manager and Rule Repository.
- **Rule Engine**: Rule Engine is one of the most important modules in this project, and is based in an ontology model, which uses the EWE ontology. It is divided into two parts: EYE Server and EYE Helper. EYE Server is implemented in Javascript. It is the reasoner that processes the events and rules written in Notation3 and generates accordingly a response with an action. 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 has the events and the rules, it sends them to the EYE Server for being processed and receives the response.

3.2.2 Client Modules

In this case, the client is basically the mobile aplication, but it can also be a web page or an aplication developed for another enviroment or framework. In this project, the client is a mobile application developed for Android devices, which is composed by a Rule Definition module, which allows the user to create his own IF - DO statements and store them locally and in the remote server database, and a Rule Execution module, where the smartphone can trigger different events notified to the server, and perform actions declared in rules. These actions are always performed by mobile channels, which means that third-party app channels will not be included inside the application because the server will handle them directly. Attending to figure 3.1, we can observe these client side modules:

- **Rule Definition**: Composed by a service that enables to comunicate with the server, sending the *Event* - *Condition* - *Action* (ECA) selected by the user over the rule definition sequence. During the selection process, the user must fill every rule property and define their parameter if it is required. The Rule Definition module will also save the rule locally in order to minimize the number of requests per minute sent to the server.
- **Rule Execution**: This module refers to the process of listening and execution of the event triggers, communicating with the server, where at the same time calls the EYE

engine with the corresponding input statement, obtaining a response. This response can be irrelevant or either represent the performance of an action, in which case the server will filter and adapt the output, modelling and simplifying it. This embebed sentence is sent to the user who preset the rule, performing the local channel action.

- **SQLite Database** [10]: This module represents the smartphone local database, where the rules hosted by the server are saved.

- **Smartphone Local Resources**: This module groups together all the local resources that will act in de rule creation process as Channel. Each resource has events and actions.

3.3 Modules

3.3.1 Rule Definition Module

As it is explained in the previous section, this module main function is the rule definition, starting with the channel selection and continuing with the selection of the event and action. After this, an N3 statement is generated from this choices, describing the rule and sending it to the server, where it is saved.

3.3.1.1 Channel Information

First of all, the mobile aplication needs to request the channel list. The automation server answers with a string chain in JSON format containing the channel with their particular events and actions. The aplication transforms this information and generates a *List <Channel>* object with all necessary channels for set up the rules.

This channel request process is described in Figure 3.3. We can appreciate how the client request the channels, receiving the automation server response and stocking it locally in a SQLite [10] database.

The JSON structure is showed below. It is composed by an array of channels with their own attributes. At the same time, the events ans actions of the current channel are described, being grouped in a list:

```
[{
        "title":"",
        "description":"",
        "events":[{
                        "title":"",
                        "prefix":"",
                        "rule":"",
                        "numParameters":""
                                        }],
        "actions":[{
                        "title":"",
                        "prefix":"",
                        "rule":"",
                        "numParameters":""
                                    }]
}]
```

Figure 3.2: Get Channels JSON structure

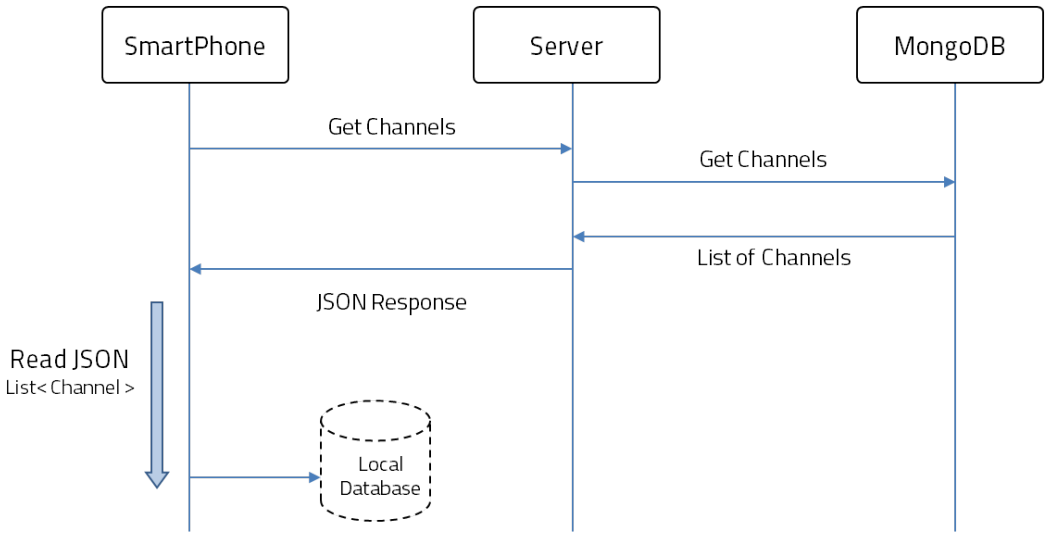

Figure 3.3: Sequence diagram of channel obtaining

3.3.1.2 Database

In the Rule Definition Module persistence layer we have implemented a local database to store the downloaded channels and the generated rules. We stock the channels in a *List <Channel>*, according to the Figure 3.4 channel model. These channels are downloaded every time the application is launched so it is not necessary to store them for later sessions. For the rule storage we have chosen an SQLite [10] database because its easier to manage

the item addition and removal. It has been implemented a *SQLiteDatabaseHelper* which facilitates the inout communication.

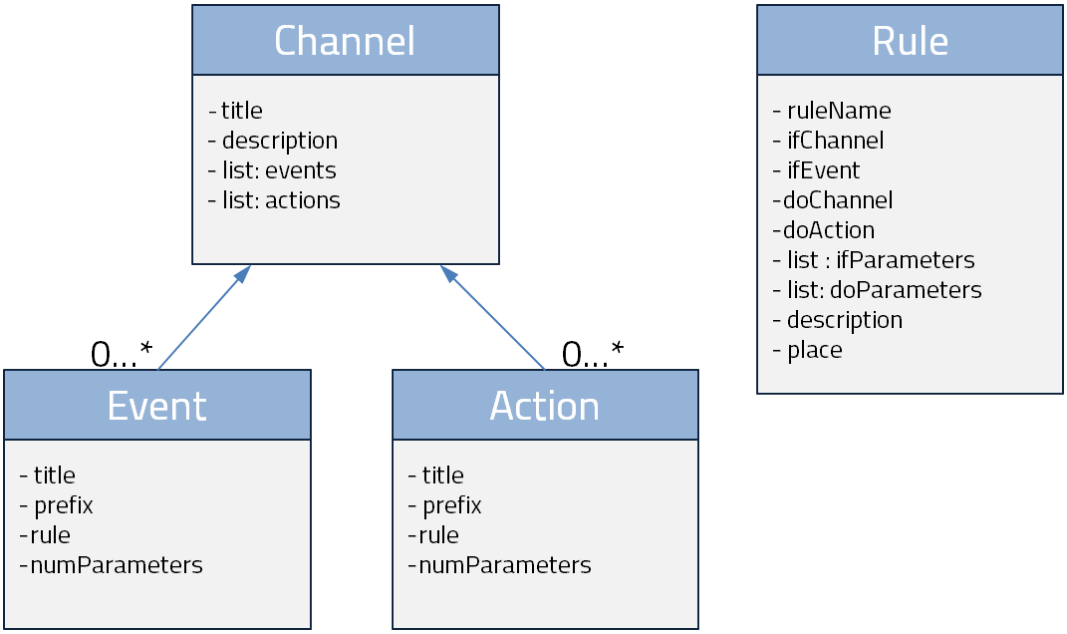

Figure 3.4: Database model

A channel has descriptive attributes like title or description because the server has to be able to distinguish between channels with similar events and actions. For example the Notification module can belong to the smartphone or either to an email or alert reminder. Moreover, the channel has a list of events and actions attached to them, which can trigger or perform respectively.

Each rule object is composed by contextual attributes (title, description and place), and main attributes that define the rule behaviour (ifChannel, ifEvent, doChannel, doAction and the array parameters for IF and DO clauses).

3.3.1.3 Creation Process

In this section we will introduce the rule creation process and explain the steps that the application has to follow to set up a rule from the channel selection to the remote server storage. Every decision the aplication makes will be sent to the Rule Definition service, where it will be processed and packaged for the server delivery.

In Figure 3.5 is explained how this sequence works. First of all, the aplication gets the channels we presented in section 3.3.1.1. Secondly, the user chooses the channel that will trigger the rule execution. Afterwards, the user has to select what event of the channel

selected will be the trigger. If the event selected has parameters, the activity will request it. Once the IF elements are chosen, the aplication sends this information to the local service, filling the corresponding parameters.

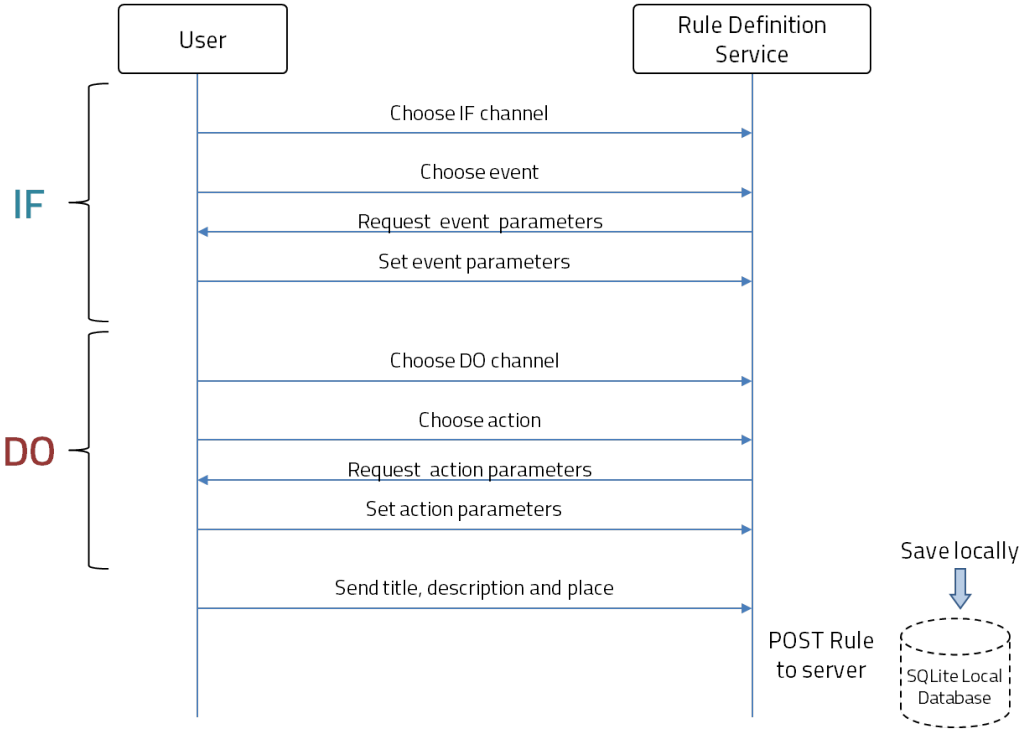

Figure 3.5: Description of the Rule Creation process in sequence diagram

The second part consists on DO part configuration. The aplication requests the channel which will perform the action. In the same way, the user will have to specify the action that will perform the rule, asking for parameters if necessary. Finally, these data are sent to the service completing the main rule attributes.

By last, the aplication asks for complementary parameters like the rule name, description or the place where it will be executed. With all these parameters, the app will launch an asyncronous task, generating a POST request to the server with all these parameters. At the same time, the Database Helper will save this rule locally adding one new row to the Rules table. With the latter, the rule definition process concludes.

3.3.2 Rule Execution Module

In this section we will explain the proccess that occurs when an event is triggered and how the aplication handles the server response performing the corresponding preset action. This module is always listening any changes on local channels, constantly tracking events to the

server, mostly without any consequence. In addition, the execution module acts as the brain of the aplication, coordinating the event triggering and action performance.

3.3.2.1 Execution Process

The aplication has several listeners that notify any modification in the channel events. These listeners behave as broadcast receivers when they notice the aplication, starting a process which involves client and server. All the receivers have the same structure, set by the number of events the channel have. When an event is triggered, a request is created and sent to the Rule Execution module, where it analyses what channel and event have been triggered and sends this input to the remote server in N3 format.

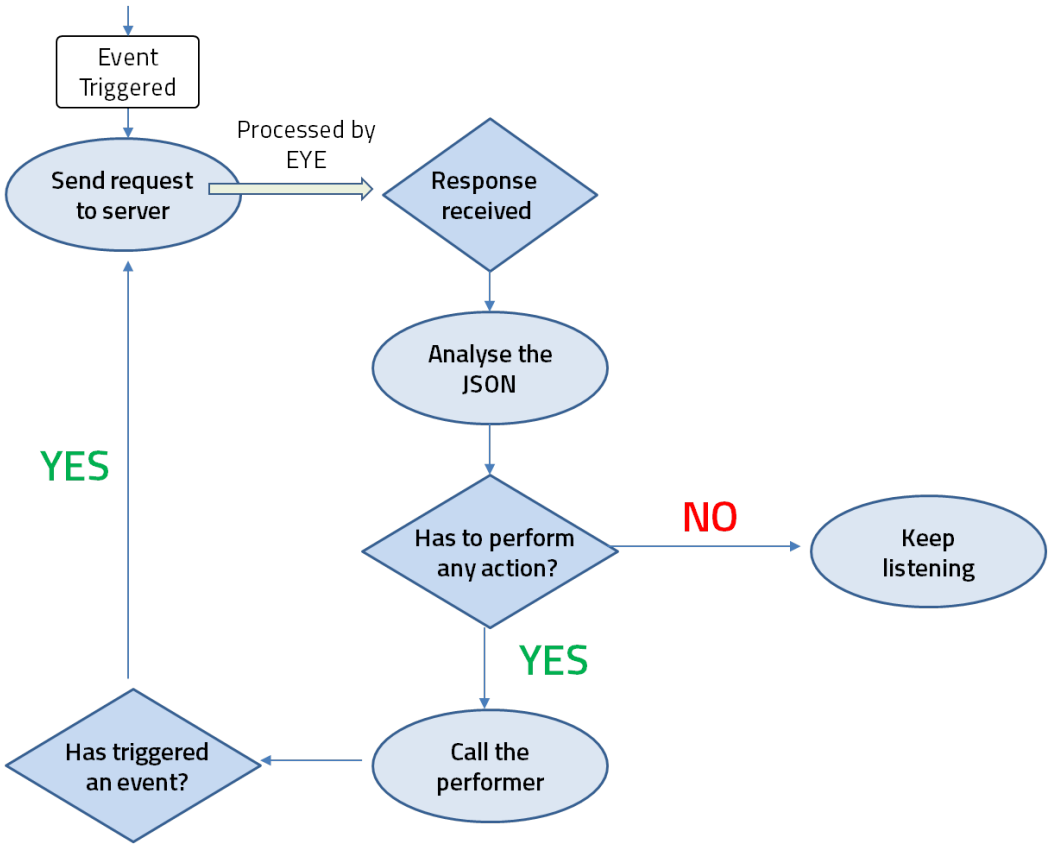

Figure 3.6: Decision diagram of the execution cycle

The server passes this input to the EYE engine, which returns a string corpus containing prefixes, the input inserted and, if it is successful, the output. The server filters this output and generates a JSON with the desired response. This JSON is sent back to the aplication, where it is parsed. Finally the real response is extracted and executed, causing the performance of an action, or nothing happens if there is not a rule defined with the event

triggered.

Sometimes an action performance can be the trigger of another action, restarting the process described above and converting this action to event, listening again the server response and causing an action chain process.

Attending to Figure 3.6 we can observe the whole execution cycle from the event triggering to the action performance. Furthermore, we can notice the loop perspective when an action becomes an event as we explained in the previous paragraph.

The JSON server response looks like the code fragment included below. It is compodsed by a success variable which means if the EYE engine has generated any successful answer. This has attached an action array which contains the result processed by the server. We can extract the channel that has to perform the action, the action that has to be performed and the parameter in case it is needed.

```
{
        "success":"1",
        "actions":[{
                                "channel":"",
                                "action":"",
                                "parameter":""
                                }]
}
```

Figure 3.7: Example of server response JSON structure

3.3.2.2 Receivers and Performers

To perform the rule execution it is necessary to implement receivers and performers for every channel we want to listen to. In this project, we have picked the most important ones, because the smartphone has lots of resource modules to listen about, most of them irrelevant for trigger events.

In the mobile aplication we have selected the most important smartphone resources.

In Table 3.1 and 3.2 we have described those channels properties, with their events and actions.

Table 3.1: Receivers table

Receivers	
Name	**Events**
Bluetooth	Turn On, Turn Off
Wifi	Turn On, Turn Off
GPS	Turn On, Turn Off
Calendar	Event Alert
Location	Enter Area, Exit Area
Data Network	Turn On, Turn Off

Table 3.2: Performers table

Performers	
Name	**Actions**
Notification	Show text
Toast	Show text
Bluetooth	Turn On, Turn Off
Wifi	Turn On, Turn Off
GPS	Turn On, Turn Off
AudioManager	Vibration, Silent or Normal mode
Brightness	Set Level
Data Network	Turn On, Turn Off

This framework enables new channel aditions through the server website, where the user can set the new event action properties. When the mobile aplication gets the channels, this new one will appear and it will be possible to build new rules using it.

Some of the events and actions have optimal parameters attached. These parameters are usually string type, but they could be a number or boolean. An event or action can have more than one parameter, for example the Location channel entered event need the latitude, longitude and radius of the area.

3.4 Semantic rules

3.4.1 Modelling channels with EWE

In this section we are going to explain how the channels are modeled using the EWE ontology [1]. For this I will support the explanation with a channel definition example, describing their events and actions behaviour. It's neccesary to define a vocabulary for each channel used in rule creation. Every channel follows the same structure, a class which defines the channel and another for each event and action the channel has. This classes will be useful to filter the rules in the server and trigger an event only when it belongs to an specific class. For example, for the Bluetooth channel:

```
ewe-bluetooth:Bluetooth a owl:Class ;
    rdfs:label "Bluetooth smartphone module"@en ;
    rdfs:comment "This channel represents a bluetooth module."@en ;
    rdfs:subClassOf ewe:Channel .
```

Figure 3.8: Bluetooth channel with EWE ontology modelling

Now the channel is defined, it is necessary to implement the events and actions of that channel (Figure 3.9).

Because the Bluetooth events and action are very similar each other we are going to describe the action of the notification module. We can notice that a string parameter is required. This parameter represents the text that the user wants to be displayed (Figure 3.10).

```
ewe-bluetooth:TurnON a owl:Class ;
    rdfs:label "Bluetooth turned ON"@en ;
    rdfs:comment "This Trigger fires every time you enable
    Bluetooth."@en ;
    rdfs:subclassOf ewe:Event ;
    rdfs:domain ewe-bluetooth:Bluetooth .

ewe-bluetooth:TurnOFF a owl:Class ;
    rdfs:label "Bluetooth turned OFF"@en ;
    rdfs:comment "This Trigger fires every time you disable
    Bluetooth."@en ;
    rdfs:subclassOf ewe:Event ;
    rdfs:domain ewe-bluetooth:Bluetooth .
```

Figure 3.9: Bluetooth events using EWE ontology modelling

```
ewe-notification:Show a owl:Class ;
    rdfs:label "Show a notification"@en ;
    rdfs:comment "This action will show a notification."@en ;
    rdfs:subclassOf ewe:Action ;
    rdfs:domain ewe-notification:Notification .
```

Figure 3.10: Notification actions using EWE ontology modelling

3.4.2 Rule declaration example

Once the channel is fully described let's try to create a rule. According to the channels defined on the previous section (*Bluetooth* and *Notification*), the rule statement example could be *If I enable Bluetooth then show a Notification saying "Hello World"* (Figure 3.11).

```
{
    ?event rdf:type ewe-bluetooth:ON .
}
=>
{
    ewe-notification:Notification rdf:type ewe-notification:Show;
       ov:message "Hello World".
}.
```

Figure 3.11: IF - THEN rule declaration using vocabularies

The last step is define how the input should be to trigger the rule we have just created (Figure 3.12).

```
ex:smartphone rdf:type ewe-bluetooth:Bluetooth .

ex:event rdf:type ewe-bluetooth:TurnON ;
    ewe:generatedBy ex:smartphone ;
```

Figure 3.12: Input declaration using vocabularies

3.5 Android aplication

3.5.1 User Interface

In this section we are going to explain the mobile aplication structure, which can be divided in activities. Basically the aplication is formed by 2 activities, the *ListRules Activity* (Figure 3.13) and the *NewRule Activity* (Figure 3.14). In the first one we regroup the content stored in the rule SQLite [10] database, showing them in a ListView. Each row has the rule description inclueded when the rule was created and the *If- Then* clause, with the corresponding channel pictures.

The second activity handles the creation process and its divided into two different frag-

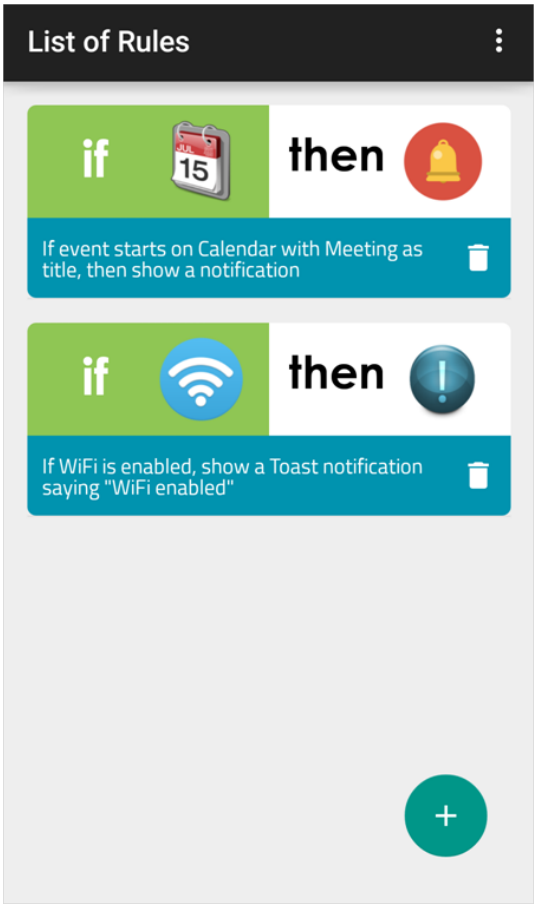

Figure 3.13: ListRules activity on Android app.

ments: the IF fragment wher the user has to select the channel and the event, and the DO fragment where the action is selected and finally the rule is saved in the database with the name, description and place. To end this process, the activity calls an asynctask which post the rule into the server and close the activity.

During the definition process some alert dialogs can be showed for filling the event and action parameters, or just the rule attributes. We can observe this behaviour in Figure 3.15. The user can add new rules by tapping the plus symbol button in the *ListRules* activity. All the rule definition process is supported by the *Rule Definition Service*, completing the server post request. The management of the local and remote database allows the user to remove the rules by clicking on the remove icon.

The channels showed in the IF and DO tab comes from the server. This list can be modified adding new channel through the website, where you can define the events and actions, and set the title, description and a representative icon.

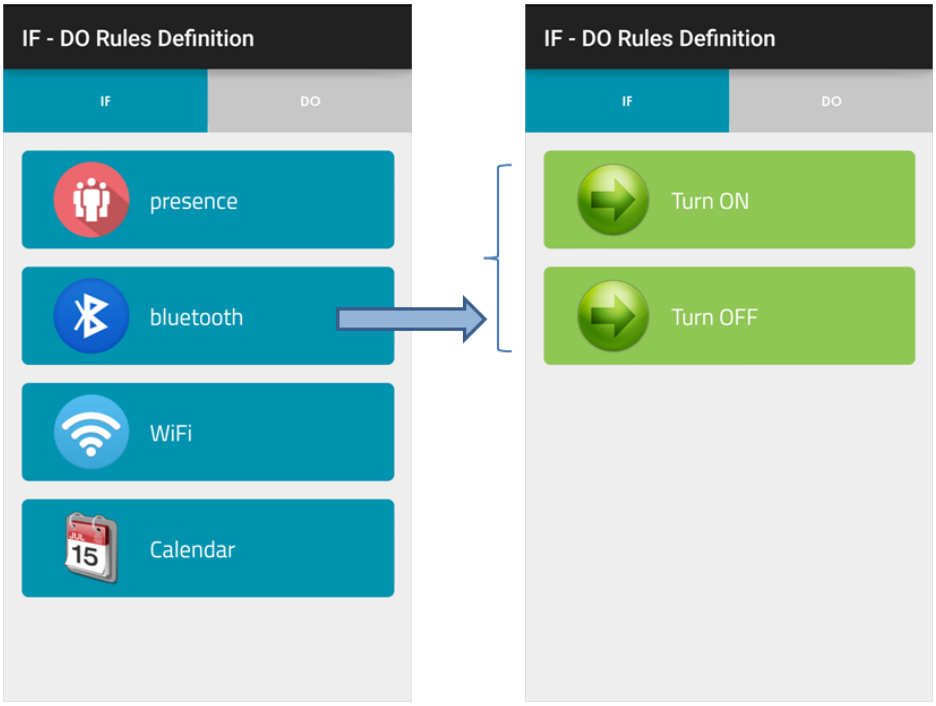

Figure 3.14: Channel - Event selection on New Rule activity.

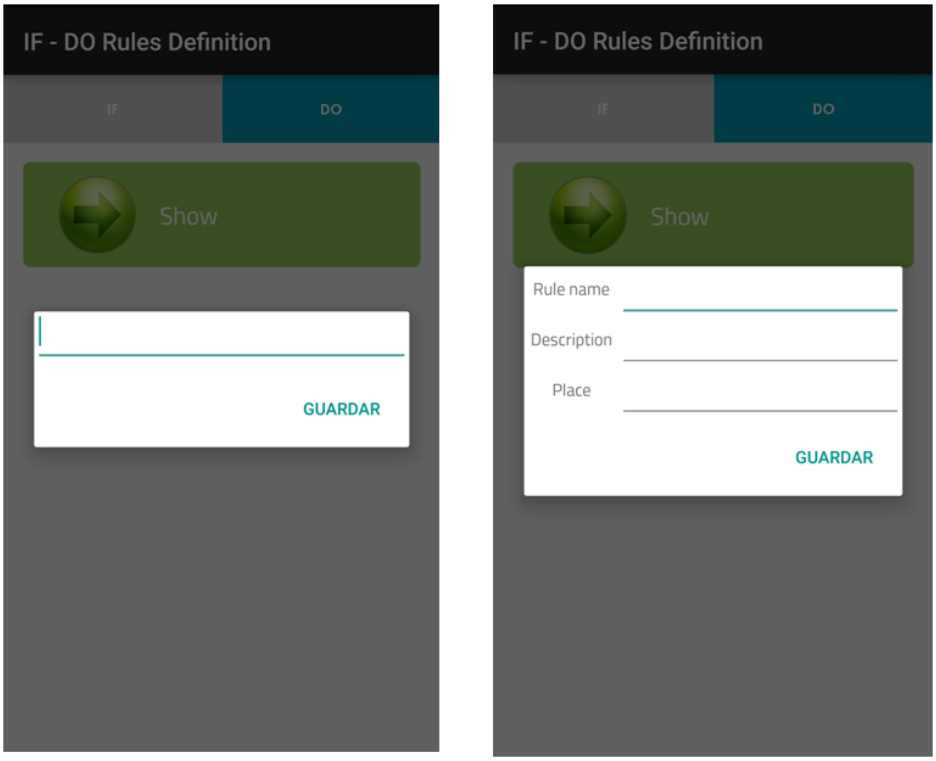

Figure 3.15: Request parameters and rule general attributes.

3.5.2 Channel Definition

For channel definition it is required to use the task automation service website because fill the fields from the mobile phone can be really hard. The channel creation process is quite simple and fast, and it can be edited whenever is needed. The attributes required for setting up the channel are the title, description, the channel nice name, the logo, events and actions. Each event and action is formed by the prefixes, the rule statement of the event/action and the title.

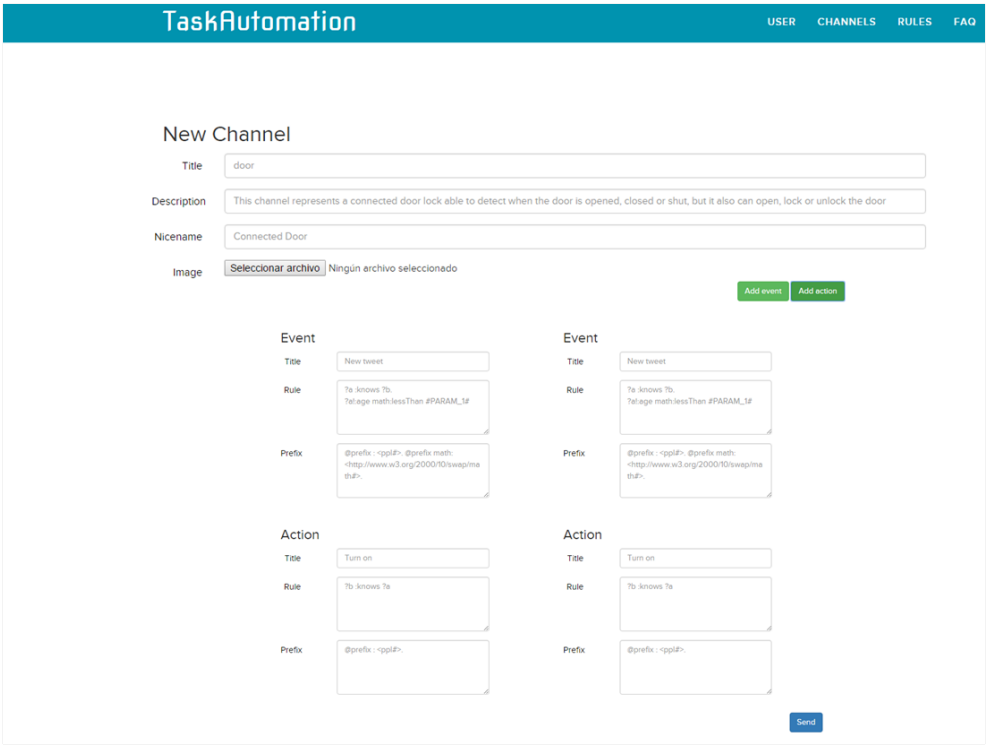

Figure 3.16: Channel creation in the task automation website.

3.5.3 Structure

The architecture of the Android framework for task automation of the smartphone has been structured as a set of cooperating classes, following Android abstractions and components, such as *Activities*, *BroadcastReceivers*, *AsyncTasks*, *Services* and auxiliary classes. In this project, these components have been selected according to what the framework needs.

When the app is launched, the *onCreate* method executes the *GetChannelsAsyncTask*. This task will obtain the complete channel list from server in JSON format. In the *onPostExecute* method, this JSON is saved locally in *SharedPreferences* for future usages. When

a *NewRuleActivity* intent is triggered, this JSON is processed, obtaining a *List<Channel>* (procedure explained on fig. 3.3).

After the rule definition process has ended up, a new *PostRuleAsyncTask* is executed, making a *POST* request to the automation server with the attributes described on figure 3.17. These attributes have been saved in the *RuleDefinitionService* using static variables because this service is a *singleton*. The task extracts these values to generate the request in the *doInBackground* method.

Every local resource (3.1) in the smartphone is a channel, and has one *BroadcastReceiver* which listens changes in the channel, and one *Performer* that has a single method for each action. Everytime an event is triggered, the *BroadcastReceiver* calls the rule execution module giving the channel and the event as parameters. Calling the *createInput* method, the aplication extract the prefixes an the rule writen in N3 format and generates the complete input string that will be inserted into EYE [8] engine by the remote server.

A new *PostInputAsyncTask* is created, giving attributes such as place, user and the input string in the *doInBackground* method. The task waits for the response asyncronously and handles the JSON on the *onPostExecute* function. This JSON is processed, extracting the actions array. For each action, the rule execution module tries to find the channel expressed on the *channel* attribute (fig. 3.7)and associates it a performer. If success, the performer action with the same action title will be called, attaching the parameters if necessary.

Once the performer is selected, depending on the multiple actions implemented on it, the aplication executes the one specified in the response (procedure explained in fig. 3.6).

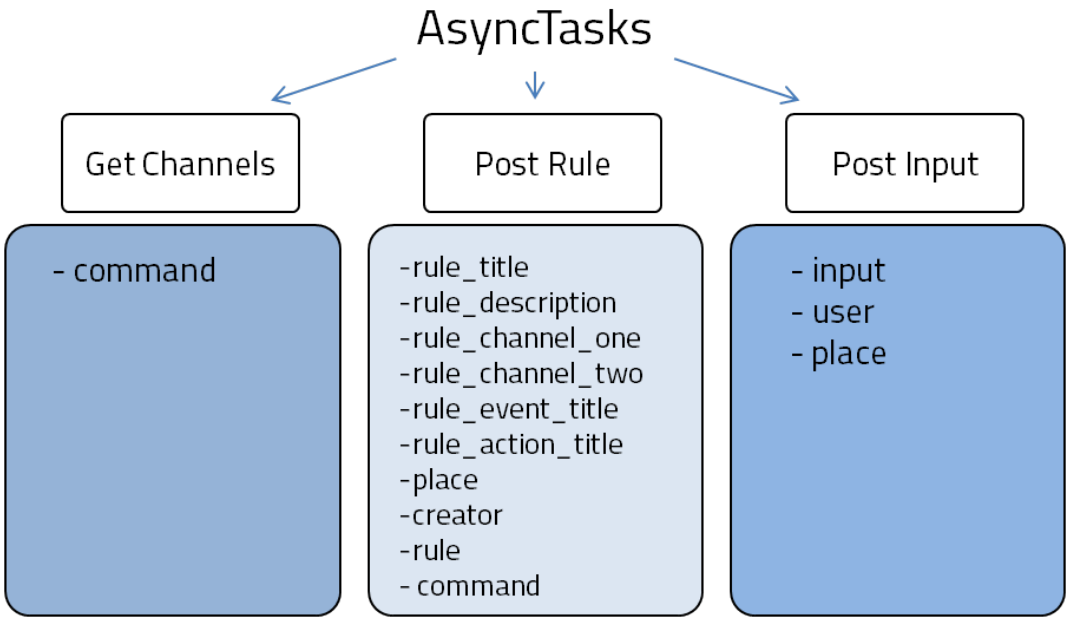

Figure 3.17: Asynctasks attributes attached to the remote server request.

3.6 Conclusions

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

To sum up, this project counts with an Android aplication from where the user can create and manage semantic rules, using local or remote channels. All these rules are sent to the server, where they are saved. When an event is triggered, the smartphone aplication generates an input depending on the event launched and post it to the server. This server will generate the corresponding action response connecting with the EYE semantic reasoner. The mobile aplication will receive this response and link it to a channel action, executing it locally.

CHAPTER 4

Case study

4.1 Introduction

In this chapter we are going to describe the features provided by the developed Android aplication. For that, it's necessary to consider a main use case to explain how the implemented functionalities can be applied, and help to understand the project value.

The actor of this case is the user who runs the aplication, whose objective is to import a channel, and use it to create an automated task that will involve other channels. To achieve this purpose, the user will have to use the server website to define the channel attributes and the mobile aplication to edit and execute the rule.

4.2 Channel Edition

The first step the user needs to take is the channel creation. For this, he has to access the Task Automation website [2] and choose the option *Create New Channel* by clicking on the Channels tab. A form will be showed with a field for every parameter the channel needs to be created. In the figure 4.1 and 4.2 we can observe this form.

In the first group of fields we can see channel attributes such as title, description, the nice name and the image that will represents the channel. Below those attibutes we can observe the events and actions selector. The user can add or remove them in case the channel does have multiple events or actions. Each event and action has attributes like title, rule and prefix. The title represents the event or action that is being defined, the rule represents the N3 fragment code that will be used when the event is triggered or the action is performed. By last, the prefixes import the libraries needed to execute the rule statement in the EYE engine.

Once all these fields are completed, the user has to click the *Send* button, saving this channel in the MongoDB database. The user must repeat this process for every channel involved on rule creation process.

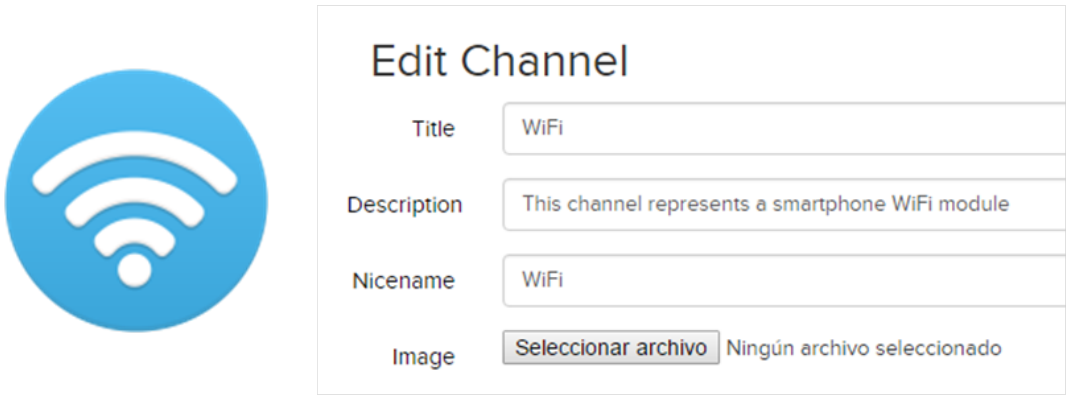

Figure 4.1: Wifi channel general attributes.

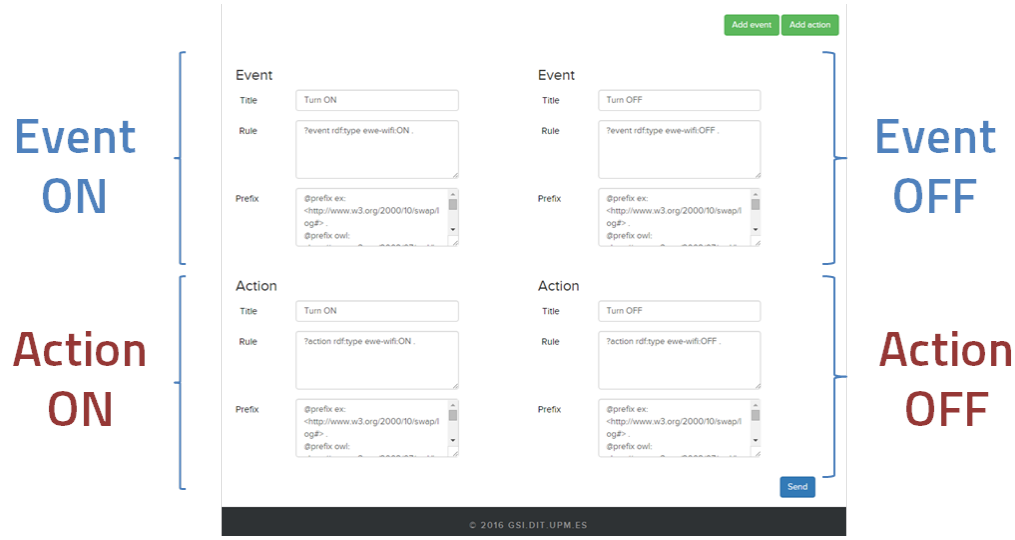

Figure 4.2: Wifi events and actions. [2]

4.3 Rule Edition

Now the channels have been created, the users must launch the aplication on his device. The *ListRules* activity will looks empty because there is not any rule configured. By tapping the plus button, the user will start the rule definition process. The asyncronous task will get all the available channels from the server and display them in the IF tab, in case the channel has at least one event.

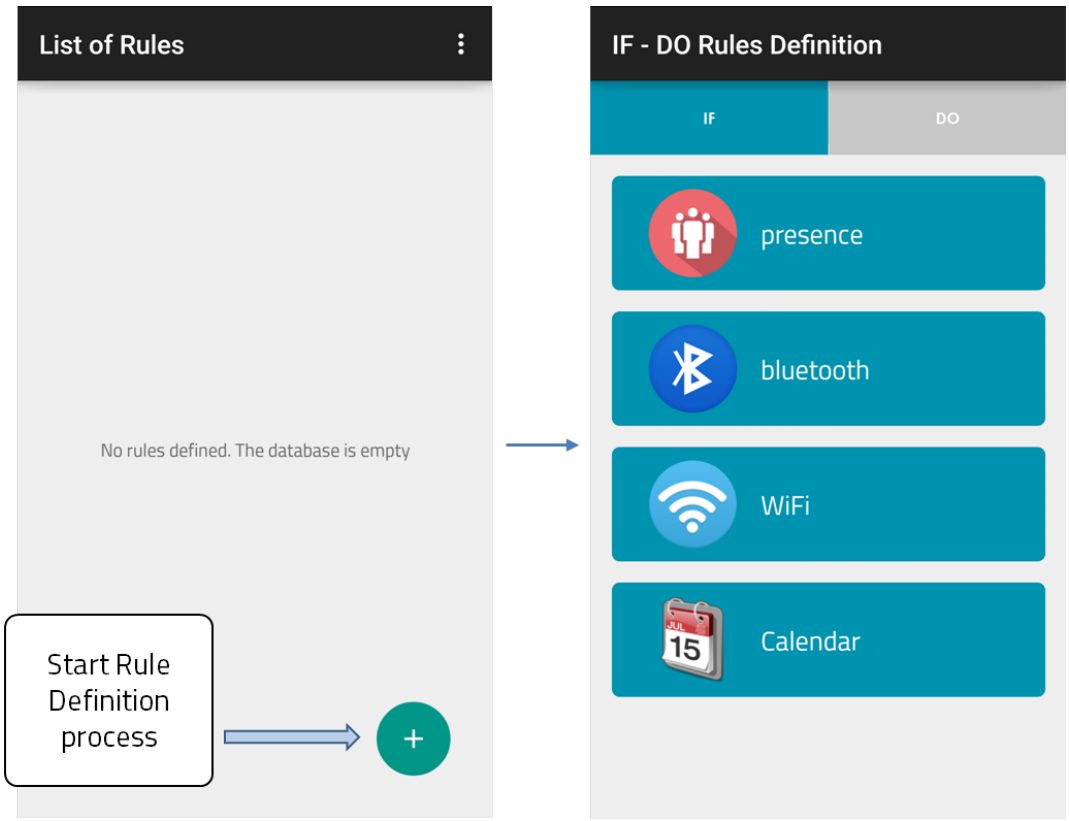

Figure 4.3: Start Rule Definition process from aplication

After the *NewRule* activity initialization, the process starts with the selection of the IF tab parameters. During this, the DO tab is disabled so the user cant set up the rule haphazardly. First of all, the channels that have any event will be showed. The user just have to tap the channel card and the aplication will automatically add it to the service. Afterwards, the events of the channel selected previously will appear.The user can press the back button in case he has chosen the wrong channel. This is showed in Figure 4.4.

In case the user choose an event with parameters, the aplication will display an alert dialog where the user has complete writing the value and tapping the *Save* button. In this point, the IF part will be completed, changing the tab window to the DO one.

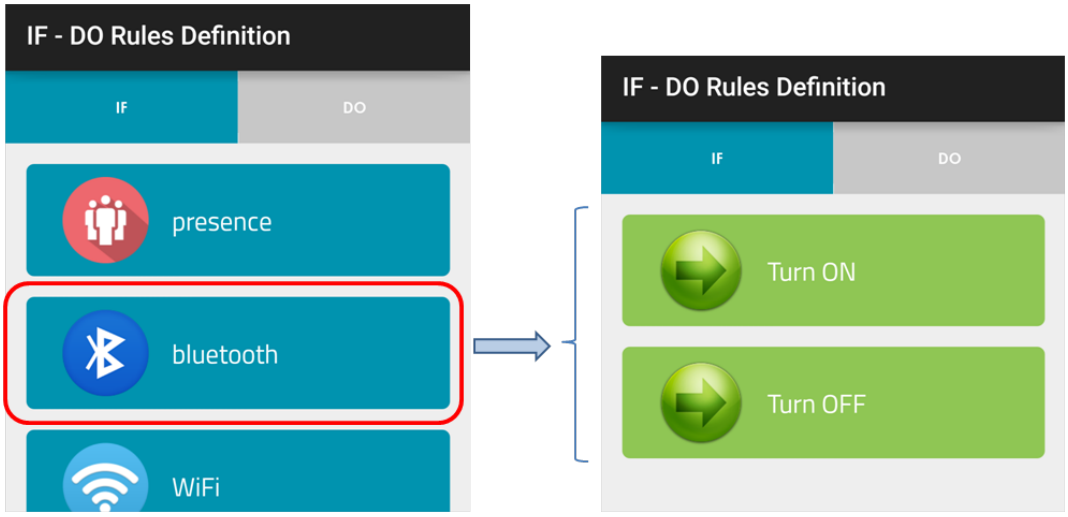

Figure 4.4: IF tab behaviour on Rule Definition process

In the DO tab the user will visualize the channels that have actions available. In the same way, he will select the channel and the action that will be performed. Analogously, if the action has parameters an alert dialog will be displayed requesting them.

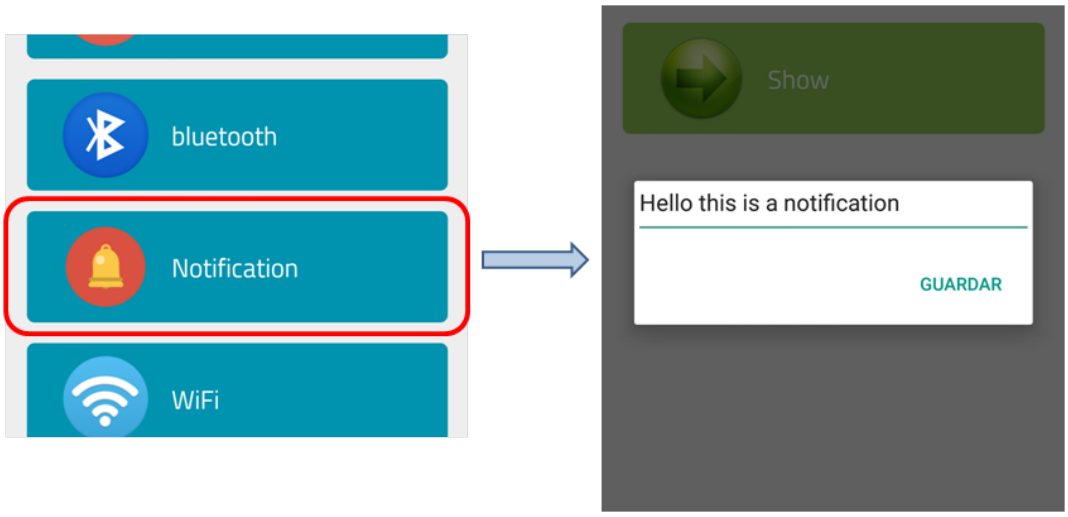

Figure 4.5: DO tab behaviour on Rule Definition process

Once the rule is completely defined, the aplication will show an alert requesting the context parameters (title, description and place). Finally the user needs to press the *Save* button and the asynctask will begin. All the choices will be sent to the server and stored locally. The aplication will close the Rule Definition activity and reset the service parameters.

The list of rules now won't be empty and will have the rule element that the user have

just created.

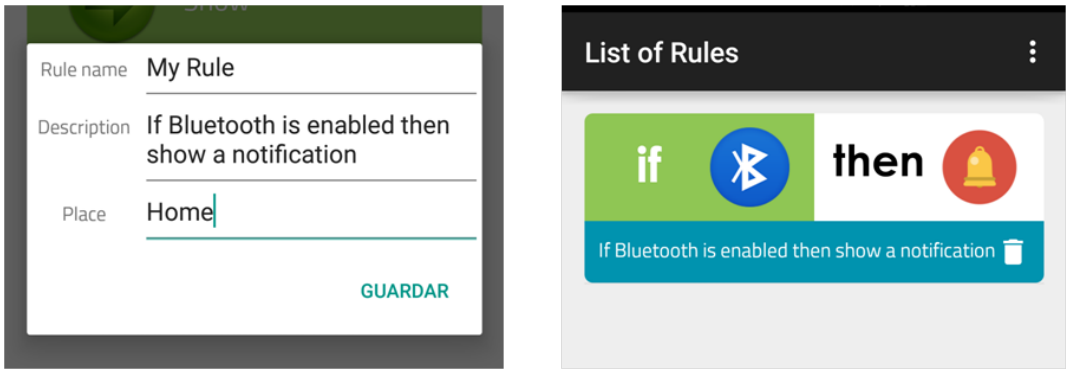

Figure 4.6: The final step on Rule Definition process

4.4 Rule Execution

In this situation, the rule has already been created and the user just needs to trigger the action to execute the rule. The process that will be explained is described in depth in figure 3.6 presented in the architecture chapter.

In order to clarify this process I will use the rule defined in the previous rule definition process *If I turn on Bluetooth then show a notification*. When the user enables Bluetooth, the receiver listening the intent *STATE CHANGED* will be triggered with *STATE ON* value. The fragment of code implemented in that section will be executed, calling the Rule Execution module. In this module, the rule input of the event triggered will be send with more attributes for the server. This will launch a background task obtaining the EYE engine output as server response.

After process this response, the action *Notification Show "Hello this is a notification"* will be performed. In case the action supposes the launch of another event, this process will be repeated cyclically.

CHAPTER 5

Conclusions and future work

5.1 Conclusions

To conclude this project we will resume the principal concepts that are explained in this document. We have developed an Android aplication which communicates with an automation task server and allows the user to create and define customizable rules using local or remote resources. The channels that participate in the rule creation process are modeled by the EWE ontology. Each channel has events and actions, declared in N3 format for the rule definition. All the rules defined from the smartphone are saved locally and in the server. These rules can be removed or edited whenever the user wants.

The execution of these rules is based on the implementation of receivers and performers for every channe event and action respectively. These receivers and performers define the posible events and action that a channel may have. When an event is triggered, the aplication contacts with the server and receive as response the result of inserting the input in the EYE [8] engine.

The main objective during the project development has been the creation of an aplication that facilitates the smartphone usability and makes people life easier with a platform that allows one to automate procedures executed frecuently.

For the project implementacion we have keep in mind innovative tecnologies like the usage of an inference engine to execute the rules, basing the user interface on aplications with the same purpose which has come up recently, using an intuitive and optimal ditribution in screens limited by small sizes. Likewise, in the persistence layer we have chosen SQLite [10] for storing rules locally and MongoDB in the server.

Finally we have suggested an implementation that facilitates the implementation of Smart Homes through the smartphone aplication with an approach which focuses on save time and program routines in an easy and intuitive way.

5.2 Achieved goals

The achieved goals obtained developing this project are:

Create a graphical interface which allow user to define rules We have created an accessible enviroment inside Android aplication where the user can customize his own rules and adapt the framework to define the IF - DO statements that he needs. The interface simplicity allows the user to know what rules are enabled or not, and what happens when the rule is executed showing a brief description.

Build a local persistent model to save those rules The persistence layer of the aplication store rules locally to prevent excesive requests to the server every time the user launches the aplication, and for showing only the rules he has configured. On the other hand, every time a rule is created, it is sent to the server, where it is saved until an input executes it. This model is flexible and allows the user to manage the database by removing or editing any rule from aplication or the server website.

Develop an interface which enables to communicate with a remote server The aplication depends partially on the server. This involves the implementation of a class that handles the request and responses, filling the parameter with the appropriate values when the aplication needs to post a rule or an event has been triggered.

Execute rules through EYE engine All the events and actions are described in N3 and modelled following the EWE ontology. This means that each channel has it own vocabulary. This N3 statements are inserted in the EYE engine by the server. The implementation of the EYE engine with the aplication has been an important achievement in this project because the engine returns the prefixes mixed with the input and output, which complicates the action extraction.

Define receivers and performers that execute events and perform actions This is the main module of the aplication because is responsible of the rule execution. We have developed a receiver which listens all the changes that can trigger an event in a particular channel, handling the communication with the server. The performers have functions with the action performance and can be executed when the server responses an action involved with that performer.

5.3 Future work

There are several tips that can be followed to improve some of the project features but were not implemented into this project because of the time limitation. We will mention some upgrades from a future perspective that could help to continue the development of the project we have presented inside this document.

- Implement a login system where each user could have their own profile and share rules each other
- Give the possibility to the user of creating category of one specific topic (home, work, gaming...) and group rules by categories. This with the previous point could create a social network where users might download rule packages oriented to a particular life situation and export their own rules.
- Create a top rated rules ranking where the most interesting rules will appear.
- Increase the channel list with some third-party application like social networks. In the current version of the project, this is viable but the server is the one who accesses directly to these apps.
- Add the active/inactive field to created rules, which one allows to disable certain rules without deleting them from database.
- Improve the access to the inference engine with EYE, creating a better filter which handles multiples parameters and types offering much more possibilities.
- Adapt the application to other devices like tablets or different platforms like iOS framework.

APPENDIX A

Rule and channel templates using N3

In this appendix we will detail in depth the channel and rule definition using the EWE ontology [1]. This appendix is essential to understand how the rule execution module works and the way the channels are modelled in N3 for the EYE [8] inference engine.

A.1 Channels

We will start with an example channel definition that belongs to the WiFi channel, which has two events and two actions. This template can be generalized to every channel with turn on and turn off events, for example Bluetooth, GPS or Data network connection.

In case WiFi is enabled, the input inserted into inference engine will be *?event rdf:type ewe-wifi:ON.*. However, if any event cause the WiFi enable action, EYE will return *ewe-wifi:Wifi rdf:type ewe-wifi:ON .*

A.1.0.1 Event parameters

Some certain events may have attached parameters, for example the Location channel, which needs the latitude longitude and radius of the geofence area. It is necessary to follow a specific

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

#Events
?event rdf:type ewe-wifi:ON.
?event rdf:type ewe-wifi:OFF.

#Actions
ewe-wifi:Wifi rdf:type ewe-wifi:ON .
ewe-wifi:Wifi rdf:type ewe-wifi:OFF .
```

Figure A.1: WiFi channel template described in N3

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

?event rdf:type ewe-location:Entered;
?event!ewe:latitude math:EqualTo #PARAM_1#.
?event!ewe:longitude math:EqualTo #PARAM_2#.
?event!ewe:radius math:EqualTo #PARAM_3#.
```

Figure A.2: Location channel template described in N3 using parameters.

structure when the channel is being defined. We can observe this implementation in figure A.2.

A.1.1 Action parameters

In the same way, some actions may have attached parameters, for instance the text we want to show inside a notification or the message we want to post on Twitter. These implementations must follow the template described in figure A.3.

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

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

Figure A.3: Notification actions template described in N3

A.2 Rules

Once we have define several channel structure templates, we are going to create some rule templates using these channel examples.

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix math: <http://www.w3.org/2000/10/swap/math#>.
@prefix ewe: <http://gsi.dit.upm.es/ontologies/ewe/ns/#> .
@prefix ewe-location: <http://gsi.dit.upm.es/ontologies/ewe-location/ns/#>
    .
@prefix ewe-wifi: <http://gsi.dit.upm.es/ontologies/ewe-wifi/ns/#> .
{
  ?event rdf:type ewe-location:Entered;
  ?event!ewe:latitude math:EqualTo 40.453217 .
  ?event!ewe:longitude math:EqualTo -3.725631 .
  ?event!ewe:radius math:EqualTo 30 .
}
=>
{
  ewe-wifi:Wifi rdf:type ewe-wifi:ON .
}
```

Figure A.4: Rule example. If I enter GSI the enable WiFi in my smartphone

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix math: <http://www.w3.org/2000/10/swap/math#>.
@prefix ewe: <http://gsi.dit.upm.es/ontologies/ewe/ns/#> .
@prefix ewe-wifi: <http://gsi.dit.upm.es/ontologies/ewe-wifi/ns/#> .
@prefix ewe-notification: <http://gsi.dit.upm.es/ontologies/ewe-
    notification/ns/#> .
@prefix ov: <http://vocab.org/open/#> .
{
  ?event rdf:type ewe-wifi:ON .
}
=>
{
  ewe-notification:Notification rdf:type ewe-notification:Show ;
  ov:message "You have enabled WiFi".
}
```

Figure A.5: Rule example. If I enable WiFi then show a notification

Bibliography

[1] 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

[2] S. M. López, "Development of a task automation platform for beacon enabled smart homes," http://www.

[3] IFTTT, "Ifttt's official website," https://ifttt.com/, accessed March X, 2015.

[4] Atooma, "Atooma's official website," https://www.atooma.com/.

[5] M. Coronado and C. A. Iglesias, "Task Automation Services: Automation for the masses," *Internet Computing, IEEE*, vol. PP, no. 99, pp. 1–1, 2015.

[6] Atooma, "Resonance android sdk's documentation," http://doc-resonance-sdk.readthedocs.org/en/latest/.

[7] J. de Roo, "Eye note," http://eulersharp.sourceforge.net/2006/02swap/eye-note, accessed March X, 2015.

[8] R. Verborgh and J. de Roo, "Drawing conclusions from linked data on the web. the eye reasoner," http://online.qmags.com/ISW0515?cid=3244717&eid=19361&pg=25#pg25&mode2.

[9] O. Araque, "Design and implementation of an event rules web editor," July 2014.

[10] SQLite, "Sqlite documentation," https://www.sqlite.org/docs.html.