TRABAJO DE FIN DE GRADO

Título:	Diseño e implementación de un editor web para reglas automatizadas disparadas por eventos
Título (inglés):	Design and implementation of an event driven automation rules web editor
Autor:	Óscar Araque Iborra
Tutor:	Carlos A. Iglesias Fernández
Departamento:	Ingeniería de Sistemas Telemáticos

MIEMBROS DEL TRIBUNAL CALIFICADOR

Presidente:	Gregorio Fernández Fernández
Vocal:	Mercedes Garijo Ayestarán
Secretario:	Carlos Ángel Iglesias Fernández
Suplente:	Tomás Robles Valladares

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 DE FIN DE GRADO

DESIGN AND IMPLEMENTATION OF AN EVENT DRIVEN AUTOMATION RULES WEB EDITOR

Óscar Araque Iborra

Julio de 2014

Resumen

Esta memoria es el resultado de un proyecto cuyo objetivo es diseñar y desarrollar un editor web de reglas automatizadas disparadas por eventos.

El editor nos permite crear y editar estas reglas automatizadas no usando sintaxis compleja, sino mediante de una interfaz gráfica simple. Esta herramienta es capaz de crear un gran número de reglas diferentes a partir de varios canales predefinidos que proveen de recursos para estas reglas.

Este editor esta respaldado por varios módulos que le proveen con funcionalidades más complejas. En este contexto presentamos una herramienta que nos permite gestionar todas estas reglas y canales, además de persistirlos usando dos bases de datos diferentes.

Para continuar, se presentará otra herramienta. Esta permitirá al usuario final activar las reglas creadas desplegándolas en un motor semántico que las tratará y provocará diferentes acciones dependiendo de los eventos generados. Además, presentaremos un simulador que nos permita probar todas estas funcionalidades.

Por último, recogemos las conclusiones extraídas del proyecto, las tecnologías que hemos aprendido durante el desarrollo del mismo y las posibles líneas de trabajo futuro en relación con la continuación de este proyecto.

Palabras clave: Tecnologías semánticas, RDF, SPARQL, SPIN, PHP, JavaScript, Java, Knowckout JS, EWE, MySQL, MongoDB

Abstract

This thesis collects the result of a project whose objective is to design and develop a event driven automation rule web editor.

The editor allows us to create and edit these automation rules not using complex syntax, but a simple graphical interface. This tool is able to create a great number of different rules as of several pre-defined channels that provide resources for these rules.

This editor is supported by several modules that provide the editor with more complex functionalities. In this context, we present a tool that allows us to manage all these rules and channels, and to persist them using two different databases.

To continue, another tool will be presented. It will allow the final user to activate the created rules deploying them into a semantic motor that will handle them and trigger different actions depending on the generated events. Besides, we will present a simulator that allows us to test these functionalities.

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

Keywords: Semantic technologies, RDF, SPARQL, SPIN, PHP, JavaScript, Java, Knowckout JS, EWE, MySQL, MongoDB

Agradecimientos

A mis padres

Contents

List of Figures

CHAPTER 1

Introduction

1.1 Context

The term *Live Web* [14] describes a new stage in the evolution of the Web that extends Web2.0 [11] interactive Web. Instead of simply browsing static web pages or even interacting with a web site or social network, the Live Web is characterized by a brand new style of interaction through dynamic streams of information to present contextual, relevant experiences to users [14]. There are several sources of these dynamic streams, such as social networks, sensor networks, and mobile phones which together provide the necessary location-aware, relationship-aware, preference-aware and sensory context to achieve a new generation of context-aware applications [1].

However, what makes these tools really useful for consumers is that they combine incoming data with many other services available on the Web, especially when the interconnection involves some kind of data transformation. Thus they create new personalized streams and services that behave in a particular way for each different user [5].

Therefore, the Live Web endorses a new way of composing services. The Live Web uses the cloud [10] as platform (i.e. every sensor, data source and service is available in the Cloud). Thus, cloud computing allows us interconnection, hastening the rise of applications

which allows users to configure their services based on the data and events from their personal streams [7].

On top, for the last years the number of web services [12] have increased markedly. Many of these services, especially those of them already mature, offer a public API (*Application Programming Interface*) in order to interconnect them with other independent services, making them easily accessible. Thanks to these third party application and services are appearing, making use of these APIs. This new outlook offers plenty of possibilities. However, performing this connection can be complex, requires advanced skills sometimes and the learning curve is not lineal -but usually exponential- to the number of services involved. These reasons are what are not allowing the user to benefit from these features.

Into this scenario, plenty of new companies are emerging as task automation platforms. Some examples of these are IFTTT[1] or Zapier[2]. These tool are in charge of allowing the user to automatically perform tasks such as " *When I am mentioned in Twitter, send me an email*". In other words, these platforms are able to trigger rules that produces some sort of actions when certain events [13] are computed connecting multiple services [9]. These rules are triggered by user's events, which means that they work only with personal accounts from each one of the used services [8].

However, the current format for these rules are event-condition-action so they do not benefit from all the potential that these platform could offer. In this project, we make use of DrEWE[3]. That will allow us to provide an enhanced platform allowing the user to generate more complex rules.

In this context, DrEWE exists as a Task Automation Platform, which has some big features such as: EWE ontology support, which means that DrEWE implements the EWE ontology; and complex event processing. This last feature allows us to have a temporal reasoning in the described rules ([4] and [2]). One example of that could be: *"When a meeting is scheduled, if the corresponding attendees enter their Id cards at the entrance during ten minutes before the start time: generate an event"*.

These type of rules -such as the above described- are difficult to generate in a systematic and flexible manner. Considering this background, we present **Wool**: a web editor of event driven automation rules that possibilities the creation, edition, acquisition and deletion of these rules. The main characteristic of Wool is the graphic interface which is totally oriented to ease the edition of automation rules. Besides, Wool is oriented to complement

[1] http://www.ifttt.com
[2] http://www.zapier.com
[3] https://github.com/gsi-upm/DrEWE

the platform DrEWE, which means that the resulting rules from Wool can be used with DrEWE. That allows us to easily create a rule, and then deploy it into the DrEWE motor.

1.2 Project goals

In the long term, this project aims to provide a scalable platform for edition of automation rules driven by events. This includes, but is not limited to: an accessible platform for task automation; accessible event driven rules; a web handler for rule edition as well as to administrative tasks; an interface for DrEWE tasks.

Among the main goals inside this project, we can find:

- Design and build a web editor that allows users to create, edit, obtain and remove event driven rules for task automation.
- Build a software module which is able to communicate with the platform for task automation. That is, DrEWE.
- Extend the DrEWE functionalities, easing the rules management.
- Design and build a web platform for user authentication, allowing users the edition of their own rules.
- Deepen the knowledge and usage of technologies covered in this project such as: rule engines, event networks and ontology management.

1.3 Structure of this document

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

Chapter 1 provides an introduction to the context in which this project is developed. Besides, it describes the main objectives to achieve once concluded. ***Chapter 2*** offers a description of the main standards and technologies on which this project rely. ***Chapter 3*** describes a brief requirement analysis of the system of this project. ***Chapter 4*** describes the complete architecture of the Wool system, decomposing it into several modules that will interact between them. ***Chapter 5*** offers an overview of a selected use case. It is explained the running of the modules to offer the detailed functionalities. ***Chapter 6*** sums up the conclusions extracted from this project, and we offer a brief view about the lines of future

work. Finally, the appendix provides with concrete information covering the more detailed aspects of the implementation of this system.

CHAPTER 2

Enabling Technologies

2.1 Rule Automation

A number of now prominent web sites, 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. One example of this simple task automation is *When I am mentioned in Twitter, send me an email*; *When I receive an important email, send me a SMS* or *Turn Bluetooth on when I leave work*. They are called Task Automation Service (TAS). Some TASs allow users to share the rules they have defined, so that other users can reuse these tools and adapt them to their own preferences [5].

Following the EWE Ontology model [5], it is possible to classify task automation services into *internet task automation* (Sect. 2.1.1) and *device task automation* (Sect. 2.1.2), although there is some overlap between these two categories since some task automation services can cover both task automation types.

Figure 2.1 presents the general architecture for both types. As shown, different TASs that exist on the Web in the present day are disconnected from each other. While Internet TASs filter notifications coming from Internet services, device-based TASs filter notifications from device sensors such as those in a smartphone, as explained below.

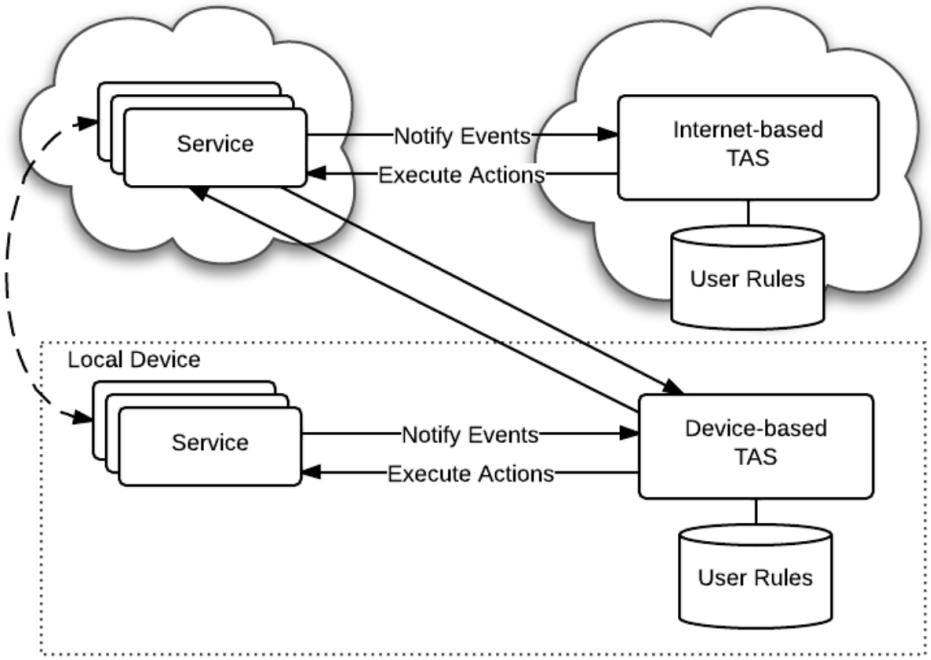

Figure 2.1: Internet based and device based TAS general architecture

2.1.1 Internet task automation

Internet based task automation services allow users to connect events and actions from two or more services. The most popular services are [5]:

1. *Ifttt*[1]. Nowadays IFTTT is the most popular task automation service on the Internet, with more than 65000 rules published. The so called recipes -rules- allow users connecting services -channels-. IFTTT includes more than 50 channels, and it is focused on social features.

2. *Zapier*[2]. Zapier is a paid service with more than 1.5 million API requests per month. Zapier's channels are thought of as for business users and are focused on integrating business services.

3. *CloudWork*[3]. CloudWork focuses on task automation for business apps. It is more specialized than Zapier: their approach to task automation and service integration is made from the point of view of Customer Relationship Management (CRM). This platform was designed for interoperability with most of the CRM software. Users can

[1] http://ifttt.com
[2] http://zapier.com
[3] https://cloudwork.com

connect data and processes that occurs within these applications to the web services available.

4. *Kinetic*[4]. Kinetic Rule Engine is an open source project designed for developing services in the Live Web. It has been developed by Kynetx[5]. The rules are programmed in a programming language denominated Kinetic Rule Language. This system classifies events into several domains: web, email, physical devices, voice and SMS and social services.

2.1.2 Device based task automation

Device based task automation remains in the configuration of a desktop or a mobile phone, as they do not combine web services at the event level yet. The two main applications available are [5]

1. *on{x}*[6]. This is a Microsoft project for automating Android devices and is aimed to developers -as it requires JavaScript skills-. The rules are JavaScript code deployed into an Android device from a web site where the rules are shared between users. These rule can notify each other using custom signals. Besides, the actions can be scheduled to be triggered at a specific time.

2. *Tasker*[7]. Tasker is and Android application for configuring the functioning of and Android device. It offers many built-in actions, which are grouped in sequential execution queries called *Tasks*. These Tasks are triggered by *profiles*, that group a set of conditions that must be met to trigger the task. These conditions are events of different kinds, date timer events, third party events, changes in internal variables, or manually triggered events.

2.2 EWE Ontology

Evented WEb Ontology(EWE)[8] is a standardized data schema (also referred as "ontology" or "vocabulary") designed to describe elements within Task Automation Services (detailed in Section 2.1) enabling rule interoperability. Referring to the EWE definition [3], the goals of the EWE ontology to achieve are:

[4]Available at https://github.com/kre/Kinetic-Rules-Engine
[5]http://www.kynetx.com/
[6]https://www.onx.ms
[7]http://tasker.dinglisch.net/
[8]http://www.gsi.dit.upm.es/ontologies/ewe/

- Provide a common model to represent TAS's rules so that it enables rule interoperability
- Enable to publish raw data from Task Automation Services (Rules and Channels) online and in compliance with current and future Internet trends.
- Provide a base vocabulary for building domain specific vocabularies e.g. Twitter Task Ontology or Evernote Task Ontology.

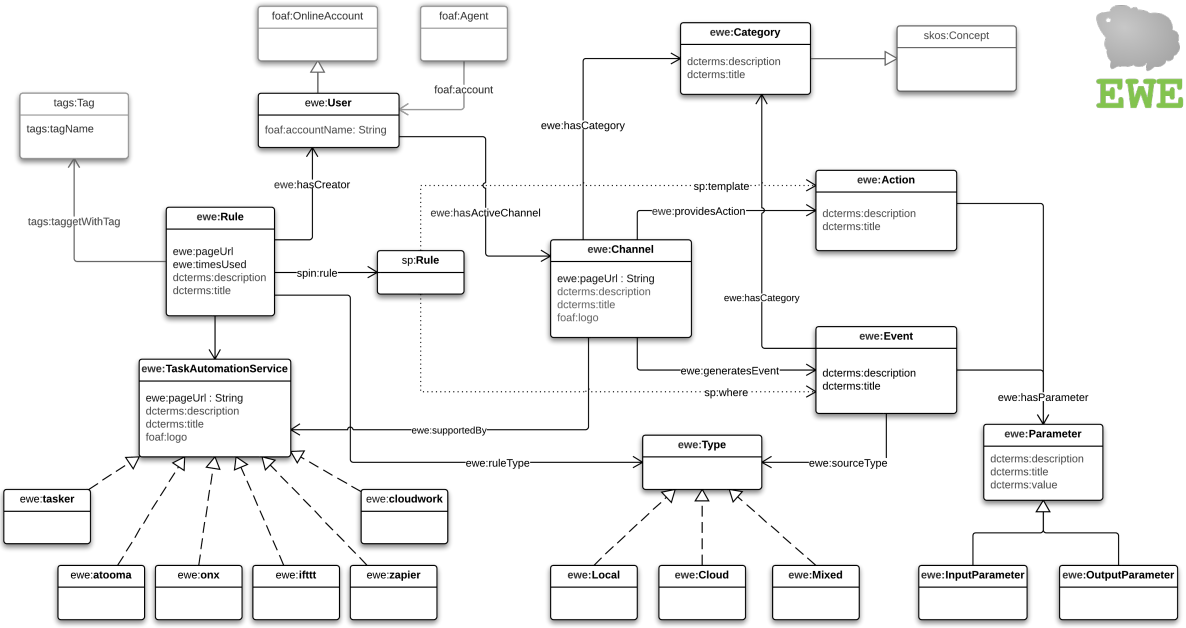

Figure 2.2: Class Diagram for the EWE Ontology

2.2.1 EWE Ontology main classes

As it can be seen form the figure 2.2 the core of the ontology comprises four major classes: Channel, Event, Action and Rule. They are briefly explained next [5]:

1. *Channel.* It defines individuals which either generates Events, provide Actions, or both. In this context, Channel can define a Web service or a device -or actuator- thus these last ones provides Events and Actions. Moreover, the definition of Channel also includes website wrappers -web apps that generate events based on occurrences or changes in third party websites-.
2. *Event.* This class defines a particular occurrence of a process. Events are instantaneous: they have no duration over time. Event individuals are generated by a certain

Channel, and they are triggered by the occurrence of the process which defines them. Events usually provide further details that can be used within Rules to customize Actions: they are modelled as output-parameters. Events also let users describe under which conditions should they be triggered. These are the configuration parameters, modelled as input-parameters. Event definitions are not bound to certain Channels since different services may generate the same events.

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

4. *Rule.* The class Rule defines an "Event-Condition-Action" (ECA) rule. This rule is triggered, and means the execution of an Action. Rules defines particular interconnections between instances of the Event and Action classes; those include the configuration parameters set for both of them: output from Events to input of Actions. EWE makes use of the SPIN framework[9] to represent the implementation of the ECA rules, since these rules can be described as SPARQL construct queries: SPIN rules [6].

A channel example of how the Channel class is expressed in RDF is shown next [5]:

Listing 2.1: EWE Channel definition

```
<owl:Class rdf:about="&ewe-mail;gmail">
  <dcterms:title>Gmail</dcterms:title>
  <dcterms:description>Webmail by Google.</dcterms:description>
  <ewe:generatesEvent rdf:resource="&ewe-mail;newEmail"/>
  <ewe:generatesEvent rdf:resource="&ewe-mail;newEmailLabeled"/>
  <ewe:hasAction rdf:resource="&ewe-mail;sendNewEmail"/>
  <rdfs:subClassOf rdf:resource="&ewe;Channel"/>
</owl:Class>
```

2.3 DrEWE

DrEWE is an intelligent platform for task automation. It works in a similar way to IFTTT, Zapier or CloudWork; but it provides with a important difference: the semantic approach.

[9] http://spinrdf.org/

It uses the new ontology -or standardized schema- EWE. The Drools Expert[10] rule engine to process low level events created by devices and sensors, and create high level events -at the level of the web services-. SPIN SPARQL syntax to process the high level events and order actions at high level. GSN[11] middleware as an Event Network. Besides, DrEWE incorporates a huge variety of scripts and application that generates event and process actions, such as Raspberry Pi[12] scripts or node.js[13] applications.

DrEWE sets an example of viability of EWE ontology, and has interoperability between all the EWE's platforms. This offers the possibility of processing rules not matter the source, or even a huge semantic database that would provide all the advantages from Big Data.

Besides, DrEWE is able to process complex events. This feature allows the 'event' part of the rule to have temporal reasoning. One example of that would be: "When a meeting is scheduled, if the correspondingattendees enter their Id cards at the entrance during ten minutes before the start time: generate an event".

2.3.1 DrEWE modules

DrEWE consists of five main modules, each one carrying out one function:

1. *GCalendar.* A Node.js module that simplifies the use of RESTful Google Calendar API without any interaction with the user: it retrieves all events on a given calendar and sends them to a GSN server. Before sending the events, it checks if it has already been added to the server.

2. *Berries.* This module is a group of Raspberry scripts which communicate with GSN and/or SPIN to produce events, make actions and handle requests. Among the Berries functions, it can be found: generate events when somebody inserts the Id card at the entrance of the laboratory; take photos with a Raspberry camera board periodically or under request and serve these images via HTTP; Generate periodically events with the current light value -such as a lamp-.

3. *GSN.* GSN is a middleware for rapid deployment and integration of heterogeneous wireless sensor networks. In DrEWE, it has been implemented as a Event Network. One remarkable feature is the timed database: GSN provides a timestamp column

[10] https://www.jboss.org/drools/
[11] http://sourceforge.net/projects/gsn/
[12] http://www.raspberrypi.org/
[13] http://nodejs.org/

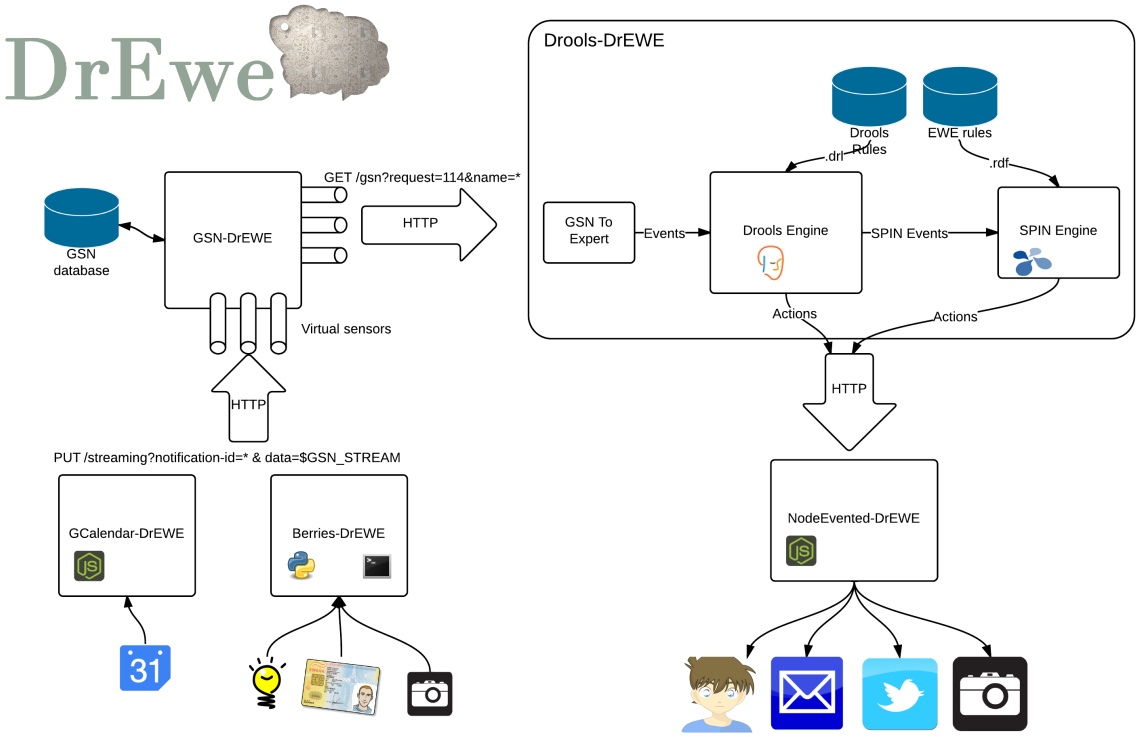

Figure 2.3: DrEWE general architecture

for each type of data that it receives. This timestamp is used by the following modules (the rule engines) for complex event reasoning in order to provide time-based reasoning.

4. *Drools*. This is the module in charge of processing events and ordering actions. It consists of two different rule engines that work together: the Drools-based engine and the SPIN-based engine. The Drools engine is a well known rule engine that provides Complex Event Processing. The SPIN engine comes with a SPARQL inference module. The SPARQL inferences are run over a semantic model, and the inferenced triples are put into a new model. All these processes are implemented using the RDF Apache Jena Software[14]. This is the most important module for the Wool project.

5. *NodeEvented*. This Node.js module is in charge of generating events and processing actions. It is connected with the Drools module and the GSN module.

[14]http://jena.apache.org/

2.4 Knockout

Knockout[15] is a JavaScript library that helps the process of creating responsive display and editor user interfaces with a underlying data model. Knockout is recommended for cases in which the User Interface (UI) updates dynamically, helping to implementing it more simply and maintainably.

The dependency tracking is one of the most important and attractive features of Knockout. It automatically updates the right parts of the UI whenever the data model changes, making the correspondence between graphic interface and data model -the logic- easy. The binding of the UI to the data model is simple, and it allows developers to construct complex and dynamic UIs using nested binding contexts.

Besides, using Knockout is possible to extends its functionalities by implementing custom behaviours as new declarative bindings for reuse. Knockout works with any server or client-side technology; as such as it can be added on top of a web application without requiring major architectural changes.

2.5 FuelPHP

FuelPHP[16] is a MVC (Model-View-Controller) framework that has support to the Hierarchical MVC pattern (HMVC). Besides, this framework adds the concept of ViewModel[17], which offers the option to add a layer between the Controller and the View. Fuel also supports a router based approach where it is possible to route directly to a closure which process an input URI, making the closure the controller and giving it control of further execution.

Almost every class in FuelPHP's core package can be extended without modifying the code where it is used. It is possible to package more functionality into packages that extend or replace the FuelPHP core, and allows us to keep the application modular by dividing it into application modules.

Security is a great feature inside FuelPHP. Among the security functions there are: input filtering, URI filtering, output encoding, token protection, SQL injection prevention, a secure authentication framework. This authentication framework sets a default interface drivers to make use of. Inside FuelPHP two sets of drivers are included. The most useful is

[15] http://knockoutjs.com/
[16] http://fuelphp.com/
[17] http://fuelphp.com/docs/general/viewmodels.html

the so-called *Ormauth* framework, which is ORM based. This driver provides a full-featured Access Control List system with permissions at user, group and role level. Moreover, that Auth framework has integrated Opauth library, allowing us to integrate authentication for social media interaction.

One important FuelPHP feature is the integration with the command line utility *oil*. This utility is designed to help speed up development, increase efficiency and assist testing and debugging. It allows developers code generation, scaffolding and admin generation. It also allows us to run database migrations, and facilitates the interactive debugging. Besides, it offers the use of *tasks*: activities such as importing data or other background operations.

For rapid coding, FuelPHP provides with some base classes which implements several services: *Controller_Template*, allows us page templating to the controller; *Controller_Rest*, provides a RESTful API; *Controller_Hybrid*, combine two features into a single controller; *Model_Crud*, provides with all methods for CRUD operations.

Lastly, FuelPHP offers a ORM for worl with different databases, make CRUD operations and manage relations between rows. The main characteristics of this module are: relationship types -belongs to, has one, has many, many many-; nested relations; entity attribute-value model implementation; the use of observers to process the objects instances -i.e. to validate before saving or to auto-update a property .

2.6 MongoDB

MongoDB[18] is an open-source document database written in C++. MongoDB provides high performance, high availability, and easy scalability. MongoDB is a JSON-style[19] document based database: it allows us to map the stored documents to programming language data types. The documents can be embedded and exported into arrays. The schema of the database is dynamic, which facilitates the polymorphism of the data.

The data model is as follows: a MongoDB deployment hosts a number of databases. A database holds a set of collections. A collection holds a set of documents. A document is a set of key-value pairs. As the documents have a dynamic schema, the documents in the same collection do not require to have the same set of fields or structure; besides, common fields in a document of a collection may hold different types of data.

Queries in MongoDB provide a set of operators to define how the find()-basic request-

[18] http://www.mongodb.org/
[19] http://bsonspec.org/

ing method in MongoDB- method selects documents from a collection based on a query specification document.

Although MongoDB supports single-instance operation, production MongoDB deployments are distributed by default. Replica sets provide high performance replication with automated fail-over, while shared clusters make it possible to partition large data sets over many machines transparently to the users. MongoDB users combine replica sets and shared clusters to provide high levels redundancy for large data sets transparently for applications.

CHAPTER 3

Requirement Analysis

3.1 Introduction

The result of this chapter is a requirement analysis which will enable a more complete vision of the system to be developed. Besides, this chapter also helps the reader in the process of understanding the purpose of the Wool project.

The analysis will use the Unified Modeling Language (UML)[1]. This language allows us to specify, build and document a software system using graphical language.

This analysis is important when understanding a system, but also when designing a software system. Because of this, we present this analysis chapter.

That being said, the aim of this chapter is not to cover all the system requirements, or all the Wool functionalities. In this chapter the analysis will be made briefly; it is not a thorough analysis of the Wool system.

[1] http://www.uml.org/

3.2 Use cases

This section identifies the main use cases of the Wool system. This helps to obtain the specifications of the uses of the system, and therefore defines a list of requisites to match.

In 3.2.1, a list of the main actors will be presented and a UML diagram representing all the actors participating in the different use cases in 3.2.2. This representation allows us to identify the actors that interact with the system, as well as the interconnection between them. Then, several subsections -3.2.2.1 and 3.2.2.2- will show the sequence diagrams of some of the use cases. The sequence diagrams are developed following the UML language.

3.2.1 System actors

Identifying the actors of the system is the first step to take into consideration when an analysis of a system is being made. The actors of the Wool system are:

User. Final user of the system, and the main actor. It accesses the Wool system aiming to edit rules using the available offered channels. This actor also manages the rules, creating and deleting them. It deploys the rules into the SPIN Motor.

Admin. Administrator of Wool, in charge of managing the system users. It is able to create and delete Wool users, and manages the rules deployed into the SPIN Motor.

DrEWE. This is a secondary actor. It collects the rules that are created and deploys them into the SPIN Motor. It is in charge of contacting with the outer services.

3.2.2 Use cases

Next a use case diagram is presented. In this graphic it is shown the main Wool use case, and the interconnection with the actors of the system.

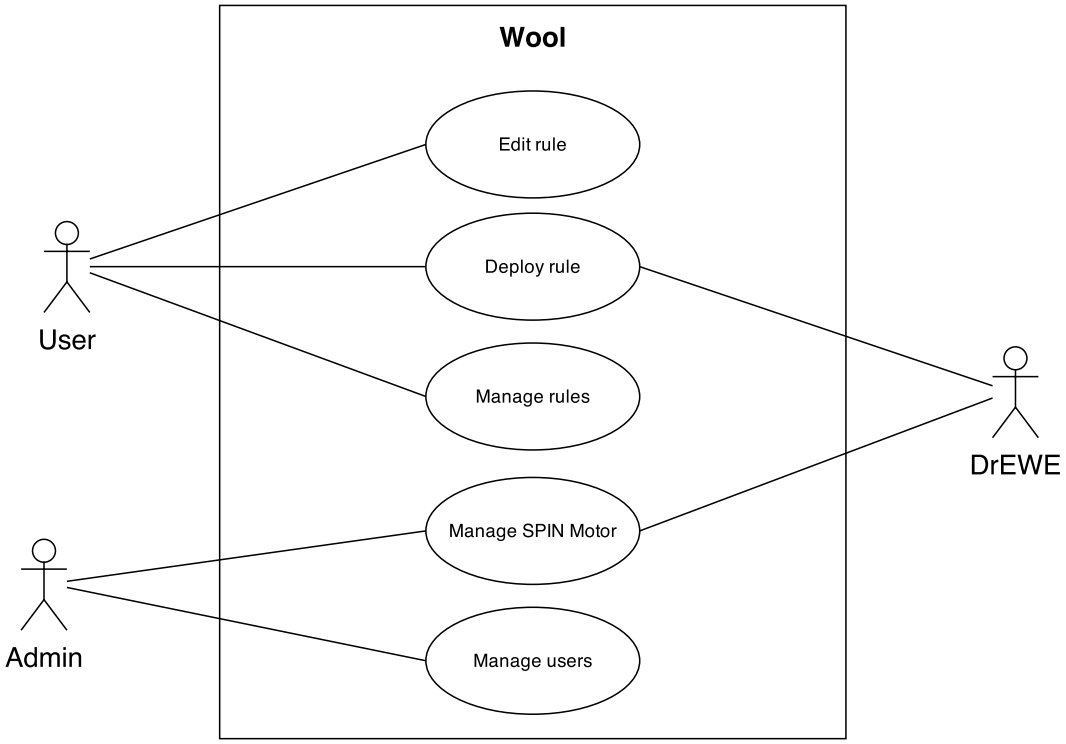

Figure 3.1: Use case diagram

3.2.2.1 Edit rule

In this use case the only actor is the User. This actor accesses the Rule Editor tool via web, and then follows the edition process. In this process Rule Editor connects several times with Rule Manager, as can be seen in figure 3.2. These communications are used for obtaining data about channels, spaces and rules. The method *getChannels()* makes a request to the available channels for this particular user. The other methods work in the same way: *getSpaces()* is for obtaining the information about the available spaces; and *getRule()* is used to obtain the rule to be edited. Every data is obtained form the Rule Manager persistence layer, which is also reflected in the figure.

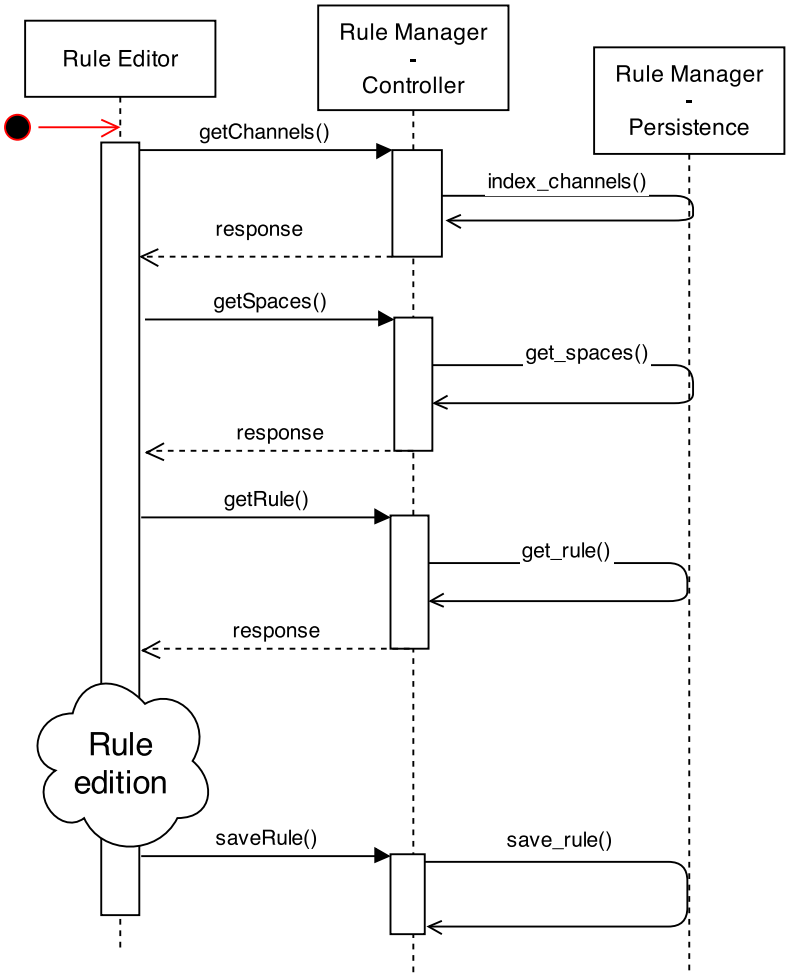

Figure 3.2: Edit Rule case use

3.2.2.2 Deploy rule

The actor of this use case is the User. In this case, the user accesses Rule Manager with the intention of deploying into the SPIN Motor a previously edited rule. This rule is stored into the Rule Manager persistence module. When a rule is deployed Rule Manager accesses to its persistence module several times, obtaining data regaining the rule to be deployed and the template to be used in the deployment process, as can be seen in figure 3.3. Once the rule is composed using the template, it is saved into the Rule Manager persistence, and then sent to the SPIN Motor in DrEWE. Once the rule is successfully deployed, this process terminates.

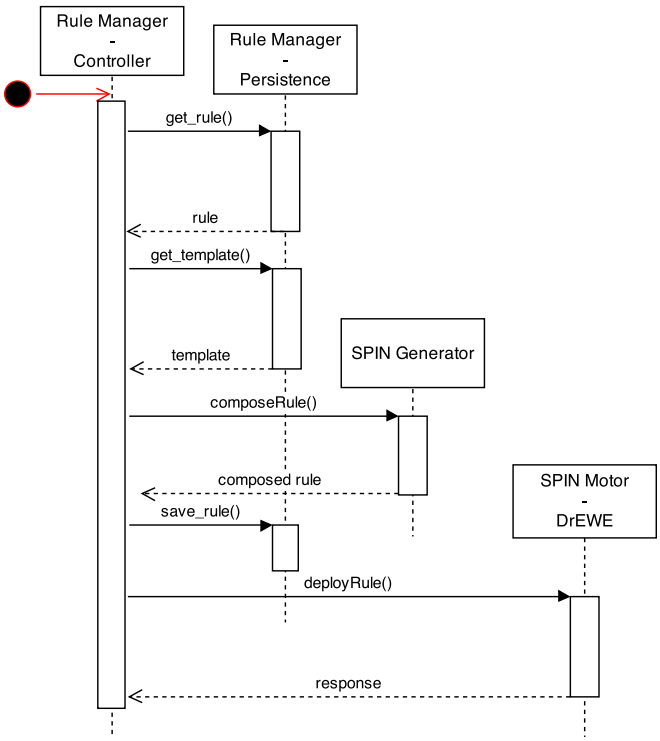

Figure 3.3: Deploy Rule case use

and the us

CHAPTER 4

Architecture

4.1 Introduction

In this chapter, we cover the design phase of this project, as well as implementation details involving its architecture. Firstly, we present an overview of the project, divided into several modules. This is intended to offer the reader a general view of this project architecture. After that, we present each module separately and in much more depth.

The main purpose of this project is to obtain an automation rule editor in which users can configure and adapt their preferences about Internet services as well as sensor-space devices. First we need a tool for editing the automation rules: we call this tool **Rule Editor**, which is the main core of this project. Rule Editor is a web application, and is implemented using client side technology. For this reason, a server side technology was needed. In addition, Rule Editor needs to rely on rules persistence, user authentication and differentiation, etc. These reasons make an management module necessary. This module is called **Rule Manager**. Rule Manager is implemented as server side technology.

With all the modules above, we have a platform that allows us editing and managing automation rules by itself. Nevertheless, it is not able to extends its possibilities to the outer: such as Internet services or sensor devices. This is the reason why this project

extends the functionalities of DrEWE: to make all these rule editions fruitful for a network of services and sensors outside Rule Editor. For these reasons **SPIN Generator** is created. This module allows the user to export his rules to SPIN-SPARQL format, that will be able to be interpreted by automation tasks platforms such as DrEWE. With this, the user can create editable rules that will be fully compatible with DrEWE, and also EWE ontology.

A diagram of the general architecture is shown in Figure 4.1. Each module will be detailed in the following sections.

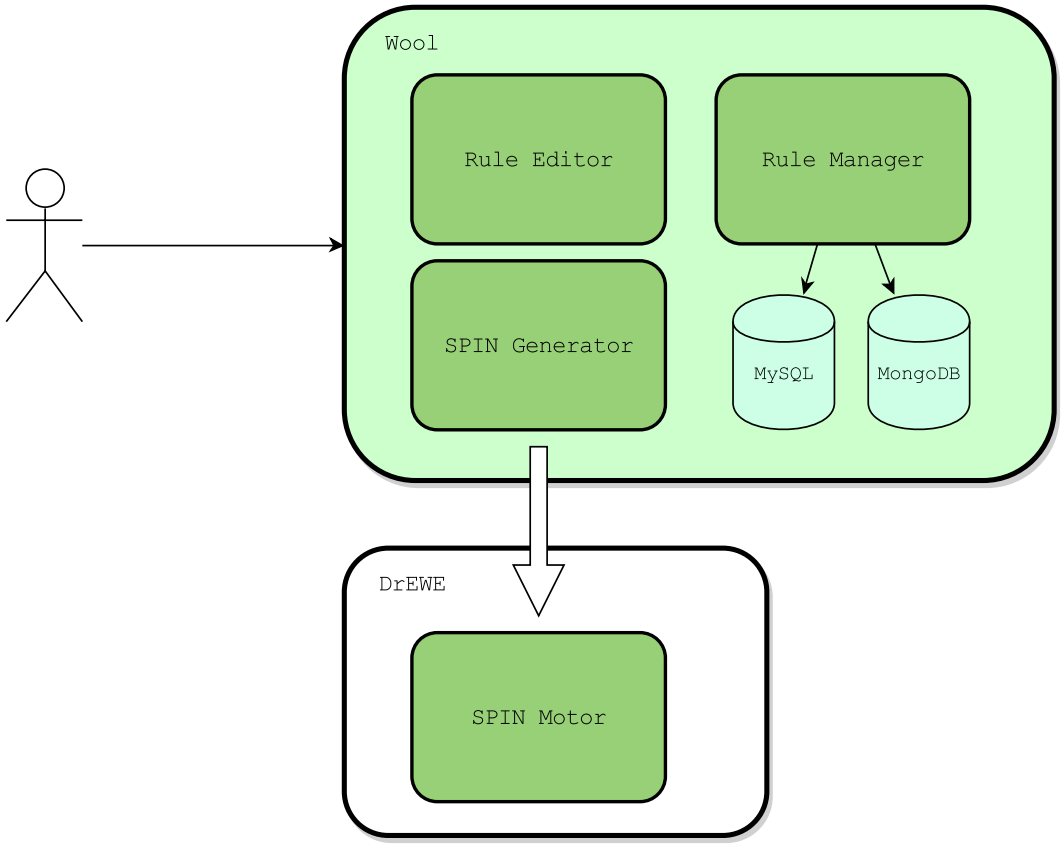

Figure 4.1: General Architecture

4.2 General overview

The core of this project is the **Rule Editor Server**. As it can be seen in 4.1, Wool is composed by three different modules: *Rule Editor*, *Rule Manager*, and *SPIN Generator*. All these modules are differentiated following a functional criteria. The interconnection of these modules into a major functionality is represented by the following flow diagram:

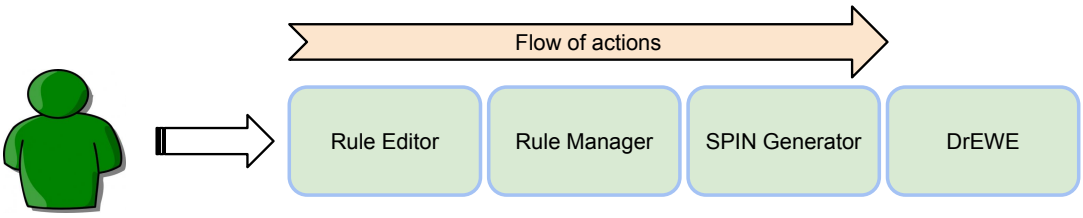

Figure 4.2: Modules Flow Diagram

In the figure 4.2 can be seen that the user interacts with the *Rule Editor* module. That is, the user creates or edits an automation rule using the web editor. The flow represented above is described as follows.

1. *Rule Editor*. The user interacts with the Rule Editor web interface for automation rule editing. This interface is primarily graphic, icon based. The main function of this module is to connect two channels of events or actions, and create a rule that describes this connection. By using the interface, the user will have created an automation rule of this type. Once created, the user may finish the edition process. Then, Rule Editor sends the correspondent automation rule to the server.

2. *Rule Manager*. Once the rule has been created and sent to the server, Rule Manager receives it. The main function of this module is to persist the rules. Each rule will be stored into a database, with all the information related to that rule, including the user who has created it. With this, each user has available all the rules he or she has created for edition or deletion.

3. *SPIN Generator*. Once an automation rule has been created and saved, this module is in charge of exporting the data from to rules to a EWE compatible format: JSON-LD. This module creates rule descriptions saved into JSON-LD format using SPARQL requests templates. These documents are sent to DrEWE, where they will be processed.

4. *SPIN Motor*. The rules which have been formatted into SPARQL sentences are sent to this engine, that will test the rule to check its correctness. Once this is completed this module will deploy the rule, allowing its use.

4.3 Rule Editor

Rule Editor is the core of this project, one of the most important modules. This module provides an interface for automation rules edition. During the whole project, an effort has been made to elaborate a graphical interface, based on icons and 'drag and drop' actions. These are intended to make easy and fast the process of creating or editing an event driven automation rule.

A channel, represented by an icon, is a source of the so called events or actions. For example, '*Twitter*'[1] is a channel from the Internet services field. This channel provides events such as '*I get a new follower*'[2] that can trigger actions. An example of an action in this '*Twitter*' channel is '*Post a tweet*'[3]. Being this way it is possible to connect an event to an action through the structure "***If this then that***", creating a automation rule. In that structure, *this* would be the event, and *that* would be the action. Referring to the example above, the rule resulting from the action and event is "*If I get a new follower then post a tweet*". These type of rules are the automation rules that Rule Editor is intended to create and edit.

4.3.1 Graphic interface

As it is said, Rule Editor interface is icon-based. An icon represents a channel of events or actions. In addition, there are two containers into the interface of Rule Editor, which the user can utilize for selecting the channel that will provides events (left channel), and the channel which will provide actions (right channels). This configuration matches the structure *If this then that* previously presented: 'If this' (left channel); 'then that' (right channel).

With the Rule Editor interface, a user can drag a channel-icon, and drop it onto one container. Then, navigating through several menus that will be presented, the user can select the events or actions which will shape the automation rule.

An example of this interface is presented in figure 4.3.

[1] https://ifttt.com/twitter
[2] https://ifttt.com/channels/twitter/triggers/96-new-follower
[3] https://ifttt.com/channels/twitter/actions/1-post-a-tweet

Figure 4.3: Rule Editor graphical interface

All this logic is implemented using client-side technologies: JavaScript, CSS, HTML5, jQuery[4] and Knockout. Knockout is specially useful in this project considering that the user can make actions like 'drag and drop' a channel, and the interface must respond in real time. Search functions and language changes are implemented using Knockout too.

4.3.2 Data structure

Just as the user is creating a rule, Rule Editor has to store all the information relating this process. For example, Rule Editor has to maintain a record of which channel will provide the event, and which event is selected between the several that a channel has.

For fulfilling this goal, a structure of objects is used for storing all this data. These objects are observables (Knockout provides this type of objects). It is important to add that this structure will be sent to the server, in order of communicate the created rule to Rule Manager.

A schema of this structure is detailed next. First, the data involving the event channel; and then the data of the action channel.

[4]http://jquery.com/

Listing 4.1: Rule template

```
{
  "created_at": "",
  "deployed": false,
  "edited_ruleId": "",
  "ewe_spin": "",
  "when": {
    "@id": "",
    "dcterms:title": "",
    "dcterms:description": "",
    "@type": "",
    "from_channel": "",
    "ewe:hasInputParameter": {
      "@type": "",
      "dcterms:title": "",
      "xsd:type": "",
      "dcterms:description": ""
    },
    "ewe:hasOutputParameter": [
      {
        "@type": "",
        "ewe:Property": "",
        "dcterms:title": ""
      }
    ],
    "inputform": []
  },
  "whenOutputs": [],
  "do": {
    "@id": "",
    "dcterms:title": "",
    "dcterms:description": "",
    "@type": "",
    "from_channel": "",
    "ewe:hasOutputParameter": [],
    "inputform": "",
    "ewe:hasInputParameter": [
      {
        "@type": "",
        "dcterms:title": "",
        "ewe:Property": ""
      }
    ]
  },
  "user": ""
}
```

4.3.3 Endpoint communication

Rule Editor is a client-side application, so all the operations made in it must be supported by the endpoint: persisting and managing the data Rule Editor generates. For this reason, Rule Editor communicates with the endpoint -the Wool Server- for sending the edited rules; as also receiving the available channels or the already created rules.

This communication is implemented using JSON format and AJAX protocol[5]. JSON format is specially useful when communicating the data structure of Rule Editor, since it is using Knockout observables, which are basically JavaScript objects. AJAX allows us to execute different functions depending on the state of the link with the endpoint. These two tools provide Rule Editor with a useful interface for communicating with the Wool Server.

4.4 Rule Manager

The Wool platform works with automation rules: creating, editing, removing them. These actions are, in the end, the same: management of the rules. In addition, these rules need the channels; and the users that use them. For fulfilling all these functionalities we have developed Rule Manager. This module, enclosed into the Wool Server, serves for these purposes. Rule Manager acts as a controller into the Wool platform, coordinating the actions and data from Rule Editor, SPIN Generator and the DrEWE SPIN Module.

Rule Manager allows the user to manage the rules which have been created or edited through a graphic web interface. Besides, with Rule Manager, a user can view the channels that Wool has available, and check the characteristics that each channel has. This module provides an access to all the rest of the functionalities of Wool.

Rule Manager has been implemented with PHP using the FuelPHP framework, following the MVC pattern. This has been very important during the design and development since the data that is managed comes from different sources: users, channels and rules are dealt differently, and the necessity of isolate the data from the rest of the process arises. More specifically, Rule Manager makes use of two different databases: SQL and NoSQL ones. These are two radically different type of databases, so the MVC pattern fits perfectly in this design.

This module offers two different interfaces: a graphical HTML-based interface for users using a web browser, and a REST interface which serves content depending on the request.

[5] http://api.jquery.com/category/ajax/

4.4.1 Rule Manager architecture

As it has been explained previously, Rule Manager follows the MVC pattern. This pattern has a specific architecture, that in this project is understood as a three layered architecture: Model, View, Controller. Next, a schema describing this architecture and the Rule Manager components is shown.

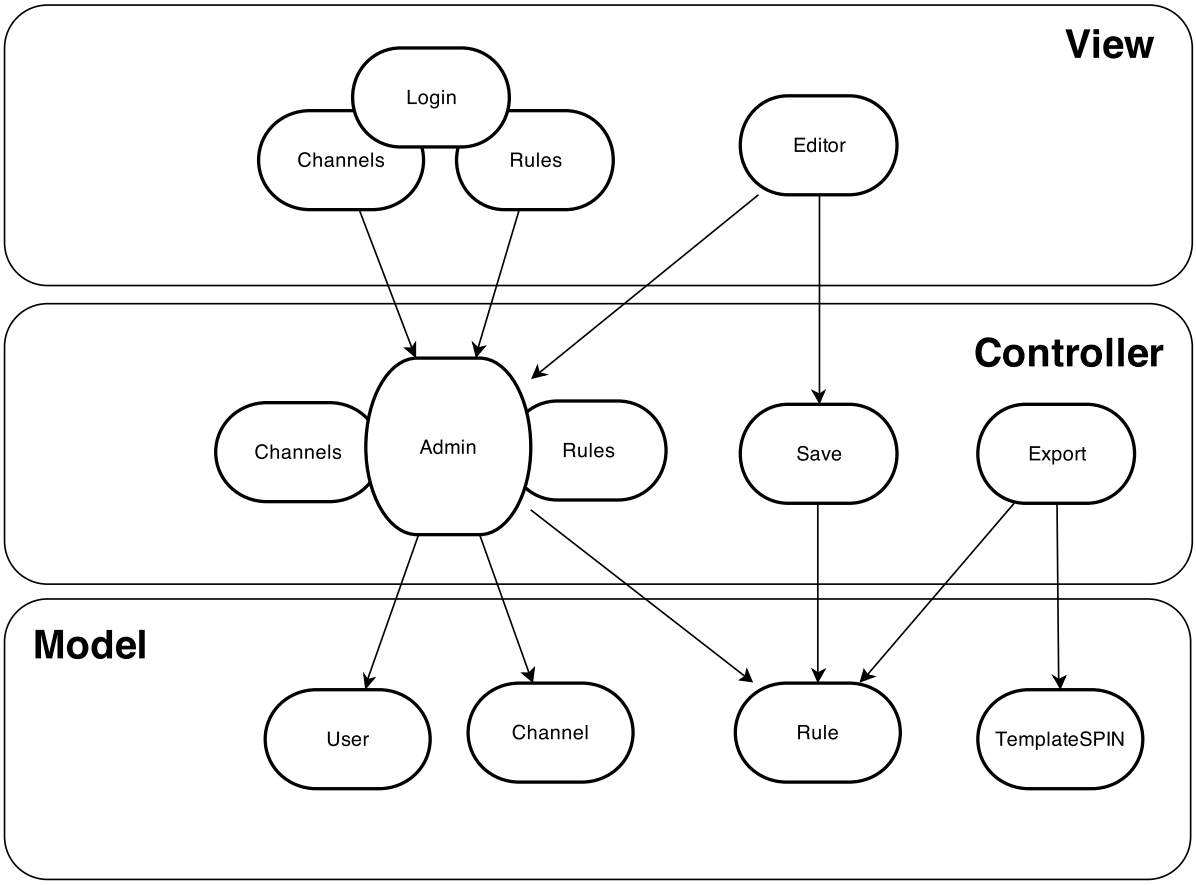

Figure 4.4: Rule Manager architecture

Usually, the user accesses the pages belonging to the *View*. These pages have all the content the user needs to access all the functionalities. All the pages have dynamic content which is processed by the *Controller*. The *Controller* manages all the data coming from the *View* and the *Model*. Besides, the *Controller* processes the requests sent from the user: as much from requests made to the REST interface as from the *View* pages requests.

The *Model* offers an abstract information layer to the *Controller*. The *Model* accesses the information under the *Controller* request, creating a connection to the corresponding database and extracting the information and returning it to the *Controller*. In this way Rule Manager offers the possibility of changing the Database implementation, and conserving the

Model untouched.

4.4.2 SQL database

A SQL database is used for maintaining users persistence. This database is controlled by the default SQL driver, provided by FuelPHP; and follows the ORM technique. This technique allows us to map the database tables to objects; besides, it is possible to stablish relations between these objects. ORM makes easier the users management, dealing with the users as objects. A schema representing the structure of the most important table inside this database is shown next.

Listing 4.2: Users table structure

```
mysql> describe users;
+----------------+------------------+------+-----+---------+----------------+
| Field          | Type             | Null | Key | Default | Extra          |
+----------------+------------------+------+-----+---------+----------------+
| id             | int(11) unsigned | NO   | PRI | NULL    | auto_increment |
| username       | varchar(50)      | NO   |     | NULL    |                |
| password       | varchar(255)     | NO   |     | NULL    |                |
| group          | int(11)          | NO   |     | NULL    |                |
| email          | varchar(255)     | NO   |     | NULL    |                |
| last_login     | int(11)          | NO   |     | NULL    |                |
| login_hash     | varchar(255)     | NO   |     | NULL    |                |
| profile_fields | text             | NO   |     | NULL    |                |
| created_at     | int(11)          | YES  |     | NULL    |                |
| updated_at     | int(11)          | YES  |     | NULL    |                |
+----------------+------------------+------+-----+---------+----------------+
```

4.4.3 Mongo database

The NoSQL database is a MongoDB database. This database, which is object-oriented, is used for the channels and rules persistence; besides, it is used by the SPIN Generator module. MongoDB persists the data using JSON format documents, which allows us to store all the data involving rules and channls using JSON and JSON-LD formats. In Rule Editor, the structure of data uses JSON. This has the advantage that Rule Manager makes the management of this data in a fast and natural way.

This last database could have been made using SQL, but because of the difficulty of the management between JSON-data and SQL static tables, NoSQL-type database was chosen. In addition, during the design phase the use of an SQL database was considered

seriously; nevertheless, the process of transforming a JSON document into data stored in static interconnected tables is difficult compared to the easiness of the process -during the development state as well as the database exploiting phase- using Mongo, which is document oriented.

In spite of this last subject, the benefit of utilizing SQL or NoSQL database for this project only could be measured testing and comparing between the two ones.

The Mongo database is organized into collections, each one serving for one purpose. The specific collections are:

- *Rules*. This collection stores the rules which have been created. Each document inside the collection includes a rule. Each rule has an identification that connects with the rule author. Besides, the rules have references to the channels they come from.
- *Channels*. Collection where all the channels are stored. Each document of this collection represents a channel.
- *SPIN templates*. In this collection the SPIN templates are stored. This collection is used by the SPIN Generator module, which will be explained in the section 4.5.
- *Spaces*. In this collection, the channel spaces are stored. Each channel belongs to one space (maybe two).

4.5 SPIN Generator

The created and edited rules do not remain into the Rule Manager persistence module only and exclusively, but are also exported into the SPIN Motor module, where they will be inserted. This is what is called *rule deployment*. For this process to be successful, the rule to be deployed needs to be expressed into the SPARQL format. Basically, this is the function of SPIN Generator: transform the rules expressed in a JSON format to rules in SPARQL format. This transformation is called *rule composition*.

This module is implemented in PHP. SPIN Generator bases the rule composition in a query substitution technique as of template using. The process of obtaining these templates is detailed in the Appendix A of this document.

SPIN Generator is implemented as a REST service. When a certain request is received -this request must include a rule identifier-, this module extracts the specific rule from the Rule Manager persistence module. Once obtained, SPIN Generator finds the number

of parameters the rule needs to reflect into the SPARQL query. Using this information, SPIN Generator extracts the corresponding template -once again from the Rule Manager persistence module-. Applying the method specified in the appendix A, this module creates the SPARQL query that contains the EWE rule.

The last objective of the SPIN Generator module is to communicate the SPARQL rule to the SPIN Motor. For this, this module creates a request containing the rule as a request parameter. It is worth mentioning that this a POST request -thus the request parameters are specified in the request body, and not in the request URL- since it is not advisable to include something as large as a SPARQL rule into the URL of a GET request. How the SPIN Motor module handles this request is explained in the section 4.6.

In the figure 4.5 a diagram of all the process is presented.

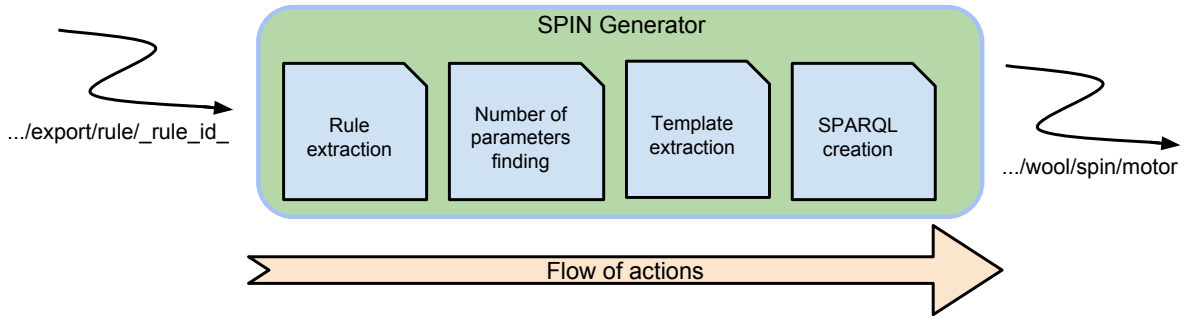

Figure 4.5: SPIN Generator flow diagram

4.6 SPIN Motor

Once composed, the SPIN Generator module communicates the SPARQL rule to this module: the SPIN Motor. The SPIN Motor module belongs to DrEWE, but not completely. SPIN Motor has been implemented starting from zero for the Wool project, and the inserted into DrEWE. In this way, SPIN Motor improves some of the DrEWE functionalities, adapting them to the Wool paradigm.

SPIN Motor is implemented using Java and JavaEE. It bases its functionalities in the Apache Jena implementation[6]. This module is based in a ontology model, which uses the EWE ontology.

SPIN Motor loads the rules that receives from the SPIN Generator module to the ontology model. Once loaded, this module is capable of receiving events -identified by an URI-,

[6]https://jena.apache.org/

process them, and act accordingly to the rules triggering the corresponding actions. These actions, as was mentioned in A.4.4, are identified by a unique URI. This process is showed in the figure 4.6.

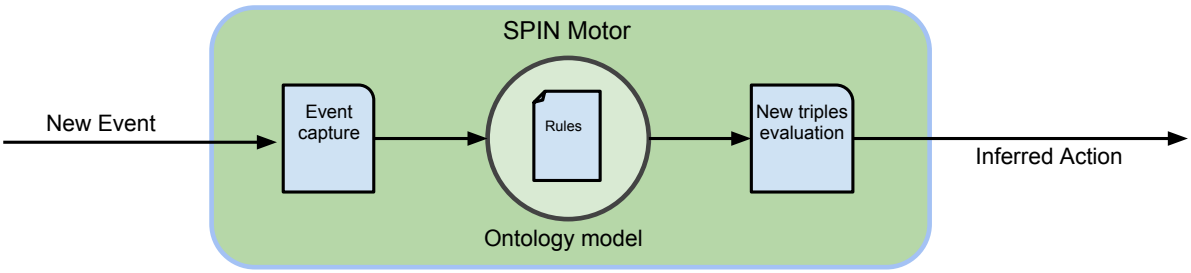

Figure 4.6: SPIN Motor process diagram

When a new event is received, SPIN Motor captures it and creates a new event object. This object is an ontology model resource. This resource is inserted into the ontology model. Then, SPIN Motor runs the ontology model inferences. This process combines the loaded rules and the inserted events, and draws conclusions from them. These conclusions are represented by triples. The resulting triples are actions representations: they have the actions URIs, and their properties. In this way, the actions are inferred -triggered-: the conclusions from the inserted events and loaded rules of the ontology model represent these actions.

A important aspect is the management of the resources inside the ontology model. In SPIN Motor, the rules are always loaded: once a rule is loaded into the model, it is kept there. Nevertheless, the events are treated in a different way: once a event has been inserted into the ontology model and the inferences have been made, this event in removed from the model. This is because the EWE events have no temporal reasoning: a event occurs, and then disappears.

In the same way, the action triples are resources of the ontology model. An Action has no temporal reasoning neither: it is inferred and the removed from the ontology model.

CHAPTER 5

Case study

5.1 Introduction

In this chapter we are going to describe a selected use case. This description will cover the main Wool features, and its main purpose is to completely understand the functionalities of Wool, and how to use it.

The actor of this case is the user. The goal of the user is to create a automation rule, which is generated from two channels, and to deploy it into the SPIN context. In this way, the user creates a rule that will automate the events and actions of the selected services once activated.

5.2 Rule edition

The first step to take is the authentication. The user must fill the form showed in figure 5.1 to be authenticated in the system.

Once authenticated, the user may navigate to the Rules page. This page shows a list of the rules created by the user. In this page the user can edit a created rule or create a new

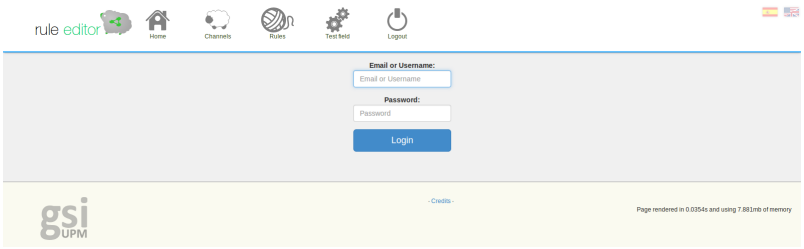

Figure 5.1: Wool authentication page

one. In this case, the user creates a new rule. For this, the user may press the button "Add new rule". The rule list is showed in figure 5.2.

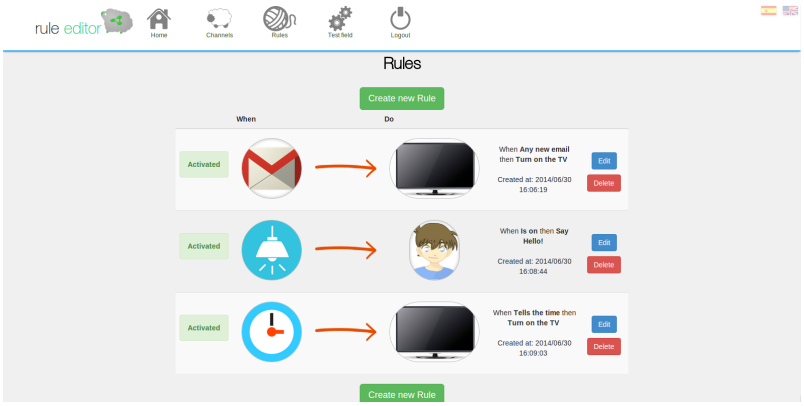

Figure 5.2: Rule list

When the user presses the button, it is redirected to the Rule Editor graphical interface. The interface is designed to be easy to use and friendly to the user. The process of creating a rule is indicated step by step in the same page. This can be seen in the figure 5.3.

First, the user has to select between the different available spaces. These spaces classify the channels, and each channel is in one space. This allows the user to differentiate the events and actions that each channel has: a channel that is classified into Web Service will trigger actions regarding the web -send emails, send text messages-; while a channel that is classified into the GSI Laboratory will trigger actions regarding the context of this laboratory -turn on lights, take a shot with a camera, send a message through the GSIBot-.

Second, the user selects the channel that will represent the event. This selection is made by dragging the box representing the corresponding channel and dropping it into the available container. After this, a modal view appears. This view presents a so-called "trigger selection". Basically, it is a menu which allows the user to select the event that will trigger the rule. Rule Editor offers the user the events regarding the selected event channel. This is showed in the figure 5.4.

Figure 5.3: Channel boxes, spaces tags, event and action containers

Figure 5.4: Event selection

Once the event-part of the rule is selected and configured, the user proceeds with the action-part, that will be enabled now. The action selection is similar to the event selection. The user has to drag and drop a action box into the action container -right container-. In the same way that the event, a modal view appears. In this case, it offers the actions available for the selected channel. Once the user has selected the desired action to be triggered, another modal view appears. This second view allows the user to select the output of the

event to connect with the input of the action. This selection allows the rule to map these two parameters. This view is the showed in the figure 5.5.

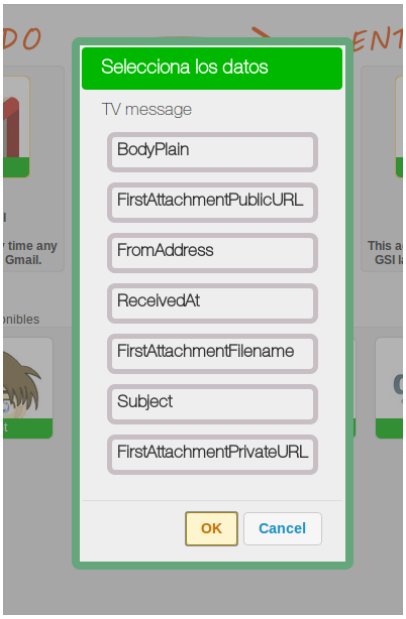

Figure 5.5: Event and action parameters mapping selection

After these configurations have been made, the user can save the rule pressing the "Save Rule" button. Once pressed, Rule Editor collects the information about the created rule and sends it to the Rule Manager module, where it is persisted. Then, the user can go back to the rule list, or create another rule. With this, the process of creating a rule is finished.

5.3 Rule deployment

Once the user has created the rule, it is stored into the Rule Manager persistence module. Nevertheless, this rule is not functional: it is not into the SPIN Motor module. For this reason, the user has to activate the rule. In this section it is considered that the user is authenticated into the Wool system, and that a rule is created and stored in the Rule Manager persistence module.

For deploying the rule, first the user navigates to the Wool rule list. In this list, the created and deployed rules are represented, but also the created and not deployed rules are in it. The deployed rules are marked by a tag, as in the not deployed rules. These different tags allows the user to know if a rule is deployed, and in case of not, deploy it into the SPIN Motor module. This is shown in the figure 5.6.

For an user, deploying a rule is very easy. Once created the rule, the user presses the

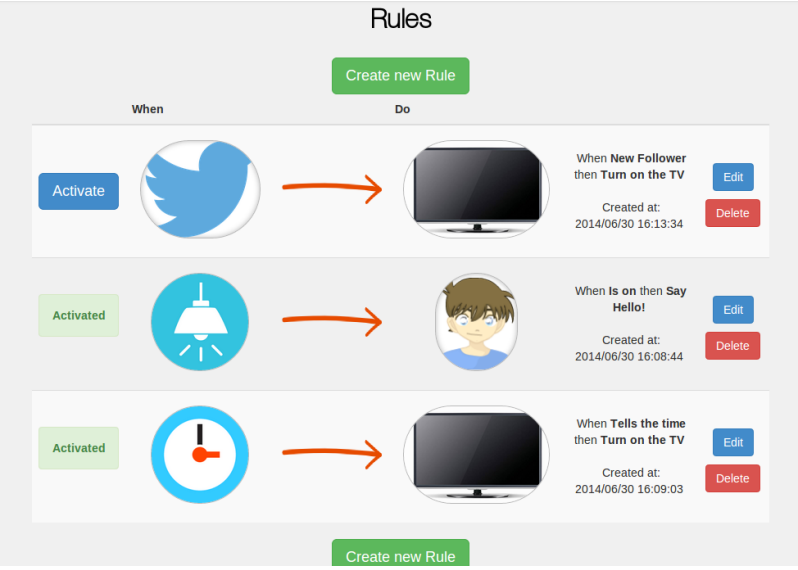

Figure 5.6: Rule list: activated and not activated rules

"Activate" button. This button deploys the rule into the SPIN Motor. In case of success, the rule is marked as *Activated*.

5.4 Rule execution

When the created rule has been deployed, it is into the SPIN Motor. Thus, when an event of a certain characteristics launches into the motor, the rule can trigger the defined action. This behaviour can be observed by the user in the Wool Test Field, where we have simulated the so called ***Virtual GSI*** -vGSI-. In the Virtual GSI the user can launch the events which the deployed rules define, and observe the actions that these events cause. The vGSI interface is shown in the figure 5.7.

For executing these rules, the user presses the boxes that represents the events to launch. These boxes will appear when a rule has been created and deployed into the SPIN Motor. Once pressed the event box, the vGSI logic will communicate the launched to the Wool system and will wait for a response. This event is inserted into the SPIN Motor, and then the actions are inferred. These actions are communicated to the vGSI, that will present them to the user. This behaviour can be observed in the figure 5.8.

In the EWE ontology the events and actions have not temporal reasoning. A event launches, and then disappears. In this same way occurs with the actions. In vGSI we have represented this behaviour by making disappear the actions when they have been presented

Figure 5.7: Virtual GSI interface

to the user once a lapse of time have passed -a few seconds-.

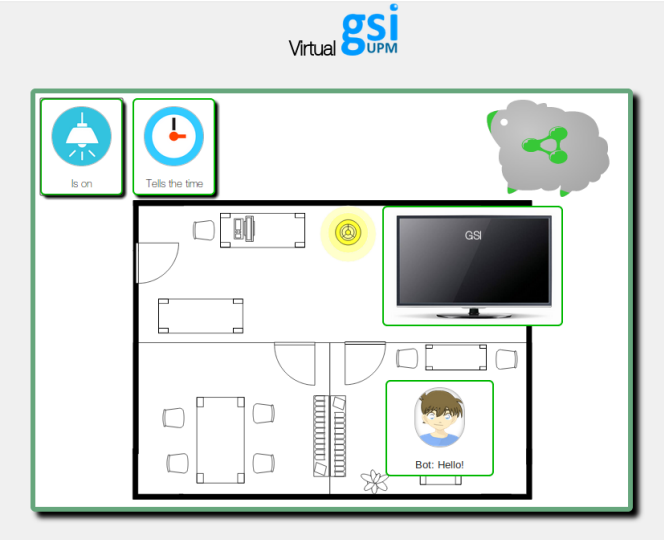

Figure 5.8: Virtual GSI triggered actions

In the figure 5.7 and figure 5.8 is shown a map of the GSI laboratory. This map represents the location of some devices that could be located in the GSI laboratory. These devices can launch events: the lamp can launch an event when it is turned on, and the eDNI[1] reader can launch an event when a DNI is inserted. Besides, there are devices located in the laboratory that can execute actions: the TV can show messages coming from an event, and the GSI bot[2] can greet us by showing messages.

[1]http://www.dnielectronico.es/
[2]https://github.com/gsi-upm/calista-bot

Nevertheless, this simulator can also launch events coming from Internet services. For example, it can be simulated that when a new follower in twitter event is launched, the TV shows a message indicating this fact.

CHAPTER 6

Conclusions and future work

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

6.1 Conclusions

In this project we have created a tool that allows us to edit event driven automation rules using a graphical interface, making easier the creation of these rules. In addition, this project allows the user to manage these rules: create, edit, remove them, as well as deploy them into a semantic engine that will work according with these rules. With this, users can coordinate different resources -in the web or located in spaces- to work for them in the way they desire.

This project follows the EWE Ontology. During the development of the Wool project, several requirements have arisen that have required modifications in EWE. This has allowed us to improve EWE, and to use it in the best way possible.

Wool is supported by DrEWE, but in some cases it has been necessary to create or improve some aspects of it. This has extended the functionalities of DrEWE.

Wool uses several different technologies, thus we have learnt these varied technologies, and how to coordinate all of them. We have used web technologies: FuelPHP, Knockout; database technologies: MySQL, MongoDB; and also semantic technologies: SPARQL, SPIN, RDF.

The division of Wool into several modules facilitates the low dependency between the different offered functionalities. Each module can be deployed independently, and it can be used with or without the another ones. This offers several possibilities when it comes time to improve Wool, or one of its modules. Besides, this has forced us to rely on software standards: the interconnection of these module is made following some of these standards.

During the whole project, we have tried to develop Wool such that it will be relatively easy to extend its functionalities: dividing the project into several modules, and using data structures and code patterns which can be extended.

6.2 Achieved goals

Create a graphical interface that allows us to edit rules One of the main requirements of this project is to develop a graphical interface that allows the user to create or edit automation rules in an easy way. This process is based in a user-friendly "drag and drop" technique. The data regarding these rules is dynamic, and is communicated to another Wool modules. The implementation if this is detailed in 4.3.

Persist and manage rules This is a necessary requirement considering that users need to store and manage their edited rules. Users can manage their rules using the Rule Manager graphical interface. In addition, through this interface users can reach all the Wool features. The details of this goal can be found in 4.4.

Create SPIN rules Once the rules have been created using Rule Editor, this rule can be transformed into a EWE rule that will be inserted into the SPIN Motor. This a big step to make Wool useful for the semantic web. This transformation is made using the templates we have defined -the extraction of these templates is detailed in A-. The EWE rules creation is detailed in 4.5.

Run a semantic motor This is what we call the SPIN Motor. The created EWE rules are loaded into this motor, allowing us to run actions that are triggered by events according to these rules. This motor is based on the EWE ontology, and is supported by DrEWE. The events are generated in the corresponding DrEWE modules, and the actions are also run in several DrEWE modules. This behaviour is detailed in 4.6.

6.3 Future work

There are several lines that can be followed to extend some of the Wool features but were not included into this project due to the time limitation. In the next points some lines of work or improvement to continue the development are presented.

- During the creation of a new rule, this process maps the event and action parameters one to one. This could be improved by mapping the parameters dynamically, and adding dynamic information to these parameters.
- Modify dynamically the resources included in the SPIN Motor. In this module, the loaded rules stay in the engine. It is possible to change this resources depending on the user necessities.
- Test the possibilities of the different databases options. As detailed in 4.4.3, the performance of a SQL or noSQL database within the Rule Manager context could be measured in order to determine the optimal configuration.
- Use the location of each channel to locate a channel in a geographic map, allowing the user to use a channel depending on the position in the map.
- Add resources to Rule Manager using a graphical interface. In Rule Manager, channels and users are added through the direct manipulation of the corresponding database.
- Automatic discovery of located channels. It could determine the channels available depending on the current position of the user.
- Enable or disable channels depending on the user permissions. Now, every user is allowed to use all the channels available.

APPENDIX A

Extraction of templates from EWE Rules using SPARQL

In this appendix the process of SPARQL templates extraction from EWE rules is detailed. This mechanism has been an important step during the development of this project, and it is of great importance when fully understanding the Wool project.

A.1 Introduction

First, we can consider as a starting point the following rule expressed in SPARQL:

```
CONSTRUCT {
    ?action a <http://gsi.dit.upm.es/ontologies/ewe/channels/ns/SendMeAnSms> .
    ?action <http://gsi.dit.upm.es/ontologies/ewe/channels/ns/Message> ?message .
}
WHERE {
    ?event a <http://gsi.dit.upm.es/ontologies/ewe/channels/ns/NewFollower> .
    ?event <http://gsi.dit.upm.es/ontologies/ewe/channels/ns/Fullname> ?name .
    BIND ( URI("http://example.org/action") as ?action )
    BIND( fn:concat(?name, " is now following you!") AS ?message )
}
```

This rule is executed when a new follower in twitter event is launched, and it sends a new SMS with the message "-New follower name- is following you!".

This query, in the WHERE statement selects an instance of the type identified by the URI 'http://gsi.dit.upm.es/ontologies/ewe/channels/ns/NewFollower' (?event) that has the property identified by 'http://gsi.dit.upm.es/ontologies/ewe/channels/ns/Fullname' and stores it into the ?name variable. In the CONSTRUCT statement this query creates a new instance of the type 'http://gsi.dit.upm.es/ontologies/ewe/channels/ns/SendMeAnSms' with the property 'http://gsi.dit.upm.es/ontologies/ewe/channels/ns/Message' that is stored into the ?message variable.

The first BIND instruction is necessary for creating new instances. A instance must be identified by an URI, and that is what this BIND achieves. These instances generation must be made into the WHERE statement. The second BIND instruction makes a string concatenation: gets the ?name variable and concatenates it with the string 'is now following you!', and stores it into the message variable.

We can see that this rule is executed when a event with a certain properties is received, and a certain action is created. Besides, a parameter mapping is made: event parameters are mapped to action parameters.

A.2 Templates for SPARQL rules generation

We call template to a generic SPARQL query with parameters that will be replaced during pre-processing. This query is not functional considering that it has parameters that must be replaced by specific values. When these parameters are replaced by the specific values, then a completely functional SPARQL query is achieved. In this case, a EWE rule.

A.3 Template generation strategy

The strategy to follow is simple, but results very functional. We will define different templates based on the number of parameters that will be connected. In this way, there will be templates for connecting 'n' event parameters with 'n' action parameters.

We must define what concepts are to be abstracted. As it has been showed in the section A.1, the rule musts (i)**select an event**, (ii)**generate a certain action**, (iii)**make a parameter mapping**. These three processes will be abstracted and reflected as parameters in the templates.

A.4 Parameter abstraction for the templates

A.4.1 Event selection

The event selection is made from the restriction:

```
?event a <http://gsi.dit.upm.es/ontologies/ewe/channels/ns/NewFollower> .
```

These restrictions attach the event URI, but this must be a parameter. The result would be:

```
?event a <?eventURI> .
```

A.4.2 Action selection

The action selection is similar to the process in A.4.1:

```
?action a <http://gsi.dit.upm.es/ontologies/ewe/channels/ns/SendMeAnSms> .
```

This selection is composed into:

```
?action a <?actionURI> .
```

A.4.3 Parameter mapping

It consists in taking the value of one of the event output parameters and assign it to one of the action input parameters.

```
CONSTRUCT {
    ...
    ?action <http://gsi.dit.upm.es/ontologies/ewe/channels/ns/Message> ?message .
    ...
}
WHERE {
    ...
    ?event <http://gsi.dit.upm.es/ontologies/ewe/channels/ns/Fullname> ?name .
    ...
    BIND( fn:concat(?name, " is now following you!") AS ?message )
}
```

In this example the URIs are specified in the rule. They will be abstracted into parameters. Besides, the name of the variables are relative to the rule context, so they will be changed into more generic names. This is not a functional matter, just a concept one.

```
CONSTRUCT {
    ...
    ?action <?actionParam1URI> ?actionParam1 .
    ...
}
WHERE {
    ...
    ?event <?eventParam1URI> ?eventParam1 .
    ...
    BIND( fn:concat(?eventParam1, "") AS ?actionParam1 )
}
```

A.4.4 Action namespace

In the example rule the action URI is not generic.

```
BIND ( URI("http://example.org/action") as ?action )
```

We need the actions to have a unique identifier, so controlling the action namespace is necessary. We achieve this introducing a new generic parameter.

```
BIND ( URI("?actionID") as ?action )
```

A.4.5 Conclusion

The template that maps one parameter is:

```
CONSTRUCT {
  ?action a <?actionURI> .
  ?action <?actionParam1URI> ?actionParam1 .
}
WHERE {
  ?event a <?eventURI> .
  ?event <?eventParam1URI> ?eventParam1 .
  BIND ( URI("?actionID") as ?action )
  BIND( fn:concat(?eventParam1, "") AS ?actionParam1 )
}
```

In this template it is needed seven parameters. We can observe that with the defined templates, in case of mapping **n** parameters we will need **2n+3** parameters. We have designed templates until 'n=3' parameters.

The rest of the templates are specified in the Wiki[1] of the Wool project.

[1] https://github.com/gsi-upm/Wool/wiki

Bibliography

[1] Mike Gartrell Aaron Beach. Proceedings of the eleventh workshop on mobile computing systems and applications. 2010.

[2] Fabio Barbon Paolo Traverso Marco Pistore adn Michele Trainotti. Run-time monitoring of in- stances and classes of web service compositions. 2006, journal=International Conference on Web Services.

[3] Miguel Coronado. Ewe ontology specification, 2013.

[4] Diego López de Ipina. An eca rule-matching service for simpler development of reactive applications. *IEEE DSOnline*, 2001.

[5] Carlos Angel Iglesias and Miguel Coronado. Ewe: Modeling rules for automating the evented web. 2013.

[6] Dennis Spohr Philipp Cimiano nad John M Crae. Using spin to formalise accounting regulations on the semantic web. *International Workshop on Finance and Economics on the Semantic Web*, 2012.

[7] Adrian Paschke and Alexander Kozlenkov. Rule-based event processing and reaction rules. *Internationa Symposium on Rule Interchange and Applications*, 2009.

[8] Adrian Paschke and Paul Vincent. A reference architecture for event processing. *ACM Press, New York*, 2009.

[9] Adrian Paschke Vincent Paul and Springer Florian. Standards for complex event pro- cessing and reaction rules. pages 128–139, 2011.

[10] Daryl C Plummer. Cloudstreams: The next cloud integration challenge. *http://blogs.gartner.com/daryl/plummer/2010/11/08/cloudstreams-the-next-cloud-integration-challenge/*, 2010.

[11] Vivek K Singh and Ramesh Jain. Structural analysis of the emerging event-web. *ACM Press, New York*, 2010.

[12] Stamatis Guinard Dominique Savio Domnic Baecker Oliver Spiess, Patrik Karnouskos. Soa-based integration of the internet of things in enterprise services. *IEEE Internation Conference on Web Services*, pages 968–975, 2009.

[13] Ryan Shaw Raphaël Troncy and Lynda Hardman. Lode: Linking open descriptions of events. *School if Information*, 2009.

[14] Philip J. Windley. The live web: Building event-based connections in the cloud. 2011.