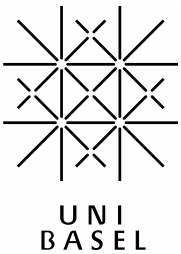
Towards Reactive Information Systems and their Services

Master Thesis

Author:
Dominic Bosch

Supervisors:
Prof. Dr. Helmar Burkhart
Dr. Martin Guggisberg

June 29, 2014



Abstract

The Web is a rapidly growing information universe, consisting of Information Systems that provide access to heterogeneous services. A large part of those Information Systems are dynamic. Changes in their Information Space trigger events, which can be detected. Moreover, such changes can also be imposed onto the Information Space over the appropriate services. If appropriate services exist to access such an Information System for read and write operations, we are able to orchestrate it. By adopting the Event-Condition-Action (ECA) paradigm to Information Systems, we are able to introduce an event-based conceptual model. This model allows the detection of events and the dispatching of actions according to predefined rules, thus imposing reactivity on top of or between Information Systems. Current approaches that use the ECA paradigm focus on action imposition on local storage, while we aim to impose actions on the hetereogeneous services of existing Information Systems. This model is not limited to the Web, but can include any accessible Information Systems.

In our work we introduce a prototype system, which uses the Web's programmability to impose reactivity on top of it. We also underline the importance of the, currently little, support of Web services for event callback addresses, the so called Webhooks. They are the only way for real-time event delivery to remote systems and free them from expensive polling for changes. Through our prototype it is possible to orchestrate Web services based on an Event-Driven Architecture (EDA), a method which pushes towards the vision of real-time reactive Information Systems. We list some example use cases for our conceptual model, as well as use cases that have been implemented in our prototype system.

Acknowledgment

I would like to express my deep gratitude to my master thesis advisor, Prof. Dr. Helmar Burkhart, for his patience and constant encouragement to keep going further. I also want to thank Dr. Martin Guggisberg for his ever helpful advices, hints to existing technologies and guidance through the whole process of this thesis. Besides my advisors, I would like to thank the members of the research group: Danilo Guerrera, Antonio Maffia, Alexander Gröflin and Robert Frank for their help, many interesting discussions and inputs. My thanks also go to Benedikt Willi for reading through this and providing valuable feedback. Finally I would like to thank my family; my parents Silvia and Peter Bosch, my sister Svenja, her husband Andreas Buser, their seven months old son Nico and my girlfriend Kathrina for their inspiring ideas, the joyfulness, irresistible smiles and financial support they gave me on my journey.

Contents

List of Figures

List of Tables

Listings

Glossary

Information Space *"[...] is a set of concepts and relations among them held by an Information System. Information Space is produced by a set of known procedures, and is changed through intentional manipulation of its content"*[27]. i, iv, 2, 3, 12–16, 21, 23

Information System is a network of software and hardware components that support collection, filtering, storing, processing and distribution of data. i, iv, 1–5, 7–9, 12–17, 21, 23, 24, 26, 28, 31, 35, 36

Mashup A Web Application that weaves two or more different Web APIs together to provide a new perspective on data. 4, 6

Semantic Web Tim Berners-Lee's vision of the machine-readable Web through standard data formats and semantic metadata descriptions on top of the resources via RDF which turn the Web into a structured data collection. 1, 5, 8

Web A common term referring to the current state of the World Wide Web, which underlines the use of recent dynamic technologies to enhance Webpages into Web Applications, also called Web 2.0. i, iv, 1, 2, 16–19, 28

Web API An Application Programming Interface to either a Web service or the browser, used for application to application communication. 3, 4, 6–8, 23, 24, 29, 35, 36

Web Application An application which runs in the browser. Often a user interface to an application sitting on a server. Single Page Applications (SPA), a subset of the Web Applications, access Web Resources over asynchronous calls to the server while the user is interacting with the application. iii, 1, 17, 27, 31, 35

Web of Things An evolution of the Internet of Things, which describes the integration of smart things (e.g. sensors, embedded devices or digitally enhanced objects) into the Internet. The Web of Things is the adoption of the REST architectural style to the smart things in order to enable uniform access to these loosely coupled entities. iii, iv, 1, 13, 15, 19, 36

Web Resource Anything in the Web which can be identified, addressed and handled. Identification and addressing is often done over URIs. Web Resources and their semantic properties are described using RDF in the Semantic Web. 1–3, 5–8, 10–12, 14, 17, 23–25, 35, 36

Web Service A collection of SOAP related Web service (note the lower-case word service) standards which are widely adopted and developed in the industry. Also called "WS-*" Web Services or the Big Web Services. 3, 4, 23

Web service An interface for communication between applications over a network. They can provide access to and control over Web Resources. Web services are also called services on the Web or just services. i, 3–7, 13, 23, 24, 31, 35, 36

Webhook A server-side Web API which accepts a URI that is used as a callback to push events to an external Web Resource. i, 7, 14, 21, 22, 24–26, 28, 31, 35, 36

World Wide Web Tim Berners-Lee's vision of interlinked hypertext documents which are accessed over the Internet via browser and allow the navigation through a global information universe. The term World Wide Web is often referred to as the first stage of the Web, the Web 1.0. iii, 1, 3, 5, 14, 16, 24

Acronyms

CED Complex Event Detection. 10, 14

CEP Complex Event Processing. 9, 10, 14, 36

CMS Content Management System. iv, 17

CORBA Common Object Request Broker Architecture. 5, 9

DOM Document Object Model. 8, 11

ECA Event-Condition-Action. i, iii, 3, 7–10, 14, 15, 21, 26, 27, 36

EDA Event-Driven Architecture. i, 7, 9, 10, 15, 21

ESB Enterprise Service Bus. 9, 10

HTML Hypertext Markup Language. 35

HTTP Hypertext Transfer Protocol. 5, 36

IaaS Infrastructure as a service. 3

IDL Interface Definition Language. 5

IIOP Internet Inter-ORB Protocol. 5

JSON JavaScript Object Notation. vi, 7–9, 11, 21–23, 26–28

KR Knowledge Representation. 9

KRE Kinetic Rules Engine. 9, 11

KRL Kinetic Rule Language. 9–11

ORB Object Request Broker. 5

PaaS Platform as a service. 3

RDF Resource Description Framework. 8, 11, 36

RDFTL RDF Triggering Language. 8, 11

REST Representational State Transfer. 3, 5, 7, 23, 35, 36

REWERSE Reasoning on the Web with Rules and Semantics. 8

RPC Remote Procedure Call. 4, 5

RuleML Rule Markup Language. 9, 11

SaaS Software as a service. 3

SOA Service-Oriented Architecture. 3, 4, 21

SOAP Simple Object Access Protocol. 3–5, 23

URI Uniform Resource Identifiers. 5, 7, 24, 25

WSDL Web Service Description Language. 4, 5

XML Extensible Markup Language. 4, 7, 9, 21

XML-RPC XML - Remote Procedure Call. 4

Chapter 1

Introduction

The term World Wide Web has been coined by Tim Berners-Lee[6] and is also referred to as the shorter term Web. Initially the Web was only associated with interlinked hypertext documents, as they built the fundament of the World Wide Web. Speaking of the Web today, is widely understood as the current state of the evolving World Wide Web, which also holds technologies to support collaboration and dynamic changing of the webpage, which are called Web Applications.

Information Systems are omnipresent in today's world. Being connected to the Web is more a common attribute than a special quality among them. As a consequence, the Web is an ever growing institution in all aspects it covers. The number of data and functionality providing Information Systems is growing in the whole spectrum from bigger computing centers down to smaller devices. Computing centers are growing in size and quantity and allow massive amounts of data to be stored and accessed. Moreover they also enable the construction and offering of more complex functionality. At the same time, an increasing number of ever smaller devices also provides more Information Systems attached to the Web. Many of them are offering access to the Web itself, granting even more devices access and thus leverage the effect of the growing Web, e.g. mobile phones can act as a hotspot to grant Web access to other devices over WiFi. A recent observed trend are all the smart things in the World, which have access to the Web and start to form the Web of Things. Today, these smart things can be everything, from a temperature sensor to all the electronic devices within a house. They do not only provide sensor data but they can also be controlled over the Web. All these different types of services available on the Web make it a heterogeneous collection of Information Systems and their services. Great efforts are made to turn them into uniformly accessible Web Resources, e.g. the Semantic Web is a widely supported initiative towards a machine-readable, structured and semantically descripted Web.

Confronted with this rapid growth of the Web, an increasing number of human beings is exposed to it in their daily life, and they get literally flooded with informations and means to retrieve or process them. Even though users have access to so many Web Resources, they often lack the knowledge, necessary time or right approach to weave them together. Great value would be added for them if they could automatically get appropriate informations, in the right moment and in a condensed matter that supports them best. They should be able to automate tedious tasks, e.g. detecting relevant changes in their preferred Web Resources and react on behalf of such changes. This requires the identification and filtering of user-relevant changes, appropriate timing, assembly and finally the placement of the outcome in the user's preferred Web Resources.

With the many existing Information Systems and their services on the Web, users do not want to be bound to specific ones for certain tasks, as it is often the case nowadays. They want

to use the functionality or data of their preferred ones, which helps them best to fulfill their needs. Hence, users should have the ability to create their own specific but still flexible Web Resource orchestrations. Since the need of users to orchestrate different services has gotten a lot of attention, some of the Information Systems on the Web offer ways to spread their data to others, but in a limited way. For example it is common for social network applications to push user-specific notifications to other social networks, e.g. signing in at a place in Foursquare can also be posted directly to the Facebook timeline. Because of existing limitations, such as customizability or action imposing on the Information System of their choice, users still end up mixing data and functionality from different Web Resources by hand, which often means to execute similar tasks repeatedly themselves. Moreover the manual reaction on changes is deferred because the changes are not detected in real-time by the user or because the user is not able to react in a timely fashion.

Since data and functionality already exist in the Web, the users are theoretically enabled to automate their work to some extent by orchestrating those Web Resources. Even though the access to resources gets simpler, the average user is still not capable to fully exploit the Web's full potential. Another challenge is, that often a lot of effort has to be made, in understanding how the specific service works, before it can be fully exploited. There is a lot of research that goes towards an easy to orchestrate Web, but those approaches are either complicated to wield, mere data copy tasks or static Web Resource compositions. Our goal is to enable user-defined resource orchestrations, and still exploiting their full potential by not limiting the set of their functionality and thus going towards a reactive Web.

A big part of the data, that becomes available to the users, is short-lived data that corresponds to state changes, which are changes in the Information Spaces and can be modeled as events for detection or actions for imposition. In this thesis we introduce an event-driven conceptual model that uses the programmability of Information Systems and their services in order to impose reactivity between them. By regarding the whole Web as an Information Space, in which we listen for triggered events and on which we impose actions as a result of user-defined rules, we are able to model a real-time reactive Web, as shown in Figure 1.1. Such a personalized reactivity allows the orchestration of the Web and a tool to govern its data flood by automating tedious tasks. Current Web Resource orchestrations concentrate on data flow rather than on event flow, which are mere copy/paste tasks of data than smart reactivity. This makes us believe that our event-based conceptual model can overcome certain shortcomings of the existing approaches and provides a step towards the real-time reactive Web.

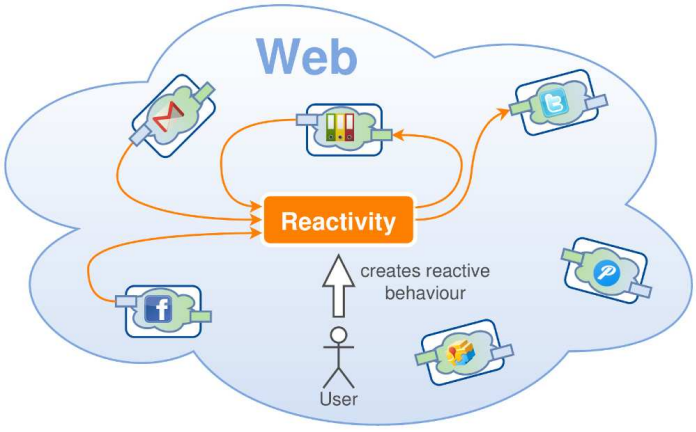


Figure 1.1: Users exploit the Web's Programmability through a Reactive Entity in the Web

Chapter 2

Related Work

In this chapter we will give a brief overview over Information Systems in the Web and their associated Web services as the main drivers for interoperability between Information Systems. We then introduce the Event-Condition-Action (ECA) paradigm, an event-driven approach to impose reactivity. And finally, we analyze existing rule languages and engines that exploit the ECA paradigm.

2.1 Data and Functionality Providers on the Web

Information Systems on the Web and their services are also called Web Information Systems and their Web services. Web Information Systems often provide access to internal resources through their services, such as documents or objects, which can be identified and addressed and thus fall into the category of Web Resources. The internal set of resources and their relations form the Web Resource's Information Space[27]. The term service in the context of the Web is somewhat ambiguous and there have been a lot of completely different approaches to offer services on the Web, some of the latest used in cloud computing are Platform as a service (PaaS), Software as a service (SaaS) and Infrastructure as a service (IaaS). The term Web Service (capitalized word Service) commonly stands for Web service based on Simple Object Access Protocol (SOAP) communication[5], which has been adopted and developed extensively by both research and the industry and is often referred to "WS-*" Web Services. With the advent of the Representational State Transfer (REST) architectural paradigm, the understanding of the term service in the Web has undergone a slight generalization so that the term Web service (lower-case word service) is not anymore bound to SOAP, but describes services as interfaces for communication between applications over a network[37]. We will point out main research areas on service-orientation within the Web and show how they make the Web programmable.

Execution of programs on remote computers has always been a strong research area, ever since computer started to exist. After the coinage of the term World Wide Web[6], the Web has become a synonym for Berners-Lee's vision of a global information universe. The adoption of remote program execution through the Web followed immediately; computers sitting in the Web waiting for a request in order to execute some application logic and return an answer. Similar to this concept is the encapsulation of functionality into services[32] in order to offer them to other applications, which is called Service-Oriented Architecture (SOA)[33]. Applying SOA to an existing Information System means splitting it into smaller loosely-coupled pieces (services), which then have to communicate with each other over proper interfaces. Well described server-side service interfaces, meant for application to application communication are called Web APIs.

But the term Web API not only comprises server-side interfaces but also client-sided ones (e.g. the browser), since they are also interfaces to programmability of the Web. Proper interfaceas do not only provide robustness, it also allows the reuse of functionality. Moreover these services can be offered to other applications and also to the Web, thus allowing others to access certain functionality or large parts of the Information Systems behind the services. All nodes in the Web are stand-alone entities, which offer services of some sort, be it a webpage, pure data, real-time measurements or functionality, such as computing an answer from input parameters. This makes the Web itself a Service-Oriented Architecture and all the services within it are naturally Web services. It is because of its advantages, that SOA has received a great deal of attention and has been widely adopted throughout the Web. This lead to an increasing number of Web accessible services and their compositions, the so called Mashups. An empirical study[24] on a directory, which the researchers of the paper call the *"[...] most active Web APIs and mashups collection"*, and statistical data taken from this directory (depicted in Figure 2.1) seem to underline a growing popularity, at least in terms of published services within this particular directory.

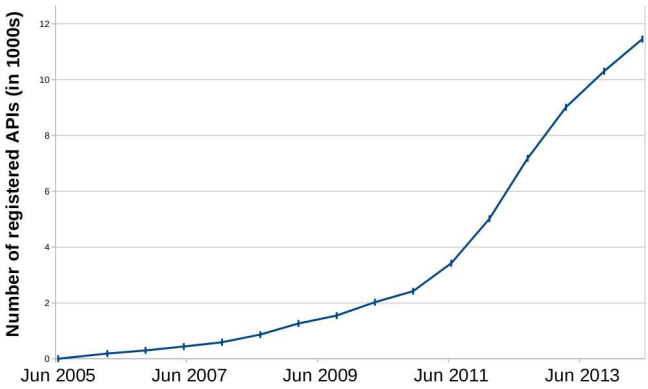


Figure 2.1: Number of registered APIs in the ProgrammableWeb directory by date

2.1.1 Accessing services on the Web

An early adoption of the service concept to computers were the Remote Procedure Calls (RPC)[8]. Through RPC a piece of code can be executed on a different machine, other than the one which is calling the procedure. It is basically an inter-process communication and does not necessarily require the Web nor distributed computers. RPC also found its use in grid computing[40] and through this, opened doors into the field of distributed computation. The RPC paradigm is not bound to certain technologies and thus, has been implemented in a lot of different programming languages. These implementations were tightly bound to the respective language that was used, which resulted in incompatibility among them. It became necessary to enhance RPC's in order to get cross platform compatibility. The abstraction of RPC through the Extensible Markup Language (XML)[11], called XML-RPC, made it easier to achieve compatibility between services that base on different technologies.

Since XML-RPC was held relatively simple but received a lot of attention, it was further enhanced. Together with additional proposed functionality, XML-RPC was the base for Simple Object Access Protocol (SOAP)[10]. SOAP is accompanied by the Web Service Description Language (WSDL)[15] which is used to describe the interfaces to SOAP services, the Web Services. With SOAP and WSDL a client of the service can issue a request for the WSDL information of the service and retrieves all interface specifications he requires in order to issue a call to the actual service. The service specifications are then usually incorporated into the existing Information System as if it is a local function call. SOAP has found its applicability

in business applications[5] and was enhanced with a lot of industrial standards, also called the "WS-*" specifications, e.g. WS-Addressing, WS-Policy, WS-Security and many more.

Another initiative that aimed for eased communication between different platforms is the one for the Common Object Request Broker Architecture (CORBA)[22]. As the name already suggests, it is an object-oriented approach and it allows the access to whole objects over a network. CORBA relies on its communication layer, the Object Request Broker (ORB), which forms the basis of its architecture. The platform-specific ORBs provide the communication abstraction, which free the application from platform dependencies. Similar to SOAP's WSDL, CORBA has its Interface Definition Language (IDL) to provide information about the accessible objects. An object is instantiated by an application and the interface to this instance is offered through the ORB. Another application attached to the ORB can then access all public variables, data structures and functions of this object. This means not only remote access to variables and data structures, but also remote function invocation as seen in RPCs. For programming languages that are not object-oriented, this behaviour has to be simulated, which can be technically difficult and become a tedious task. CORBA enables communication between applications written in different programming languages, no matter whether they run on the same physical device or another one in the network or even the Web. With the Internet Inter-ORB Protocol (IIOP) it is also possible to connect ORB's over the Web. Through this, the offered objects can become services in the Web, but they are shielded by the ORB and only accessible over it.

2.1.2 Web Resources become Services on the Web

All the afore mentioned approaches require a specific protocol and as a consequence are incompatible with each other. For this reason and its simplicity, an architectural style has gained popularity which frees Information Systems and their services from this constraint: Representational State Transfer (REST)[19]. REST concentrates on the roles of components and on constraints upon interactions between them. An important architectural constraint is that all communication is stateless, which means for a client-server communication, no state is stored on the server. Therefore all informations required for a single interaction need to be provided within one request. This allows for the definition of simple and well-defined interfaces, since responses are not bound to a certain session state. Services within the Web that adhere to the REST architecture are called RESTful Web services. RESTful Web services provide access to data and functionality of grouped Web Resources, which can be identified via Uniform Resource Identifiers (URI)[26]. In the Semantic Web[7], a Web Resource is anything in the Web that can be identified, addressed and handled. Historically this started with documents and went over objects to abstract concepts, such as operators of equations. Simple access to Web services without communication overhead and negotiation before using it, increased REST's popularity and made it spread into more application fields. RESTful Web services are often implemented using the HTTP request methods GET, POST, PUT and DELETE which allows for all the necessary operations to create, update, read and delete Web Resources. By using HTTP, the protocol on which the World Wide Web bases, a wide range of Information Systems should be able to communicate with the service. There is for example the upcoming concept of the Web of Things[23], which aims to incorporate smart things (e.g. tagged things, sensor measurements, device controllers, etc.) into the Web through REST interfaces. Even though the nature of such things is usually compatible with the REST architectural constraints, incompatible standards and protocols were used by different manufacturers. Therefore REST brings advantages into the context of smart things connected to the Web.

2.1.3 Composing Services in the Web

Through the upcoming of scripting languages, webpages emerged into dynamic sites on the Web, which actively control the browser and thus the webpage itself while the user is interacting with it. With all their server-sided infrastructure in the background they became literally applications, with more or less functionality and persistence on a server. These Web Applications (Web Apps) became even more responsive with the upcoming of asynchronous calls from the browser to the server, which allows to load data into the current webpage while the user is interacting with it. Those asynchronous calls are requests to services, which act as the Web API to the Web App, which sits on the server. For server-side Web APIs this means that these services can be accessed from other entities in the Web than just browsers, which eases application to application communication. Often the model behind a Web App can be controlled without the Web App itself, depending on how fat the server-side and how thin the client-side is. Imagine not going to the Google webpage anymore to issue a search and manually crawling through the results, but you have your own application doing it for you and processing the results instantly. There is a trend of Web App providers to publish their Web API in order to allow easy access to it. This has lead to an increasing number of Web App Mashups in the past few years.

Mashups combine Web APIs of more than one service in the Web in a new site. Simple services from different sources can be combined into more powerful ones, which can again be composed. These service compositions assemble data and services in a novel way which provides a new perspective on the data. Ever since services were accessible in a more or less convenient way, Mashups have been developed as well. One of the first Web service Mashups[35], was invented in the same year after Google Maps came up in 2005. It was a webpage that displayed CraigsList's rental houses on a Google Map. At that time no Web API was available, to provide easy access to those two services. But there was value to be observed from anybody being able to create a Mashup through publicly available services, because this leads to new ideas and an increase of popularity in all incorporated Web services. Such Mashups are often a read-only and fixed wirings of different Web APIs that provide a new perspective on specific data. Some recent Mashup examples are:

- Wifi and Plugs: MapBox, Google Docs and Import.io API's used to display where Wi-Fi and plugs are available in London.
- MapLight: GovTrack.us and OpenSecrets API's used to combine political results with financial contributions, in order to show how capital contributions to certain campaigns influence voting.
- Shared Count: Facebook, LinkedIn, Pinterest and Twitter API's used to display informations about how well spread a URL is on social media sites.

But also a number of studies[14][25][38][41] made efforts towards personalized Mashups, where users are capable of choosing what and how to link in order to enhance Web Resources according to their needs. These flexible Mashup applications often provide methods to access user-specific functionality within external Web Apps, which makes them even more user-centered and customizable.

2.1.4 Subscribing to Web Resources

There is another type of service in the Web which is about the opposite of the afore mentioned approaches in terms of the data flow. It is the concept of push notifications on state changes, which is a recent research area. There are some manifestations of this model for server to browser

communication, such as Comet[16] or Server-Sent Events[1]. The concept is called Webhooks and introduces instant delivery of data whenever it gets available, compared to the need of actively requesting a service to deliver it. Webhooks are URIs, which point to a Web Resource, which accepts the data delivered to it. Within the publish/subscribe paradigm[18], such asynchronous delivery of data is referred to as events, since it depicts the appearance of new data. Webhooks are callbacks that can be placed by a Web service provider at a remote Web API, informing a distinct event delivering Web Resource about the interest in the promised events. Both parties are a sort of Web service, since the Webhook providers accept the data delivered to their URI and the Webhook receiver accepts URIs and offers to send the data. PubSubHubbub[2] is an open server-to-server publish/subscribe protocol that uses Webhooks for servers to announce their interest in updates from other servers. Only through such push notifications a reactive system can be real-time reactive through instant event detection.

2.1.5 Towards Simple Access and Communication

With JavaScript's success as browser scripting language and recently also as server-side programming language, JavaScript Object Notation (JSON), as an alternative to XML, has become popular for data representation and communication throughout the Web. Another factor for its popularity is the human-readable format and often simple parsing into data structures of existing programming languages. There is a notable trend towards RESTful services in the Web that offer JSON communication. They benefit from simple but fully capable interfaces and easy to debug human-readable communication, which eases integration into other applications, along with low communication volume. Together with client- and server-side Web APIs the Web becomes ever more programmable.

2.2 Reactivity through Event-Condition-Action Rules

In this chapter we have so far shown research in different areas, which lead towards a programmable Web. As a result of this research, it is getting easier to compose and orchestrate Information Systems, but reactivity needs to be programmed specifically by experts and general approaches are only available in specific domains. Several studies[3][12][13][28][30] have been made on reactivity. They point out Event-Condition-Action (ECA) rules, visualized in Figure 2.2, as a natural way to impose reactivity on Information Systems.

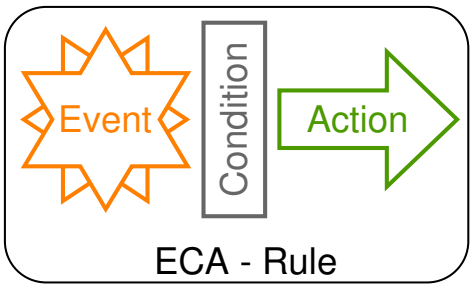


Figure 2.2: Subsuming Event, Conditions and Actions in a Rule leads to Reactivity

As the name already suggests it bases on an Event-Driven Architecture (EDA) consist of three parts:

[1] http://dev.w3.org/html5/eventsource/
[2] http://code.google.com/p/pubsubhubbub/

- Event: An event identifier for the to be detected event
- Condition: Expressions to be evaluated to determine whether the action section should be triggered
- Action: A set of instructions that complete the reactive behaviour

Several different rule languages have been developed for different domains. We will give a brief overview over research related to our vision, reactivity in the Web. During our research, apart from identifying the key properties of different rule languages, we analyzed them with respect to a certain use case, in order to determine their applicability for our research goal. The use case's ECA rule is:

- Event: Receipt of an email
- Condition: Assert a certain sender email address
- Action: Store the message remotely via a Web API

We also tried to get access to existing rule engines for each rule language, since we aim to build our model as well as our own reference implementation on top of existing work.

2.2.1 Rule Languages & Rule Engines

The Resource Description Framework (RDF) is a collection of specifications to model information in the Semantic Web. Papamarkos et al. (2004) published an ECA language for RDF; the RDF Triggering Language (RDFTL). It was designed to react on insertion and deletion events within RDF repositories and as an action propagate the changes through related resources and execute actions on the local repository. RDFTL bases on RDF resources which need to run engines in order to react on the rules. These engines retrieve events, detect changes and communicate them as events to other engines, while actions are executed on local repositories. Through distributed engines, RDF resources are made reactive. We envision an engine that orchestrates the Web, rather than relying on other Information Systems to incorporate our model. Nevertheless, their research provides important insights on reactivity through ECA rules.

The rule language XChange[31] emerged from the Reasoning on the Web with Rules and Semantics (REWERSE) project[36], which took place from 2004 to 2008. It was designed to track changes in dynamic Web Resources and add reactive behaviour in a way that such changes influence other dynamic resources. XChange incorporates the vision of distributed, event exchanging rule engines. Those rule engines execute actions on local data or issue new events. The local-only actions oppose our vision to orchestrate heterogeneous Web Resources through reactive behaviour, as does the RDFTL. The use case applicability study was promising but access to a reference implementation of an engine, in order to enhance it with our vision, could not be gained. But the thorough research done with the language XChange holds valuable concepts, especially in terms of temporal event composition.

JSON Rules[21] was introduced 2008 as a language to react on specific events in the Document Object Model (DOM) tree of a webpage and, as reactive behaviour, control the browser and also the DOM to change the webpage. The incorporation of script function calls into the action part of the language allows the abstraction of eventually complex action behaviour. This feature influenced our concept as it allows for different levels of complexity to be offered and regards the different levels of expertise possible users have. JSON Rules is bound to DOM tree events and actions, while we aim to react on all events happening in Web Resources and also execute actions on them.

The Rule Markup Language (RuleML)[9] is a language written in XML and aims to standardize many different types of rules. Reaction RuleML[30] is an enhancement of the existing standard by reactive rules. Reaction RuleML subsumes:

- Complex Event Processing (CEP)
- Knowledge Representation (KR)
- Event-Condition-Action (ECA) rules
- Production (CA) rules
- Trigger (EA) rules

Reaction RuleML represents thorough research for a language to describe virtually any type of reactivity. Together with the expression in XML it does not score with readability, but provides a way to define a multitude of rule types and ensures their interchangeability between different Information Systems. Since our vision does not require interchangeable rules, we chose an internal JSON representation, inspired by JSON Rules, for our rules to have more important properties, such as human-readability, simple parsing and efficient storage. A notable system that relies on RuleML is Rule Responder[29]. Rule Responder connects different types of heterogeneous rule engines together over the Mule open-source Enterprise Service Bus (ESB) which acts as a communication middleware to exchange rules expressed in RuleML. The introduction of a communication layer between Information Systems is the same concept as in CORBA.

A recent research outcome (Windley, 2011) is the Kinetic Rule Language (KRL)[42] together with the Kinetic Rules Engine (KRE). It was invented to impose reactivity to the Web and incorporates many different event origins and action resources. The language is based on a declarative syntax, enriched with imperative elements. An interesting feature is the incorporation of the browser into the architecture, which bridges the gap between the user's browser and the centralized KRE. Either a user can install a browser plugin which will communicate with the KRE, or a webpage provider can include a library in order to get events from accesses to the webpage. Through this, events can be raised from the browser and actions can also be executed in it. The KRL fits very well into our concept and only a few reasons kept us from realizing our reference implementation on top of the KRE, such as:

- complexity required to maintain states with a declarative syntax
- system footprint of the KRE
- Perl, a procedural programming language, as basis of the KRE

The concept of the KRL is promising in terms of orchestrating the Web through reactivity. But we made the decision towards a light weighted reference system, using an event-driven programming language that supports Event-Driven Architecture natively. Another important key property of our envisioned conceptual model, which diverges from the KRE architecture, is the abstraction of state maintenance into action dispatching modules, rather than incorporating them into the language.

2.2.2 Overview over existing Rule Languages and their Engines

Table 2.1 gives an overview of the key properties of existing rule languages and their key properties for our research:

- Event Origin: Resource type from where the events originate.

- Distributed: Whether the language is laid out to run on a centralized or distributed architecture. All examined rule languages that support distributed architectures run on a centralized architecture as well.
- Action Resource: Resource type on which actions are executed.
- Accessible Engine: Lists accessible engine reference implementations for the language.
- Applicability to our concept: Names the main difference to our envisioned concept.

Rule languages that support a distributed architecture require engines to be deployed on sites in order for them to be reactive, since actions are only imposed locally. This is not service-oriented in terms of external services and does not attempt to orchestrate the Web's heterogeneous services in the action part of rules. It seems common for ECA rules to only invoke actions on local systems, even though the KRL goes into the direction of accessing remote systems too.

Other examined Rule Engines are the Object-Oriented Java Deductive Reasoning Engine for the Web (OO jDrew), Prova, and Drools Fusion. They are all implemented in Java and have their own rules syntax which is more or less closely related to Java, with inline Java code, except for Prova which uses a Prolog like syntax. These Rule Engines either do not fully support our vision or have an unnecessary overhead, such as Drools Fusion, which bases on the Java graphical user interface eclipse, or Rule Responder which relies on the communication middleware JBoss ESB. We envision a scalable system, which is event-driven from the application layer, and allows the orchestration of heterogeneous Web Resources. These resources are already available and accessible and we do not need to alter them in order to impose reactivity to the Web. Such a system does not require a messaging middleware because the Web itself acts as the communication channel to receive events and to dispatch actions.

2.2.3 Complex Event Processing

An important research area in the field of Event-Driven Architectures is Complex Event Processing (CEP)[4] and deals with event composition, also called Complex Event Detection (CED)[2][34]. It is the research for methods to detect predefined event relations and also temporal patterns, over different streams of data or events. This topic has received a lot of attention for active databases[1][20][43] and was picked up again in the context of Event-Driven Architectures. With CED atomic and composite events are successively aggregated into higher-order events, regarding temporal constraints. The event-driven architecture of reference systems allows the processing of large amounts of data and assemble compositions in real-time. In our conceptual model we envision a CED engine that detects complex event patterns and assembles them into higher-order events. These complex events can then be detected by the ECA rules engine to dispatch appropriate actions.

Language	Event Origin	Distributed	Action Resource	Accessible Engine	Applicability to our concept
RDFTL	RDF Repository Changes	Yes	(Local) RDF Repository	-	Only Web sites with engines are reactive
XChange	Web Resources	Yes	Local Resources	-	Actions in remote Web Resources missing
JSON Rules	DOM Events	No	Browser / DOM	-	Only Browser / DOM Events
RuleML	Web Resources	Yes	Local Resource	(OO) jDrew, Prova, Rule Responder	Complex Syntax
KRL	Web Resources	No	Local & Remote Web Resources	KRE	User-specific Web App functionality missing

Table 2.1: Key Properties of existing Rule Languages

Chapter 3

Conceptual Model for Reactive Information Systems and their Services

The challenges and opportunities arising with the growth of the Web in terms of volume and complexity inspired our research towards the reactive Web. Therefore our starting point was the studies of related work in the context of reactivity on the Web, event composition and programmability of the Web. In the last chapter, we pointed out how they received a lot of attention and provide powerful tools to orchestrate the rapidly growing Web. By combining existing research in these fields we developed a conceptual model, which allows to impose smart reactivity to any Information Space, not only the Web. Even though our initial set of Information Spaces was thought to consist of Web Resources, our model is applicable to any Information System whose Information Space can be accessed and altered over interfaces, i.e. services. Thus we introduce our conceptual model for reactive Information Systems and their services in this chapter.

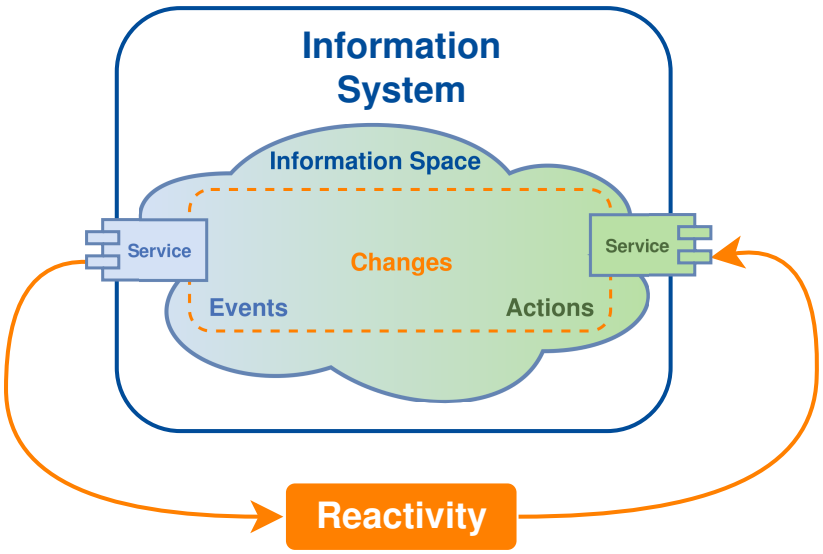


Figure 3.1: Reactivity imposed on Information Systems and their Information Spaces over Services

Data changes within an Information System can be detected and imposed from the outside,

if apropriate interfaces to the services exist. We model the detection of data changes as events, and the imposition of such changes as actions, as shown in Figure 3.1. Through this we are able to introduce an event-based model that is capable to detect events and react on behalf of them by executing actions on any Information Space. A more precise distinction of the required modules for such a reactivity imposing entity is displayed in Figure 3.2. Each of these modules is introduced in this chapter.

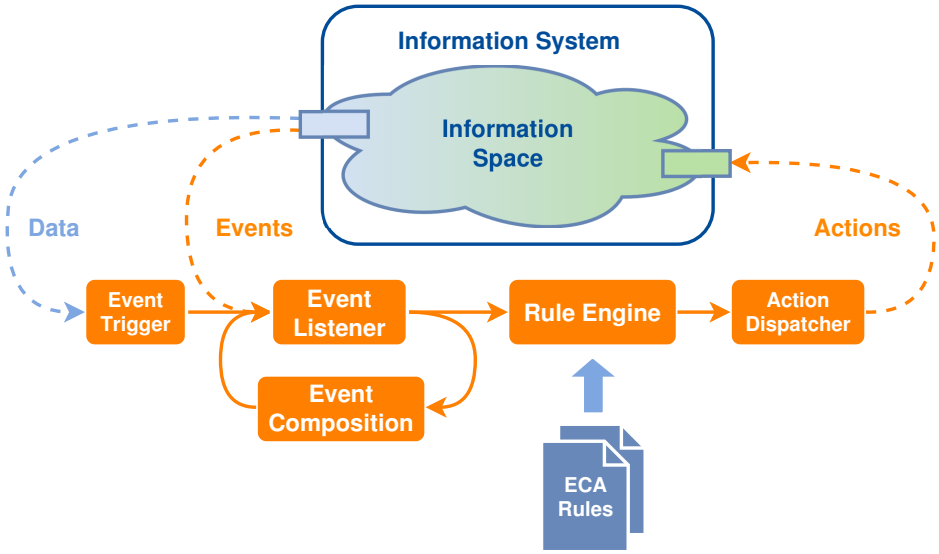


Figure 3.2: Conceptual Model for Reactive Information Systems and their Services

3.1 From Physical Events to Virtual Events

The Web of Things, where smart devices gain access to the Web, has already been mentioned shortly in the last chapter. It is based on the Internet of Things which dealt with the incorporation of sensor networks into the Internet on the network level. Such sensors bring the physical world directly into the virtual world. This transformation pictures an important difference between physical and virtual events. In physics, and in particular relativity, an event indicates a physical situation or occurrence, located at a specific point in space and time. While physical events correspond to a physical situation which is located at a specific point in space and time, virtual events primarily consist of implicit parameters, i.e. a name and occurrence time. These virtual events can be anywhere within Information Systems at any point in time, thus their actual location differs most certainly from its occurrence. Virtual events also have explicit parameters which correspond to available information about the event, such as the origin. As soon as events are transformed into the virtual world, the afore mentioned location information is transformed into explicit event parameters. But every virtual event has a name, an occurrence time and most likely some explicit parameters attached to it. If the virtual event has a physical nature, it contains a physical location, if it has a virtual nature, it is likely associated with a virtual origin. Since in our model events are changes in data of an Information Space, they can be virtually anything, e.g. physical measurements, changes on a static webpage, changes of the object behind a Web service or a login attempt.

3.2 Capturing Events from Information Systems

The optimal case for an event-driven system which requires events from a remote Information System is, that events are triggered within the remote Information Systems and then immediately communicated to interested parties, such as our envisioned reactivity imposing system. Our research has shown that such Information Systems are often passive and rarely provide ways for external systems to announce interest in changes of their data (e.g. in the context of Web Resources). Many Web Resources provide access to their data over services, but do not actively communicate changes to interested parties. This is where the upcoming concept of Webhooks comes into play. Because of effectivity, it is essential for Information Systems to push event notifications to external systems, instead of letting them poll for events. Through them such pushed events it is possible to have real-time reactivity without high costs of continuously polling for changes over all Information Spaces. We also need to take passive Information Systems into account which do not push events to external systems, therefore we need to incorporate polling for changes into our model. Wherever an Information System is not capable to provide events to external Information Systems, we can still read all the accessible data and detect changes in it, define them as events and feed them into our model. In our model the polling for changes is incorporated in the Event Trigger modules. Those are flexible modules have the proper tools to access any Information System service and therefore its Information Space and are capable of identifying changes in the data. For example the World Wide Web, is an information universe of interlinked documents, that a user can browse through. Through our model, we can pull changes in the data on the World Wide Web, i.e. document changes, and turn them into events. These events which are derived from changes in the data of Information System are then fed into the Event Listener. The Event Listener also pulls events directly from the Information Systems that offers service functions which represent events but still need to be requested actively, e.g. new mail in inbox.

3.3 Event Pattern Detection

Traditional ECA systems only react on single events, but this might often not be enough to detect meaningful situations. Primitive events occur at a point in time (e.g. a mouse button press event). When they are composed (e.g. the latter event with a mouse release event), they turn into a composite event which is more complex and also has a duration. Such a temporal event composition yields the chance to detect meaningful situations out of primitive events and react on them. This is why there is a trend towards the detection of complex event patterns, as we have pointed out in the last chapter. CEP could be incorporated into the rules of the Rule Engine, which then reacts on event patterns. Though such an approach opposes our vision of a successively growing complexity of composite events, which are defined on top of each other and are fed back into the Event Listener. Thus in our model an Event Composition module composes events into more complex events according to CED definitions. It is a very active research field, which has seen interesting studies[2][34] and outcomes[1] that could be incorporated into our model. Such an event composing service systems works loosely coupled an could be realized by any suitable system, as described in [39].

[1] such as http://drools.jboss.org/drools-fusion.html

3.4 Imposing Reactivity to Information Spaces

In the last chapter we gave an introduction into reactivity and the ECA paradigm as an approach to achieve it. So far in the introduction to our model we have introduced the foundation for an Event-Driven Architecture. We also need a module that translates events into actions on Information Systems. Almost all existing ECA system actions write on the local Information Space which opposes our vision of the orchestration of different Information Systems in order to impose reactivity on top of or between them. For that reason we introduce the Action Dispatcher modules which are located right behind the Rule Engine in terms of the data flow and complete the reactivity flow between heterogeneous Information Systems. Action Dispatcher modules are an important part of our model because they allow flexible coupling with Information System services, much like the Event Trigger modules do. Event Trigger and Action Dispatcher modules are communication abstractions to services of Information Systems, that allow us to deal with their heterogeneity in terms of communication. The Information Space of an Information System is not limited to internal data, but can also reflect a coupling with other devices, and the sensing and controlling of it. Thinking of the Web of Things this could include an Action Dispatcher that has access to an Information System which controls devices. Through this it is, for example, capable of turning down the heating in a house, as shown in Figure 3.3.

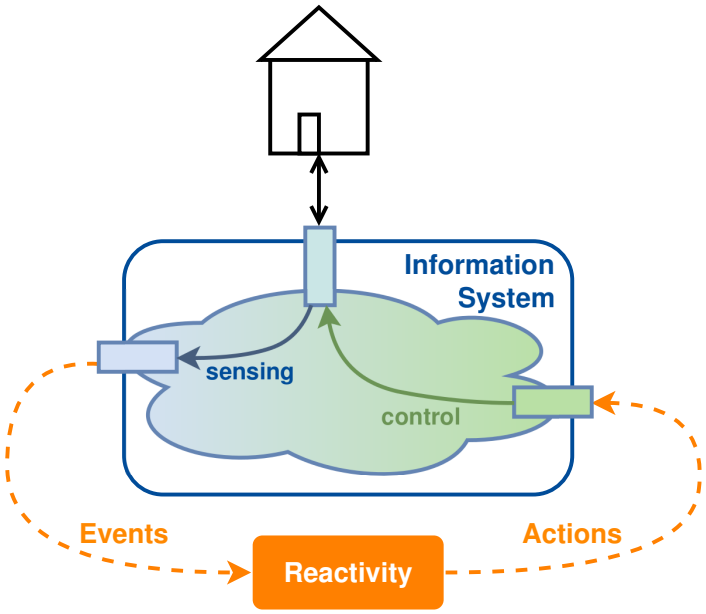


Figure 3.3: Information Systems providing access to the Web of Things over their Services

So far we defined all the modules to access Information Systems over their services, therefore we are able to define a Rule Engine that orchestrates them in a reactive way. We have shown in the last chapter that ECA rules consist of three parts; an event to be recognized, conditions to be evaluated on the event and actions to be executed if an event triggers the rule through valid conditions. In our model events are coming from the Event Listener to the Rules Engine, which checks all rules against the incoming event. If any conditiong section of an active ECA rule evaluates to true, the action section (consisting of different Action Dispatchers) of that specific rule is executed. Through this we described a complete reactive cycle that is able to impose reactivity on top of any existing Information System, if appropriate services exist.

Chapter 4

Use Cases for Reactive Information Systems

We have so far introduced a conceptual model for reactive information systems and their services. Through our model we are able to react on events happening in Information Systems and dispatch changes to it or any other accessible Information Systems. In this chapter we will give examples of what use cases can be realized through our model.

4.1 Reacting on changes in the World Wide Web

Many documents of the World Wide Web are dynamic in the way that they change over time. Some might change in an interval of a few minutes (e.g. news), while others change much slower, such as knowledge sites. To detect such changes an Information System in the Web needs to keep track of the document history and trigger events if there is a change. In our model this would be realized by an Event Trigger which monitors a certain set of Information Spaces and triggers events as soon as differences are detected. A user will then set up a rule that checks if the changes are related to a certain category of interest, such as sports or a certain observed site, and how big those changes are. If for example a Wikipedia article changed in more than 10% of its content this will be a reason for a moderator to look at it and should be entered as a task in the next free slot of his calendar, as shown in Figure 4.1.

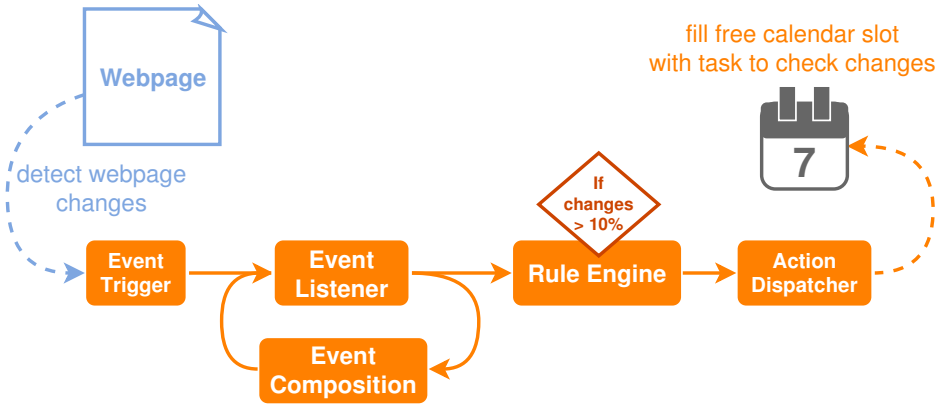


Figure 4.1: Use Case; detect big changes in the Web and fill calendar slot of moderator with task

4.2 Enhance existing Web Applications

Web applications, such as webmails, social networks or Content Management System (CMS) are widely spread and used by a large number of Internet users. Users or developers often miss some features or interoperability with other web applications, which would result in enhanced functionality and also in less work. Features of that kind could also include data and functionality from other Web Resource on the web. This would require Web Applications to communicate together and to grab data or impose functionality upon each other. A lot of enhancements will not be implemented by the Web Applications themselves because they are very specific to a small number of their users. With a reactive Information Systems, users and developers could realize such features on their own.

4.2.1 Enrich Content Management System Posts with Additional Information

Every new post to a Content Management System can be modeled as an event. In the case that a user would like to enrich such a CMS with knowledge from a remote resource, reactivity in the Web can be used to do it. Enhancing an existing CMS can be realized by a rule that evaluates new posts and checks whether knowledge tags are included in the post, which is shown in Figure 4.2. Whenever there are tags included within the post, the reactive entity will enrich the post with additional knowledge to these tags from a remote Web Resource of the user's choice.

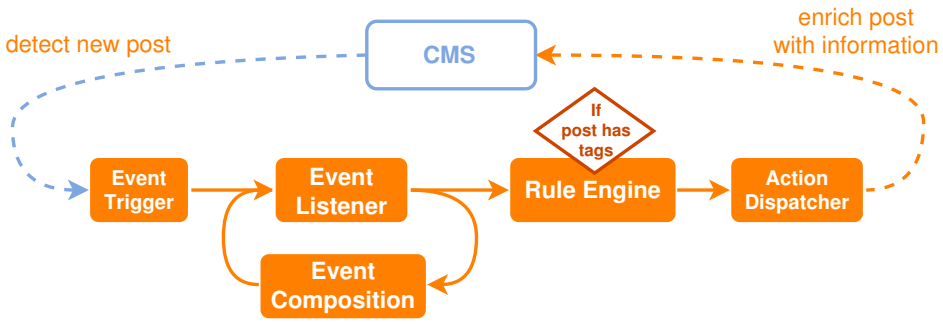


Figure 4.2: Use Case; enrich CMS post with remote knowledge data

4.2.2 Workflow Automation

Within such Web Applications, users often have very specific workflows. And because workflows always start with an event, they are predestined to be automated by a reactive entity. As an example for workflow automation, course and student exercise submission administration can be taken care of by a reactive entity. Figure 4.3 shows that whenever a new semester starts, the reactive entity will detect this through one of its rules and command an action dispatcher to set up infrastructures for courses. This can also include grabbing course data from an official webpage and including it into the infrastructure, thus eliminating the need for manual data copy tasks.

After setting up the semester courses, the reactive system is ready to process new student registrations for these courses. It automatically associates students into the afore mentioned infrastructures and sets up additional infrastructure such as an exercise submission container. Whenever a course tutor submits a new exercise to the course resource, the system will detect this and spread this information to the students, together with a deadline, as depicted in Figure

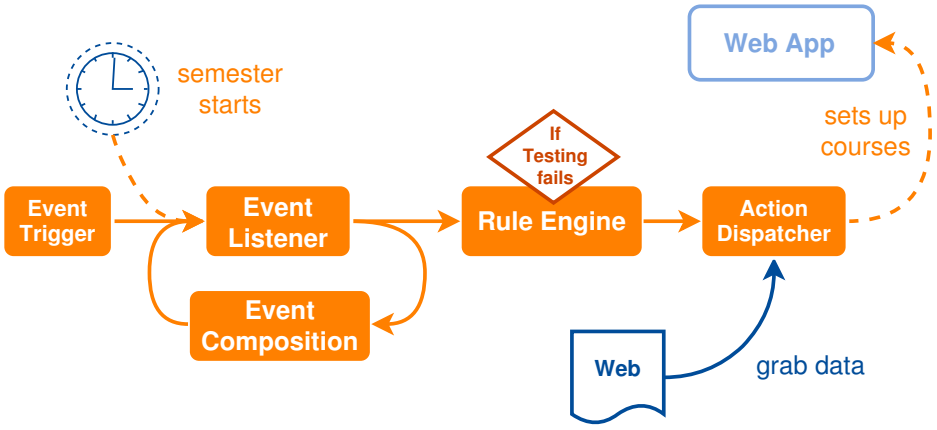


Figure 4.3: Use Case; create course resources at semester start

4.4. The students are expected to submit their exercise solutions before the deadline, to the exercise submission container, which was created reactively for them.

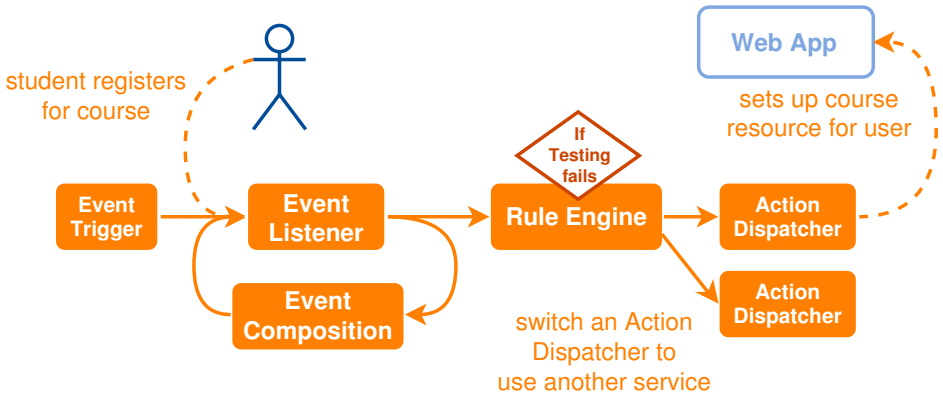


Figure 4.4: Use Case; create course resource for registered student

A certain amount of time before the deadline, e.g. one day, the reactive entity will detect the deadline and process events that depict the current exercise submission status per student. If the system detects students who have not uploaded their exercises yet, it will notify them about the deadline, which is shown in Figure 4.5. This is an additional service that gives students the chance to react on a missed exercise submission deadline. As soon as the deadline passed, the system will revoke write-rights to the exercise submission container and therefore disallow submissions which are too late.

4.3 Service Functionality and Availability Checking

Services offered through the Web are not monitored or tested by users or developers from other sites. If they rely on correct functionality or availability they need a way to assert this. It is also possible that an owner of such a service does not have the tools to monitor his own services. Whenever such a service is not working correctly anymore or stops responding, these users or developers need to be able to react contemporary on this. With a reactive rule in place that evaluates Service Testing results, countermeasurements can be taken early. One action to such a failing service test could be an automatic switching of the utilized service within an action

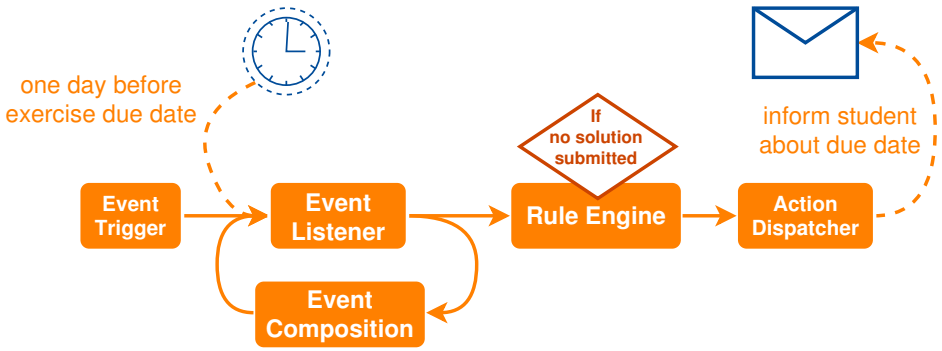


Figure 4.5: Use Case; notify student before exercise due date

dispatcher, so that from then on it uses one which still works correctly, as shown in Figure 4.6.

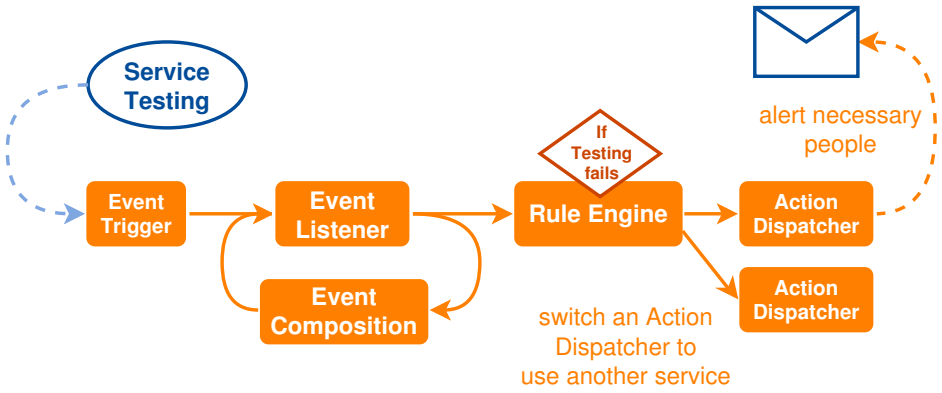


Figure 4.6: Use Case; test proper service functionality and availability

4.4 Exploiting the Web of Things

A model for reactive information systems becomes especially interesting in the context of the Web of Things. Through the small connected devices, a lot of sensor data become accessible via the Web and can be used as events to trigger actions. These actions could also be part of the Web of Things, if there are such things, that offer services. One example of a reactive rule, that has parts in the Web of Things, is that of a server room which has a defective cooling. The increasing temperature eventually causes the servers to shutdown or even fail. Servers in this room should push current state information into a reactive system. The reactive system can then take countermeasures if it detects a certain pattern that will lead to an overheating of all systems. It could inform certain (not so important) servers to gracefully shutdown and additionally inform administrators, who otherwise might miss the shutdown, as shown in Figure 4.7. It would be even better, if such a system would have the power to enable an additional emergency cooling system to prevent the shutdown of any of the servers.

Another scenario gets more realistic with the increasing number of homes that are connected to the Web. A home or apartment owner has her light controls attached to the Web. The first thing a reactive system could do, is that it detects holidays in the owner's agenda and automatically sets the light control to somewhat reasonable random during her absence. This would make suspicious characters, which are eventually interested in her wealth, think that he's still at home. In combination with another thing, that is connected to the Web and always accompanies people,

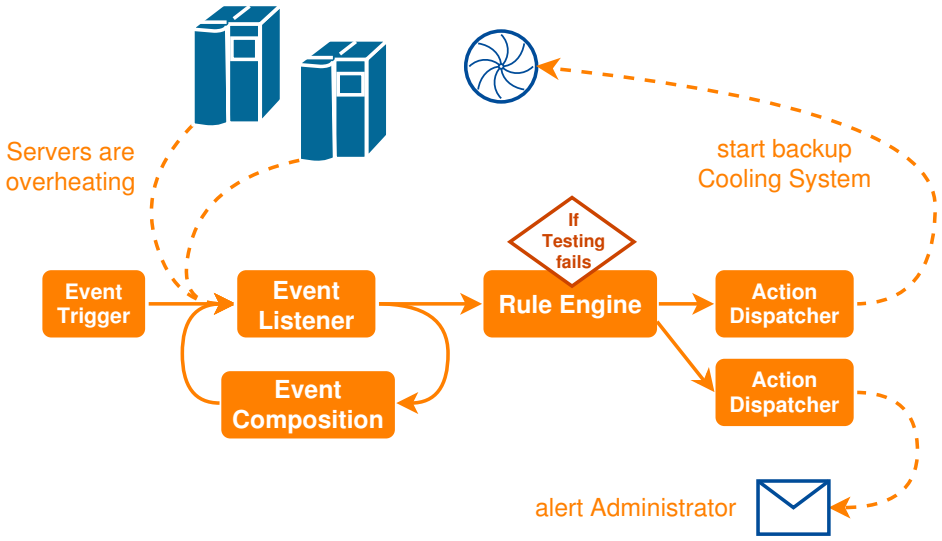


Figure 4.7: Use Case; measurements on server failure

the cell phone, an even more interesting application scenario can be thought of. The phone would push location information about the owner into the system. Whenever the owner gets close to her house, the reactive system could turn on the light in the entry area and play some music. On a Wednesday evening it could also inform the delivery service that they can deliver the owner's preferred menu when she is home, because the owner is doing this always on a Wednesday evening.

4.5 Averaged Bad Weather Prediction

There are a lot of different weather services existing in the Web. One has to check several of them in order to get an idea on how likely it is that it will be raining on the journey to work and back. By composing a higher-order event from several weather update events, a user could store a rule which alarms him early in the morning if more than 50% of the weather forecasts expect rain on the way to or back from work.

Chapter 5

Prototype System

We have so far introduced our conceptual model for reactive Information Systems and their Services and some example use cases to point out what would be possible with our model. In this chapter we present our proof of concept prototype system, which has a focus on the Web as its Information Space. We will then introduce our ECA rule language, which gives all the necessary power over our prototype system and which can be directly translated into the internal rules representation.

5.1 Architecture

The prototype system is the adoption of our conceptual model for reactive Information Systems and their services to the Web. The Web consists of many Information Systems and because of its Service-Oriented Architecture it can be seen as one large Information System, therefore we can impose reactivity on the Web. Since communication over services in the Web is often latency driven, we came to the conclusion that asynchronous communication and therefore scalability should be attributes our prototype system has to support natively. Another aspect to be regarded for the architectural decision was how the rules are going to be represented internally. We introduced XML and JSON as common ways to communicate data between services on the Web. Both formats represent data in a tree structure, and this is also what we decided to assume for the explicit parameters in the events that will enter our prototype. Together with the requirement of native support for an Event-Driven Architecture (EDA) our decision was to build upon the recent adoption of JavaScript to application development through Node.js[1] and its human-readable JSON communication format.

The prototype system consists of several modules, shown in Figure 5.1, which we are going to introduce within this section:

- **Poller:** Loads Event Trigger modules and forwards events coming from them to the Event Queue. Event Trigger modules poll for changes in the Web and transform them into events.
- **Webhook Listener:** Listens on active Webhook for events and forwards them to the Event Queue.
- **Event Queue:** Buffers events for the case of an overly busy Rule Engine.

[1] http://nodejs.org/

- **Rules Engine:** Picks an event from the Event Queue whenever there is one and it is idle.
- **User Request Handler:** The user interface modules to administrate Event Triggers, Webhook, Rules and Action Dispatchers.

When started, the prototype system loads persisted Webhook and begins to listen for new events on them. The Rule Engine then loads all persisted rules and for each rule it loads the required Action Dispatchers and notifies the Poller about the new rule, which in turn loads an Event Trigger if required. The prototype is now up and running and accepts administration requests for Event Triggers, Webhook, Rules and Action Dispatchers. Whenever a rule is created or updated, the Poller and Rule Engine load required Event Triggers or Action Dispatchers.

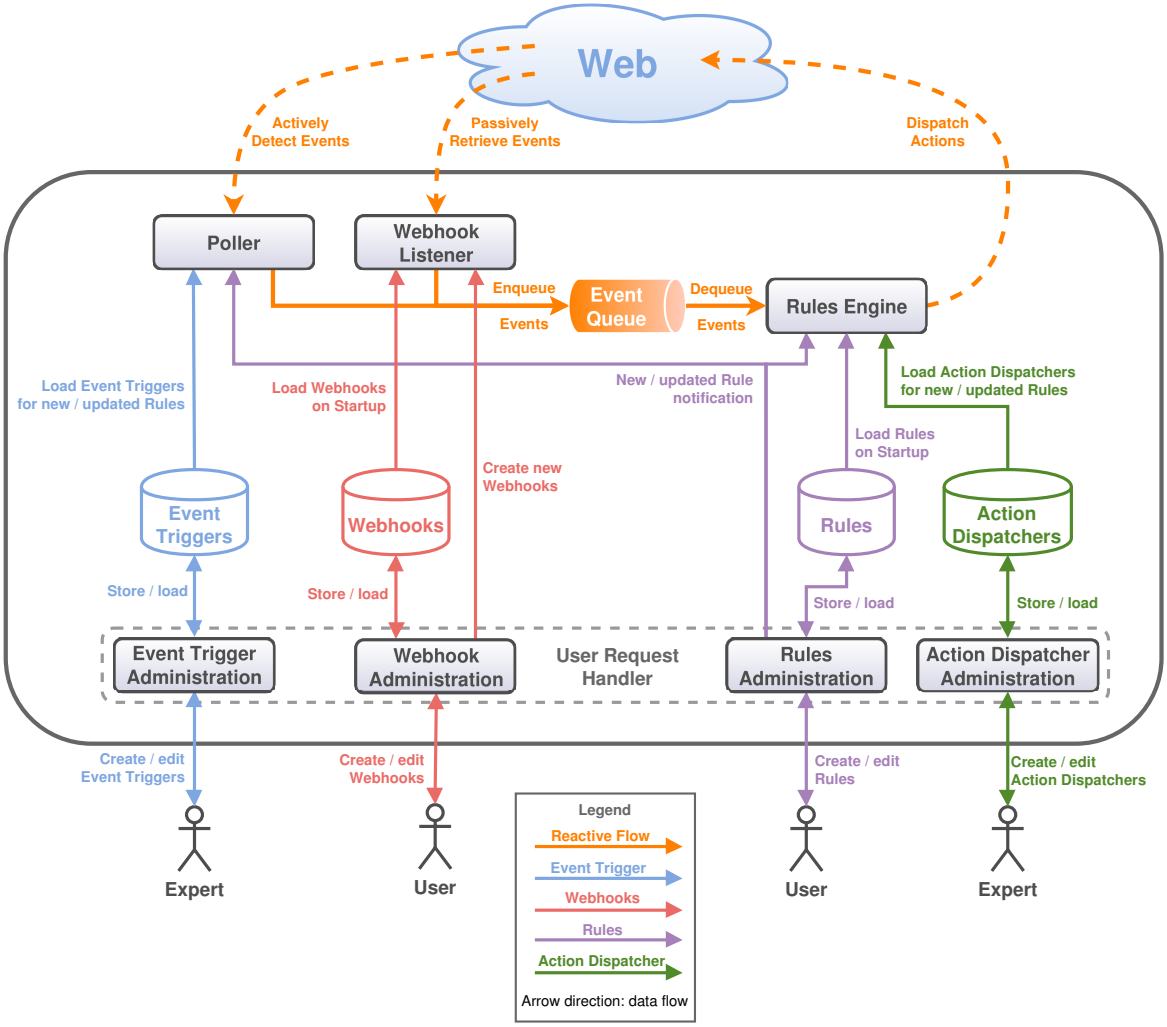


Figure 5.1: Prototype System Architecture

5.1.1 Data Structure for Event Parameters

Events in our prototype system are internally represented as tree structures in JSON format. The JSON format builds on two data structures:

- Objects: Unordered collections of name/value pairs wrapped in curly brackets { }, which can also be implemented as a hash map, dictionary or struct in other languages.

- Arrays: Ordered lists of values wrapped in brackets `[ ]`, which can also be implemented as a record, vector or list in other languages.

A value can be an object or an array, but also a unicode string, a number, a boolean value or null. This allows for any arbitrary depth and chaining of the supported data representations. It is a handy feature when we assume a tree structure for events in our prototype system since the selection of nodes in tree structures has been well studied and useful libraries exist. JSON formatted datastructures can easily be marshaled into one string and communicated to other applications without overhead Also, they are natively supported within JavaScript programming code. JSON can be implemented in virtually every programming language, therefore received a lot of attention and is supported by many Web Resources for communication.

5.1.2 Dynamic Code Loading for Event Trigger and Action Dispatcher

During our research we have seen many different Web services with thoroughly different requirements in terms of communication. The cleanest category among them were the Web APIs with their RESTful services. And still, in many cases it is only possible to access data which does not refer to an event. To detect a change in data over time we need to be able to store an earlier request and get the difference to the current request, which could then be transformed into a meaningful event. The derivation of a meaningful event from data can be a complex task which requires certain operations, which underlines the need for powerful Event Trigger modules. The complexity is even bigger for Action Dispatcher modules which alter data and require more complex communication to Web services. For those reasons we made the decision to keep these modules flexible in terms of communication. We also wanted to leave it open to expert users to encapsulate complicated logic into them for inexperienced users. We modeled the Event Trigger and Action Dispatcher to be JavaScript code modules, which are created by expert users during runtime. They are also loaded during runtime whenever an activated rule needs them. These modules run in a sandbox and got only access to certain JavaScript libraries, which are provided by the owner of the system. Through this it is possible to communicate with RESTful Web services as well as with SOAP Web Services or any other service that can be addressed through JavaScript libraries. Apart from those libraries there are two other important functions offered in both type of modules:

- log: Will store log entries on a per rule base, wherever the instruction is met during execution of a module. The log can be seen by the user who chose the module to be part of a rule.
- pushEvent: This funciton is an important part of Event Trigger modules. It is responsible to push events into the prototype system. For Action Dispatcher modules this provides the possibility for loopback events.

Only functions which are attached to the `exports` property of the module are later accessible from the outside and can be selected as Event Trigger or Action Dispatcher. The function arguments of, from outside visible, functions are identified by the according User Request Handler module and the user will be requested to provide values for all of them in order to activate the Event Trigger or Action Dispatcher. For Action Dispatchers it is also possible to use event property selectors as arguments and thus allows the passing of event data to the Action Dispatchers. Through the `pushEvent` function in the global scope of the modules, events can be pushed into the prototype system.

By using the expressiveness of JavaScript and some of its libraries, it is possible to access a large part of the existing Information Systems and transform changes in their Information

Spaces into events and also to impose changes onto them as part of actions. The power that can be expressed in those code modules needs to be controlled, and ca not be granted to anybody, thus we shield it through user access control, thus only allowing trusted users to write Event Trigger and Action Dispatcher code.

5.1.3 Retrieving Events

In the last chapters, we put emphasis on the two different ways how events can be retrieved from Information Systems, i.e. actively pulling events, or passively retrieving them. Some Information Systems offer access to data that corresponds to events and can instantly be forwarded into the system. But we have seen that there is need for the derivation of events from changes in data on the Web, therefore we need the Event Trigger modules. But still, our vision is that of an optimal real-time reactive Web which means that all possible events are offered by all Information Systems. An interested remote entity could announce interest in a certain kind of events over a Webhook and would retrieve them in real-time. For that reason we laid out our architecture for Webhooks, but still offer the Event Trigger modules to poll for events in the semi-static World Wide Web. During prototype testing we focused mainly on server-sided Web APIs, but we also generated events from the browser and pushed them to our prototype system. This was achieved with a library included in a sample webpage that pushed events to a Webhook of our prototype. Since modern browsers support geo locating, we decided to let the client browser push the current position of the device to the Webhook.

Polling with Event Triggers

As we have pointed out before, Event Trigger modules are dynamic code modules with access to a set of predefined libraries. The Poller loads Event Trigger modules whenever they are required in an active rule. The user of the Event Trigger can choose a starting point and an interval for the polling to take place. An example Web service which offers polling for events is the Email Yak[2] Web API, which responds with new emails when requested. The code required to request the new mails from this service and forward them into the prototype system is quite short and shown in Listing 5.1. For other services it can quickly get more complex, depending on how complicated a meaningful change detection is. For better readability the code is written in CoffeeScript[3]. Only expert users are expected to store such a piece of code in our prototype, which enables inexperienced users to simply choose the `"EmailYak -> newMail"` Event Trigger for their rule. A great opportunity to access data from webpages via a Web API is Import.io[4]. By browsing through the Web with the Import.io browser, it is possible to select certain parts from a webpage and store the selection as a mask. Data is instantly extracted from the webpage, using the stored mask, when sending a request to their Web API with the given mask id and the URI. This is a great tool for expert users to predefine desired data on webpages and then produce events out of an Event Trigger whenever there is a change in that data.

Webhook

As powerful as their ability to provide real-time notifications from remote Web Resources is, as simple are Webhooks to use. In our prototype, users can create as many new Webhook as they

[2] http://www.emailyak.com/
[3] http://coffeescript.org/
[4] https://import.io/

```
url = "https://api.emailyak.com/v1/#{params.apikey}/json/get/new/email/"
exports.newMail = () ->
  needle.get url, ( err, resp, body ) ->
    if not err and resp.statusCode is 200
        pushEvent mail for mail in body.Emails
```

Listing 5.1: Event Trigger code to poll Email Yak RESTful Web service for new Mails; written in CoffeeScript

like. They only need to provide an event name which will be associated to the Webhook. A new Webhook is created in the Webhook Listener, which from then on accepts events posted to it. The Webhook URI is always accessible to the user and can be placed at any desired Webhook in order to receive events from it. Any Web Resource that supports the concept of Webhook (e.g. GitHub[5]) has a place to register the Webhook URI. Whenever a remote Web Resource pushes an event to the Webhook, the user-defined event name is assigned as the implicit parameter of a freshly created internal event. The whole incoming event body is added as explicit parameters to the internal event, which is forwarded to the Event Queue.

5.1.4 Dispatching Actions

Action Dispatchers are JavaScript code modules that can be created during runtime and loaded by the engine whenever a new rule requires them, much as the Event Trigger modules are loaded by the Poller. Before actions can be used in a rule, an expert has to create them. In our prototype system, Action Dispatchers use a library for HTTP communication which allows them to address a wide range of Web Resources. Action Dispatchers can also push events back into the Event Queue which can be used to chain certain rules together. Since Webhooks are an important part of our vision we also implemented an Action Dispatcher that delivers events to external Webhook. Action Dispatchers need to have functions attached to their `exports` property so that they are visible from the outside and can be selected as actions, such as the `newContent` function in the example Listing 5.2.

```
urlService = 'https://probinder.com/service/'

requestService = ( args ) ->
  url = urlService + args.service + '/' + args.method
  needle.post url, args.data

exports.newContent = ( companyId, contextId, content ) ->
  requestService
    service: 'content'
    method: 'save'
    data:
      companyId: companyId
      context: contextId
      text: content
```

Listing 5.2: Action Dispatcher code to store a new content on the ProBinder RESTful Web service; written in CoffeeScript

[5] https://developer.github.com/Webhook/

5.1.5 ECA Rules in the Rule Engine

While a car engine converts potential energy into mechanical work, our Rule Engine converts events into changes in Information Systems. We have introduced ECA rules as sufficient approach to impose reactivity on systems and adopted the ECA paradigm for our conceptual model. For our prototype this means that the Rule Engine requires user-defined ECA rules which are compared against incoming events. The Rules Administration within the User Request Handler notifies the Rules Engine about new or updated rules from the user, which then in turn loads required Action Dispatcher modules. For each event in the Event Queue, the engine checks it against its stored ECA rules and dispatches actions whenever the event conforms to the rule's condition part. The three parts of an ECA rule have the following requirements in our prototype system:

- Event name: Any arbitrary Unicode string, can refer to the name of an Event Trigger or a Webhook, but also to a custom loopback event.
- Conditions: Zero or more instructions to be evaluated against an event. Requires a selector for a node in the tree structure of the event, a comparison operator ($<$, $<=$, $>$, $>=$, $==$, $!=$ or $instr$) and a value.
- Action Dispatchers: A list of Action Dispatchers to be invoked if all conditions of the given event evaluate to true. We assume that invocations can be expressed using common function invocation syntax (i.e. `actionFunction(param1, param2[, ...])`) in order to dispatch an action.

A valid rule in the internal JSON representation is shown in Listing 5.3, where we used the predefined EmailYak Action Dispatcher to send a mail to an interested person whenever news about soccer are detected.

Parameter Selectors for Events

Tree node selectors for event parameters are used in conditions to select a parameter which is evaluated. The selectors can also be used to pass event parameters as arguments to the Action Dispatchers. Event tree node selectors for Action Dispatcher arguments are defined by wrapping them into curly brackets and prepended with a hash: `"#{ [selector] }"`. Since an existing JavaScript library[6] is used to find event parameters with selectors, the following selectors are available[7]:

- *** :** Any node
- **T :** A node of type T, where T is one string, number, object, array, boolean, or null
- **T.key :** A node of type T which is the child of an object and is the value its parents key property
- **T:root :** A node of type T which is the root of the JSON document
- **T:nth-child(n) :** A node of type T which is the nth child of an array parent
- **T:nth-last-child(n) :** A node of type T which is the nth child of an array parent counting from the end

[6] https://github.com/harthur/js-select
[7] Explanations taken from http://jsonselect.org/, which is used by js-select

- **T:first-child :** A node of type T which is the first child of an array parent (equivalent to T:nth-child(1)
- **T:last-child :** A node of type T which is the last child of an array parent (equivalent to T:nth-last-child(1)
- **T:only-child :** A node of type T which is the only child of an array parent
- **T U :** A node of type U with an ancestor of type T
- **T > U :** A node of type U with a parent of type T
- **T ~ U :** A node of type U with a sibling of type T
- **S1, S2 :** Any node which matches either selector S1 or S2
- **T:has(S) :** A node of type T which has a child node satisfying the selector S
- **T:val(V) :** A node of type T with a value that is equal to V
- **T:contains(S) :** A node of type T with a string value contains the substring

```
{
  "eventname": "news",
  "conditions": [
    {
      "selector": ".categories",
      "operator": "instr",
      "compare": "soccer"
    }
  ],
  "actions":[
    "EmailYak->sendMail(\"fan@soccer.com\",\"News about soccer!\",\"#{ .body }\")"
  ]
}
```

Listing 5.3: Rule Example expressed in JSON Format

5.2 A Rule Language for the Prototype System

So far, we introduced the internal representation of the ECA rule language used in our prototype system. For human readability and more intuitive writing, they can be transformed into a phrase representation, through which Listing 5.3 would be written as shown in Listing 5.4. Our language is descriptive and flexible in terms of the Event Trigger and Action Dispatcher modules. Another important flexible factor is the mapping of event properties to the Action Dispatchers. To write a rule it requires a priori information from the Event Trigger and Action Dispatcher modules, but we believe this can be offered intuitively to the user through today's Web Applications. Listing 5.4 shows an example phrase of our envisioned rule language where the retrieval of a new mail will be checked for soccer news and, if confirmed, the mail body will be forwarded to an interested person. The Extended Backus-Naur Form for the prototype rule language syntax is shown in Listing 5.5.

```
ON news
IF ".categories" instr "soccer"
DO EmailYak->sendMail("fan@soccer.com","News about soccer!","#{ .body }")
```

Listing 5.4: Example Phrase in Prototype Rule Language

```
expression   ::= "ON " event " IF " conditions " DO " actions
event        ::= word* ("->" word+)?
conditions   ::= condition (" AND " condition)*
condition    ::= string operator "'" string "'"
operator     ::= (" < "|" <= "|" > "|" >= "|" == "|" != "|" instr ")
actions      ::= action (", " action)*
action       ::= word* "(" (argument ("," argument)*)? ")"
argument     ::= "'" selstring "'"
selstring    ::= (word|selector|" ")
selector     ::= "#{" string "}"
string       ::= (word|special|" ")*
special      ::= [():.*>~,]
word         ::= [A-Za-z0-9_-]+
```

Listing 5.5: Extended Backus-Naur Form of Prototype Rule Language Syntax

5.3 Prototype Use Case Implementations

In the previous chapter we introduced possible reactivity examples, which have been inspired by our conceptual model. In this section we introduce use cases that were implemented in our prototype system in order to impose reactivity on the Web.

5.3.1 Detecting responding Computers

In our department, the offices are spread over the whole floor, with other departmens between them. This makes the coffee break coordination not a simple task. Thus a network scanner has been set up, which ping the department's Internet Protocol (IP) range in the morning and pushes the results as events into the prototype system. A rule checks if more than 42 pings are returned, which means there are enough people for a coffee break. There are not more than seven persons in that group, but there are about 35 servers running in this IP range as well. If it is the case, the Action Dispatcher deploys an email invitation to the group, suggesting a coffee break, as shown in Figure 5.2. The network scanner (code in Appendix A.1) was implemented as external Information System which pushes the ping results as event to a Webhook URL. The rule phrase is shown in Listing 5.6 and the corresponding rule object in JSON format, can be found in Appendix A.2. The simplified Action Dispatcher is shown in Listing 5.7. In this particular use case we were able to show how rules are kept simple through the use of an Action Dispatcher. Apart from inviting people to a coffee break, we were also able to gather interesting informations on computer uptimes when running the Event Poller continuously, as shown in Figure 5.3. This visualization of the accumulated event data inspires the thinking for new reactivity, for example into the direction of long term composite events, through which it could be possible to reduce energy consumption.

```
ON uptimestatistics
IF "#{ .currentlyon }" > 42
DO EmailYak->sendMail("eca-engine@mscliveweb.simpleyak.com", [email
    addresses], "Coffee Break!" , "Let's go for a coffee")
```

Listing 5.6: Rule Phrase for Coffee Break Invitation

```
url = 'https://api.emailyak.com/v1/'+params.apikey+'/json/send/email/'

exports.sendMail = ( sender, receipient, subject, content ) ->
  data =
    FromAddress: sender
    ToAddress: receipient
    Subject: subject
    TextBody: content
  needle.post url, data, json: true
```

Listing 5.7: Action Dispatcher; EmailYak, in CoffeeScript

Figure 5.2: Reactive Rule informs about the Presence of Team Members

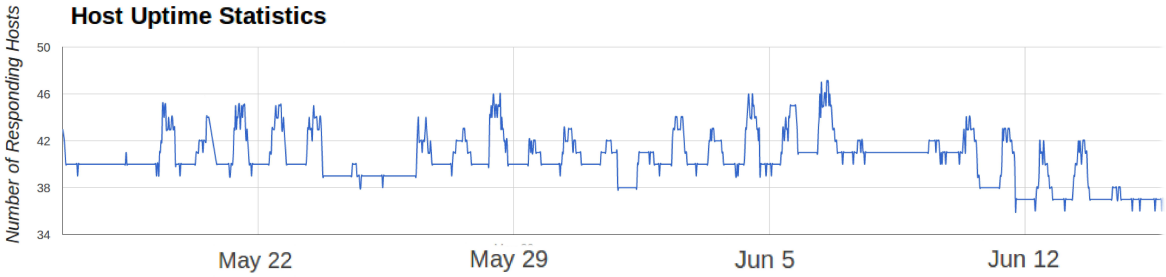


Figure 5.3: Group internal Computer Uptime Statistics

5.3.2 Enhance Existing Web Application

Several use cases have been implemented on top of the collaborative platform ProBinder. The one we introduce here, deals with the annotation of new content. Whenever users create new contents on the platform, they can tag them. Each user gets an unread flag for new contents. Since ProBinder provides a rich Web API, it is straight forward to access all the user's unread content. We created a rule in our prototype which detects new unread content, on behalf of a certain user, through an Event Trigger. The condition part of the rule checks whether the unread content is in a tab which we want to annotate. The Action Dispatcher of the rule checks all tags for existing knowledge entries on Wikipedia and adds a short summary as a comment to the fresh content entry. The rule phrase of the described use case is shown in Listing 5.8 and the corresponding JSON rule object in Appendix A.3. The ProBinder Event Trigger and Action Dispatcher modules can be found in Appendix A.4 and A.5 respectively. The outcome of

a sample annotation is shown in Figure 5.4. Another version of the rule was implemented using a bot account where no condition was set. Wherever the bot was invited to it started to annotate tagged entries, which is quite a handy feature and users only need to invite that bot account to their resource in order to get reactivity.

```
1 ON ProBinder->unreadContent
2 IF "#{ .context .id }" == 18749
3 DO ProBinder->annotateTagEntries("#{ .id }"),
4    ProBinder->setRead("#{ .id }")
```

Listing 5.8: Rule Phrase for ProBinder Annotations

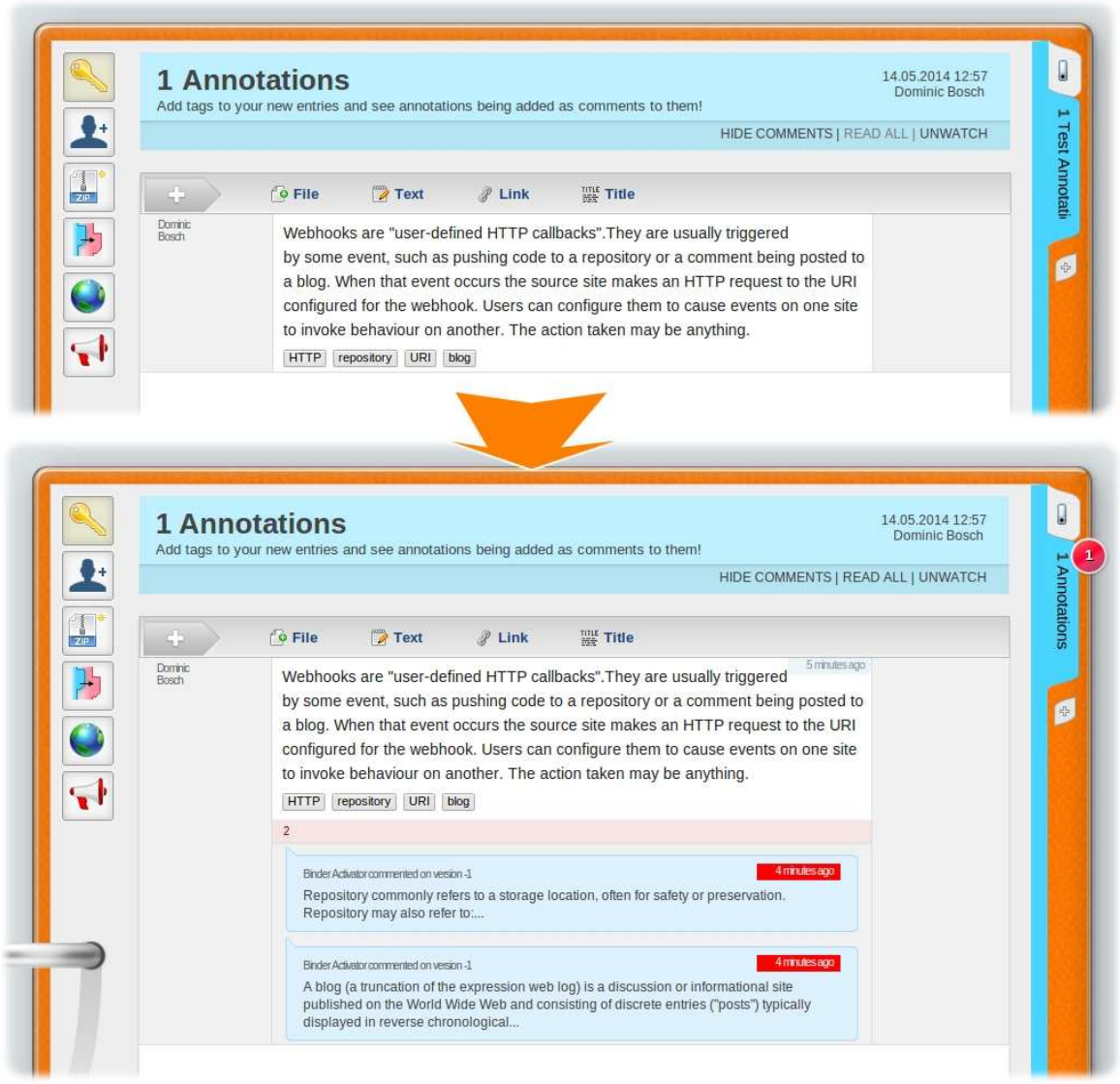


Figure 5.4: UC Binder Annotation

5.3.3 Remote System and Service Testing

It is important for a provider of a Web services to know whether the service is still running. In addition, a Web services provider has to ensure that the service is also running correctly. With our prototype system we are able to create a simple Event Trigger that queries the services, for example in a unit test like manner. Thus through our prototyp system it is possible to effectively to check such services for availability and consistency without the need for an expensive product. Whenever one of the test fails, an instant notification is sent to the Web services provider so he can react instantly. The unit testing Event Poller (code in Appendix A.6) is checking for some functions to work properly and emits an event with a success property. The condition of the rule, shown in Listing 5.9, only pushes a notification to the mobile device if the testing failed. A log is always stored in a binder (Figure 5.5) to check when the tests ran, which was set to a daily basis for this rule.

```
1 ON APITester->testProBinder
2 IF "#{ .success }" == false
3 DO Pushover->broadcast("#{ .summary }")
```

Listing 5.9: Rule Phrase for ProBinder WebAPI Testing

Figure 5.5: ProBinder Service Testing Log

5.4 Web Application Development

Event-driven programming is as imporant as asynchronous calls to remote Web services in order to impose real-time reactivity on top and between Information Systems. Another advantage which we have not mentioned yet was that this decision allowed us to focus on only one programming language for both the back- and the front-end of our prototype system. The front-end is the administration for rules, Webhooks, Event Trigger and Action Dispatchers. It is a Web Application which loads all required data through asynchronous calls to the prototype system's Web service while the user is interacting with the webpage. A concept, which caught our attention during the development process, is the concept of closures. Closures are the necessary concept that provides a context for each returning asynchronous call, even if it would have been garbage collected already.

5.4.1 Callback Functions & Asynchronous Closures

Often, optimization approaches and programming language concepts require special attention to avoid common pitfalls. When closures are used as asynchronous functions, developers need to be very careful not to end up with race conditions. Looking at an example of sequential code execution in Figure 5.6, we see that function execution of `fA` is halted until function `fB` is finished. If `fB` happens to be a latency-driven I/O operation the completion of `fA` could be deferred for a relatively long time. While the application waits for the completion of the I/O operation, some remaining operations in `fA` could eventually already be executed without causing any race conditions.

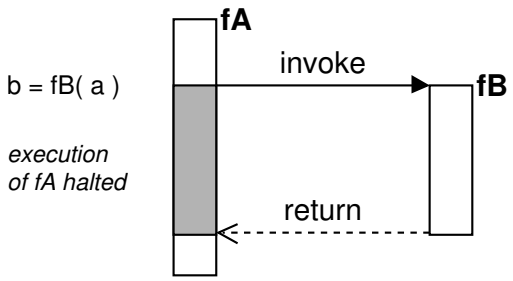


Figure 5.6: Synchronous Function Call

Asynchronous code execution, as shown in Figure 5.7, allows non-blocking and thus scalable applications. Non-blocking operations are a remedy for optimized resource allocation and open up ways to overcome previously described unnecessary resource bindings. Processing any kind of latency-driven I/O operation asynchronously (e.g. filesystem access and socket communication) exploits resources that would otherwise be bound while waiting for completion. Such operations are processed and completed whenever required resources are available.

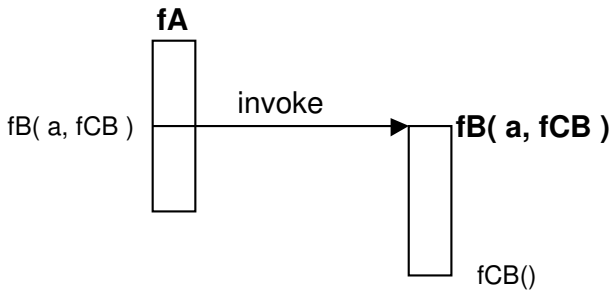


Figure 5.7: Asynchronous Function Call

Often other operations depend on the completion of asynchronous operations, hence their execution needs to be deferred. This necessary code execution deferral is achieved through the use of callback functions, denoted `fCB` in Figure 5.7. Any code placed in a callback function, which is assigned to an asynchronous operation, is only executed after the respective asynchronous operation completed. This allows stacking of functions and operations upon each other which automatically results in a flexible and event-driven application.

So far we did not regard the context for such asynchronous functions. If a function has access to the enclosing context where it was invoked in, it is called a closure. Closures play an important role in ECMAScript[17], which is the base for widely-spread script languages like JavaScript, JScript and ActionScript. Closures in ECMAScript are defined such as they have access to the

context of the function they were created in. This is shown in Figure 5.8 where c from `fA`'s context is accessible from within `fB`, assuming that `fB` was created in `fA` and not only invoked from there. Closures make it necessary for the context of the outer function to survive past its execution so no references are broken. This is labeled "extended context lifetime" in Figure 5.8. Using asynchronous closures it becomes evident, that the context in the invoking function can change while the closure is still computing and eventually referencing the outer context, thus causing race conditions. This will be most obvious in a loop that immediately invokes `fB` several times, as shown in Figure 5.9. In such a setup c will have different values in the same part of different invocations of `fB`.

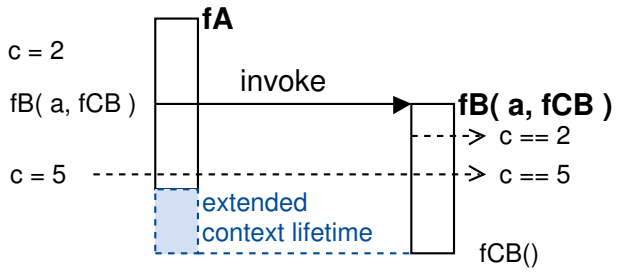


Figure 5.8: Closure Scope and referenced context

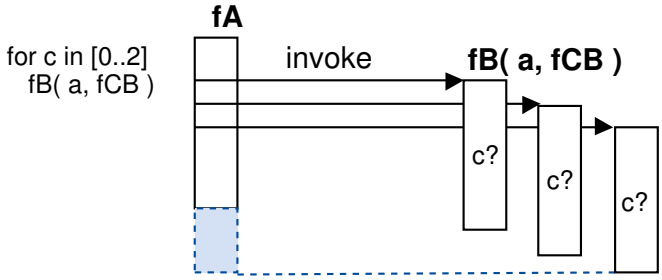


Figure 5.9: Closure context changes in a loop

Those event-driven context overwrites can be taken care of by shielding the closure from context changes, as shown in Figure 5.10. To shield the closure form context changes, closure `fB` needs to create another closure `fC` and return it to `fA`. The argument passed to `fB` is the context (c in Figure 5.10) that might change but requires to be persistent for one invocation. `fC` has now c as a fixed context, which ca not be overwritten anymore. Now the only thing left is `fC` needs to be invoked and it will retain the original context. This implementation is necessary when the closure acts as a callback function for asynchronous operations, to preserve the original context in case it is required within the callback function.

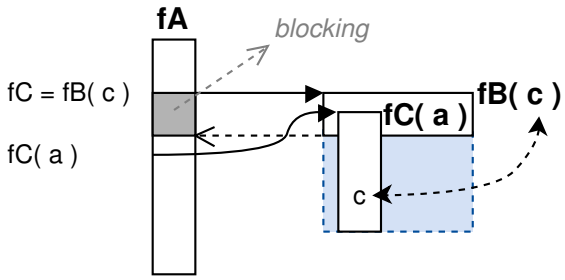


Figure 5.10: Closure Context Shielding

An example of how closure contexts can be shielded is shown in the Listing 5.10.

```
var fB = function( c ) { // Declare a function...
  var fC =  function( a ) { // ( <-- function to return )
    console.log( c );
  };
  return fC;
};
for( var c = 0; c < 100; c++ ) {
  // ... before you assign it to an event happening in the future:
  var fC = fB( c );
  setTimeout( fC, 3000); // will be executed after the loop ended
}
```

Listing 5.10: JavaScript Closure Context Shielding

Chapter 6

Conclusions & Future Work

The practical use case examples for our lightweighted prototype system showed the diversity of examples that can be run in it. As soon as there are services that allow the access to an Information Systems, we are able to orchestrate them, which is a promising insight. But even though the Web gets ever more programmable it is still a tedious task for many Web services to get into communication. Moreover it is a time consuming task to find the proper functionality in Web APIs, or certain functionality needs to be chained in order to get either to meaningful events or actions, or last but not least some functionality is just not accessible. RESTful Web services are a good example for lacking meaningful functionality, since they do not provide complex functionality but more or less read, write, update and delete logic for Web Resources. For those reasons, we believe it was a good decision to encapsulate sometimes complex logic into the Event Trigger and Action Dispatcher modules. Through this, also users that do not have a background in computer science are more likely able to forge their customzied reactive Web.

Many of the notional use case examples in chapter 4 have been implemented in the prototype system, but have brought important insight where it did not work instantly. We found that it is difficult to detect changes on any arbitrary webpage in a useful way. Our first attempt was to use a diff comparison utility, of which the result was hard to process because of the HTML tags. Another approach was to create an object tree from the HTML document and calculate the difference between the last recorded object, which was more promising to detect changes all over a webpage. The simplest approach was to use Import.io to define a part of a webpage to be observed and then detect changes only on this predefined part of the webpage over the Import.io Web API. We believe that there is a lot of value in detecting changes on static Web Resources in a proper way. Future research could mold webpage change detection into some sort of an Event Trigger or even into a novel rule language that uses Web queries in the event part. We were able to realize the enhancement of an existing Web Application in large parts since proper Web APIs provide powerful tools to weild them.

An interesting insight was, that sometimes users whish for activity at a certain point in time instead of event-driven reactivity, which means that the reaction on time events seems to be desirable for users. This could be handled in two ways through our model, either an Event Trigger is created which pushes an event at the exact point in time into the system, or an external Information System is pushing continuously primitive time events into our model over a Webhook. The first option seems to be a bit of an overkill but means lesser event load for the model. The latter option means a certain load depending on how small the event intervals are, but it reflects more the reality where we are used to time events in an interval of one second. Through continous time events it is also easily possible for all users to setup rules for their desired point in time without having to select and parameterize an Event Trigger beforehand.

Our experiences with the prototype system show us that the conceptual model is suitable to impose real-time reactivity on existing Information Systems. We were successful in capturing events from Information Systems, be it by actively pulling them over a service or passively getting it delivered over a Webhook. A somewhat surprising finding was how few Web APIs support Webhooks for real-time notifications to external Information Systems. We envision a future where the whole Web is event-driven and events are directed to any Information System, which is interested in them. This would be the optimal case for effective real-time notifications and thus reactivity on the Web. If the Web APIs remain passive without support for Webhooks, this will cause unnecessary computation and communication cost because of the polling that needs to be done.

A field which turned out to be beyond the scope of this thesis, but was intended to be part of the prototype as well, was the field of CEP. Temporal composition of events should be taken care of in future research in this field since it allows to identify situations out of primitive events. And it also allows to define semantically ever more complex situations out of existing ones. With a growing number of events and their compositions, that are flowing through such a system, it would be useful to have an event relation describing framework on top of them, much like RDF for Web Resources. Also with a growing popularity of a reactivity imposing entity in the Web, other systems will start to deliver events into that entity, which requires detection of new events and information of users about them.

We believe that the Web of Things is a very promising field for such a reactivity imposing entity, but we were not able to study it in depth due to the lacking accessability of such things. By reducing the interaction in term of event detection and action imposition to RESTful, such as it is used for the Web of Things, it might be possible to incorporate the create, read, update and delete interactions with Web services into the rule language. This would allow for an abstraction away from the Event Trigger and Action Dispatcher modules. It would add stability and a generalization in a way that rules would become more complex to be implemented, but no more expert users are required to implement the code modules. By using the read operation of the RESTful HTTP for webpages, it could be possible to define the detection of changes on them. Since it can be complex to derive an event and express an action with just one call to a service, it might be necessary to allow Web query chaining. Such Web queries could then also be used in the condition section of a rule.

We have seen that the ECA paradigm is very well suitable to impose reactivity to Information Systems and even in an intuitive way for the user to create rules. The more challenging thing was to identify meaningful events that can be added to a rule by the user. We have seen that it is common for existing ECA approaches to impose actions on local data rather than on remote Information Systems over their services. We defined a conceptual model that does not need any Information Systems to understand events but orchestrates it flexibly over its existing services.

Bibliography

[1] Raman Adaikkalavan and Sharma Chakravarthy. Event Specification and Processing for Advanced Applications: Generalization and Formalization. In Roland Wagner, Norman Revell, and Günther Pernul, editors, *Database and Expert Systems Applications*, volume 4653 of *Lecture Notes in Computer Science*, pages 369–379. Springer Berlin Heidelberg, 2007.

[2] Mert Akdere, Uğur Çetintemel, and Nesime Tatbul, Plan-based complex event detection across distributed sources. *Proceedings of the VLDB Endowment*, 1(1):66–77, 2008.

[3] JoséJúlio Alferes and Ricardo Amador. r 3– A Foundational Ontology for Reactive Rules. In Robert Meersman and Zahir Tari, editors, *On the Move to Meaningful Internet Systems 2007: CoopIS, DOA, ODBASE, GADA, and IS*, volume 4803 of *Lecture Notes in Computer Science*, pages 933–952. Springer Berlin Heidelberg, 2007.

[4] Darko Anicic, Paul Fodor, Sebastian Rudolph, Roland Stühmer, Nenad Stojanovic, and Rudi Studer. A Rule-Based Language for Complex Event Processing and Reasoning. In Thomas Lukasiewicz Pascal Hitzler, editor, *Web Reasoning and Rule Systems - Fourth International Conference*, volume 6333 of *LNCS*, pages 42–57. Springer, September 2010.

[5] Alistair P. Barros and Marlon Dumas, The Rise of Web Service Ecosystems. *IT Professional*, 8(5):31–37, 2006.

[6] Tim Berners-Lee, Robert Cailliau, Jean-François Groff, and Bernd Pollermann, World-Wide Web: The Information Universe. *Electronic Networking: Research, Applications and Policy*, 1(2):74–82, 1992.

[7] Tim Berners-Lee, James Hendler, Ora Lassila, et al., The semantic web. *Scientific american*, 284(5):28–37, 2001.

[8] Andrew D. Birrell and Bruce Jay Nelson, Implementing Remote Procedure Calls. *ACM Trans. Comput. Syst.*, 2(1):39–59, February 1984.

[9] Harold Boley. The RuleML family of web rule languages. In *Principles and Practice of Semantic Web Reasoning*, pages 1–17. Springer, 2006.

[10] Don Box, David Ehnebuske, Gopal Kakivaya, Andrew Layman, Noah Mendelsohn, Henrik Frystyk Nielsen, Satish Thatte, and Dave Winer. Simple object access protocol (SOAP) 1.1, 2000.

[11] Tim Bray, Jean Paoli, C Michael Sperberg-McQueen, Eve Maler, and François Yergeau, Extensible markup language (XML). *World Wide Web Consortium Recommendation REC-xml-19980210. http://www. w3. org/TR/1998/REC-xml-19980210*, 1998.

[12] Francois Bry and Paula lavinia Patranjan, Reactivity on the Web: Paradigms and Applications of the Language XChange. *J. of Web Engineering*, 5:2006, 2005.

[13] François Bry and Michael Eckert. Twelve Theses on Reactive Rules for the Web. In Torsten Grust, Hagen Höpfner, Arantza Illarramendi, Stefan Jablonski, Marco Mesiti, Sascha Müller, Paula-Lavinia Patranjan, Kai-Uwe Sattler, Myra Spiliopoulou, and Jef Wijsen, editors, *Current Trends in Database Technology – EDBT 2006*, volume 4254 of *Lecture Notes in Computer Science*, pages 842–854. Springer Berlin Heidelberg, 2006.

[14] Cinzia Cappiello, Maristella Matera, Matteo Picozzi, Gabriele Sprega, Donato Barbagallo, and Chiara Francalanci. DashMash: A Mashup Environment for End User Development. In Sören Auer, Oscar Díaz, and GeorgeA. Papadopoulos, editors, *Web Engineering*, volume 6757 of *Lecture Notes in Computer Science*, pages 152–166. Springer Berlin Heidelberg, 2011.

[15] Erik Christensen, Francisco Curbera, Greg Meredith, Sanjiva Weerawarana, et al. Web services description language (WSDL) 1.1, 2001.

[16] S. Duquennoy, G. Grimaud, and J.-J. Vandewalle. Consistency and scalability in event notification for embedded Web applications. In *Web Systems Evolution (WSE), 2009 11th IEEE International Symposium on*, pages 89–98, Sept 2009.

[17] ECMA International. *Standard ECMA-262 - ECMAScript Language Specification*. 5.1 edition, June 2011.

[18] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec, The Many Faces of Publish/Subscribe. *ACM Comput. Surv.*, 35(2):114–131, June 2003.

[19] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, 2000.

[20] N. H. Gehani and H. V. Jagadish. Composite event specification in active databases: Model and implementation. pages 327–338, 1992.

[21] Adrian Giurca and Emilian Pascalau, Json rules. *Proceedings of the Proceedings of 4th Knowledge Engineering and Software Engineering, KESE*, 425:7–18, 2008.

[22] Object Management Group. *The Common Object Request Broker (CORBA): Architecture and Specification*. Object Management Group, 1995.

[23] D. Guinard, V. Trifa, and E. Wilde. A resource oriented architecture for the Web of Things. In *Internet of Things (IOT), 2010*, pages 1–8, Nov 2010.

[24] Keman Huang, Yushun Fan, and Wei Tan. An Empirical Study of Programmable Web: A Network Analysis on a Service-Mashup System. In Carole A. Goble, Peter P. Chen, and Jia Zhang, editors, *ICWS*, pages 552–559. IEEE, 2012.

[25] Xuanzhe Liu, Yi Hui, Wei Sun, and Haiqi Liang. Towards Service Composition Based on Mashup. In *Services, 2007 IEEE Congress on*, pages 332–339, July 2007.

[26] Larry Masinter, Tim Berners-Lee, and Roy T Fielding, Uniform resource identifier (URI): Generic syntax. 2005.

[27] Gregory B Newby. Metric multidimensional information space. In *TREC*. Citeseer, 1996.

[28] George Papamarkos, Alexandra Poulovassilis, and Peter T Wood. RDFTL: An event-condition-action language for RDF. In *Proc. of the 3rd International Workshop on Web Dynamics*, 2004.

[29] A. Paschke, H. Boley, B. Craig, and A. Kozlenkov. Rule Responder: RuleML-Based Agents for Distributed Collaboration on the Pragmatic Web, 2007.

[30] Adrian Paschke, Harold Boley, Zhili Zhao, Kia Teymourian, and Tara Athan. Reaction RuleML 1.0: Standardized Semantic Reaction Rules. In Antonis Bikakis and Adrian Giurca, editors, *Rules on the Web: Research and Applications*, volume 7438 of *Lecture Notes in Computer Science*, pages 100–119. Springer Berlin Heidelberg, 2012.

[31] Paula-lavinia Patranjan. *The Language XChange*. PhD thesis, Ludwig-Maximilians-Universität München, 2005.

[32] C. Peltz, Web services orchestration and choreography. *Computer*, 36(10):46–52, Oct 2003.

[33] Randall Perrey and Mark Lycett. Service-oriented architecture. In *Applications and the Internet Workshops, 2003. Proceedings. 2003 Symposium on*, pages 116–119. IEEE, 2003.

[34] P.R. Pietzuch, B. Shand, and J. Bacon, Composite event detection as a generic middleware extension. *Network, IEEE*, 18(1):44–55, Jan 2004.

[35] Paul Rademacher. HousingMaps. `http://www.housingmaps.com/`, 2005. Accessed: 2014-6-6.

[36] REWERSE - Reasoning on the Web with Rules and Semantics. `http://rewerse.net`. Accessed: 2014-05-08.

[37] Leonard Richardson and Sam Ruby. *RESTful web services*. O'Reilly Media, Inc., 2008.

[38] Sven Rizzotti and Helmar Burkhart. useKit: A Step Towards the Executable Web 3.0. In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, pages 1175–1176, New York, NY, USA, 2010. ACM.

[39] D Robins. Complex event processing. In *Second International Workshop on Education Technology and Computer Science. Wuhan*, 2010.

[40] Keith Seymour, Hidemoto Nakada, Satoshi Matsuoka, Jack Dongarra, Craig Lee, and Henri Casanova. Overview of GridRPC: A remote procedure call API for Grid computing. In *Grid Computing—GRID 2002*, pages 274–278. Springer, 2002.

[41] Kathryn T. Stolee, Sebastian Elbaum, and Anita Sarma, Discovering how end-user programmers and their communities use public repositories: A study on Yahoo! Pipes. *Information and Software Technology*, 55(7):1289 – 1303, 2013.

[42] P. Windley. *The Live Web: Building Event-Based Connections in the Cloud*. Cengage Learning PTR, 2011.

[43] Detlef Zimmer and Rainer Unland. On The Semantics Of Complex Events In Active Database Management Systems. pages 392–399. IEEE Computer Society Press, 1999.

Appendices

Appendix A

Use Case Code

A.1 Node.js Code to ping IP Range and push the Result to a Remote Server

```
# A node.js module to automatically ping an IP range and push the response result
# to a remote server

fs = require "fs"
ping = require "net-ping"
needle = require "needle"

remoteUrl = "http://ec2-54-196-2-15.compute-1.amazonaws.com"
fPushEvent = ( evt ) ->
  needle.post remoteUrl + "/measurements", JSON.stringify( evt ), ( err, resp, body ) ->
    if err or resp.statusCode isnt 200
      console.log "Error in pushing event!"
      console.log err
      console.log resp.statusCode
    else
      console.log "Successfully posted an event"

try
  histData = fs.readFileSync "histoappend.json", "utf8"
catch err
  console.error err
  console.error "Error reading historical data file"
  process.exit()

session = ping.createSession retries: 2
oSum = {}
if histData
  arrPings = histData.split "\n"
  try
    for strObj, i in arrPings
      if strObj isnt ""
        oTmp = JSON.parse strObj
        oSum[ oTmp.timestamp ] =
          sum: oTmp.sum
    if oTmp
      fPushEvent
        currentlyon: oSum[ oTmp.timestamp ].sum
        pingtimes: oSum

  catch err
    console.log "Error parsing histo data"
    console.log err

i = -1
ips = []
```

```
46  pingTime = (new Date()).toISOString()
47  fPollHosts = () ->
48    session.pingHost "192.168.1.#{ ++i }", ( err, target, sent, rcvd ) ->
49      if not err
50        ips.push target
51
52    if i is 255
53      i = -1
54      console.log """#{ (new Date()).toISOString() } | All ping requests returned (#{ips.length} answered),
55        pushing event into the system and starting again at 0"""
56
57      oSum[ pingTime ] = sum: ips.length
58      fPushEvent JSON.stringify
59        currentlyon: ips.length
60        pingtimes: oSum
61
62      oPing =
63        timestamp: pingTime
64        ips: ips
65        sum: ips.length
66
67      fs.appendFile "histoappend.json", JSON.stringify( oPing ) + "\n", "utf8"
68      pingTime = (new Date()).toISOString()
69      ips = []
70
71    setTimeout fPollHosts, 7000
72
73  fPollHosts()
```

A.2 Coffee Break Invitation Rule object

```
{
  "eventname": "uptimestatistics",
  "conditions": [
    {
      "selector": ".currentlyon",
      "operator": ">",
      "compare": 42
    }
  ],
  "actions": [
    "EMailYak -> sendMail(\"eca-engine@mscliveweb.simpleyak.com\",[usermaillist],
      \"Coffee Break!\",\"Let's go for a coffee at 10!\")"
  ]
}
```

A.3 ProBinder Annotation Rule in JSON Format

```
{
  "eventname": "ProBinder -> unreadContent",
  "conditions": [
    {
    "selector": ".context .id",
    "operator": "==",
    "compare": 18749
    }
  ],
  "actions": [
    "ProBinder -> annotateTagEntries(\"#{ .id }\")" ,
    "ProBinder -> setRead(\"#{ .id }\")"
  ]
}
```

A.4 ProBinder Event Trigger

```coffee
# ###
# ProBinder EVENT POLLER
# ----------------------

# Global variables
# This module requires user-specific parameters:

# - username
# - password
# ###
urlService = 'https://probinder.com/service/'
credentials =
  username: params.username
  password: params.password

#
# The standard callback can be used if callback is not provided, e.g. if
# the function is called from outside
#
standardCallback = ( funcName ) ->
  ( err, resp, body ) ->
    if err
      log "ERROR: During function '#{ funcName }'"
    else
      if resp.statusCode is 200
        log "Function '#{ funcName }' ran through without error"
      else
        log "ERROR: During function '#{ funcName }': #{ body.error.message }"

# ###
# Call the ProBinder service with the given parameters.

#  - {Object} args the required function arguments object
#  - {Object} [args.data] the data to be posted
#  - {String} args.service the required service identifier to be appended to the url
#  - {String} args.method the required method identifier to be appended to the url
#  - {function} [args.callback] the function to receive the request answer
# ###
callService = ( args ) ->
  if not args.service or not args.method
    log 'ERROR in call function: Missing arguments!'
  else
    if not args.callback
      args.callback = standardCallback 'call'
    url = urlService + args.service + '/' + args.method
    needle.request 'post', url, args.data, credentials, args.callback

# ###
# Calls the user's unread content service.
# ###
exports.unreadContentInfo = () ->
  callService
    service: '36'
    method: 'unreadcontent'
    callback: ( err, resp, body ) ->
      if not err and resp.statusCode is 200
        pushEvent oEntry for oEntry in body
      else
        log 'Error: ' + body.error.message

# ###
# Fetches unread contents
# ###
exports.unreadContent = () ->
  exports.unreadContentInfo ( evt ) ->
    getContent
      contentId: evt.id
      contentServiceId: evt.serviceId
      callback: ( err, resp, body ) ->
```

```
70          if not err and resp.statusCode is 200
71            pushEvent
72              id: body.id
73              content: body.text
74              object: body
75          else
76            log 'Error: ' + body.error.message
77
78
79  # ###
80  # Calls the content get service with the content id and the service id provided.
81  # ###
82  getContent = ( args ) ->
83    if not args.callback
84      args.callback = standardCallback 'getContent'
85    callService
86      service: '2'
87      method: 'get'
88      data:
89        id: args.contentId
90        service: args.contentServiceId
91      callback: args.callback
92
93  #   Returns an event of the form:
94
95  #     {
96  #         "text": "test subject",
97  #         "id": 127815,
98  #         "createDate": "2014-04-19 16:27:45",
99  #         "lastModified": "2014-04-19 16:27:45",
100 #         "time": "5 days ago",
101 #         "userId": 10595,
102 #         "username": "Dominic Bosch",
103 #         "uri": "https://probinder.com/content/view/id/127815/",
104 #         "localUri": "https://probinder.com/content/view/id/127815/",
105 #         "title": "",
106 #         "serviceId": 27,
107 #         "userIds": [
108 #             10595
109 #         ],
110 #         "description": "",
111 #         "context": [
112 #             {
113 #                 "id": 18749,
114 #                 "name": "WebAPI ECA Test Binder",
115 #                 "remove": true,
116 #                 "uri": "/content/context/id/18749/webapi-eca-test-binder"
117 #             }
118 #         ]
119 #     }
```

A.5 ProBinder Action Dispatcher

```
# ###
# ProBinder ACTION INVOKER
# ------------------------

# Global variables
# This module requires user-specific parameters:

# - username
# - password
# ###
urlService = 'https://probinder.com/service/'
credentials =
  username: params.username
  password: params.password

#
# The standard callback can be used if callback is not provided, e.g. if
# the function is called from outside
#
standardCallback = ( funcName ) ->
  ( err, resp, body ) ->
    if err
      log "ERROR: During function '#{ funcName }'"
    else
      if resp.statusCode is 200
        log "Function '#{ funcName }' ran through without error"
      else
        log "ERROR: During function '#{ funcName }': #{ body.error.message }"

# ###
# Call the ProBinder service with the given parameters.

#  - {Object} args the required function arguments object
#  - {Object} [args.data] the data to be posted
#  - {String} args.service the required service identifier to be appended to the url
#  - {String} args.method the required method identifier to be appended to the url
#  - {function} [args.callback] the function to receive the request answer
# ###
callService = ( args ) ->
  if not args.service or not args.method
    log 'ERROR in call function: Missing arguments!'
  else
    if not args.callback
      args.callback = standardCallback 'call'
    url = urlService + args.service + '/' + args.method
    needle.request 'post', url, args.data, credentials, args.callback


# ###
# Does everything to post something in a binder

#  - {String} companyId the comany associated to the binder
#  - {String} contextId the binder id
#  - {String} content the content to be posted
# ###
exports.newContent = ( companyId, contextId, content ) ->
  if arguments[ 4 ]
    callback = arguments[ 4 ]
  else
    callback = standardCallback 'newContent'
  callService
    service: '27'
    method: 'save'
    data:
      companyId: companyId
      context: contextId
      text: content
    callback: callback

```

```coffee
# ###
# Does everything to post a file info in a binder tab

#  - {String} fromService the content service which grabs the content
#  - {String} fromId the content id from which the information is grabbed
# ###
exports.makeFileEntry = ( fromService, fromId, toCompany, toContext ) ->
  getContent
    serviceid: fromService
    contentid: fromId
    callback: ( err, resp, body ) ->
      content = "New file (#{ body.title }) in tab \"#{ body.context[0].name }\",
          find it here!'"
      exports.newContent toCompanyId, toContextId, content, standardCallback 'makeFileEntry'


# ###
# Calls the content get service with the content id and the service id provided.

#  - {Object} args the object containing the service id and the content id,
#    success and error callback methods
#  - {String} args.serviceid the service id that is able to process this content
#  - {String} args.contentid the content id
#  - {function} [args.callback] receives the needle answer from the "call" function
# ###
getContent = ( args ) ->
  if not args.callback
    args.callback = standardCallback 'getContent'
  callService
    service: '2'
    method: 'get'
    data:
      id: args.contentid
      service: args.serviceid
    callback: args.callback

# ###
# Sets the content as read.

#  - {Object} id the content id to be set to read.
# ###
exports.setRead = ( id ) ->
  callService
    service: '2'
    method: 'setread'
    data:
      id: id
    callback: standardCallback 'setRead'

getWikiTitle = ( title, cb ) ->
  titleUrl = 'http://en.wikipedia.org/w/api.php?format=json&action=query&prop=extracts&exintro&exchars=200&explaintext&tit
  needle.get titleUrl + encodeURIComponent( title ), ( err, resp, obj ) ->
    if err or resp.statusCode isnt 200 or not obj.query or not obj.query.pages or obj.query.pages['-1']
      cb new Error 'Unable to fetch data'
    else
      for id, page of obj.query.pages
        cb null, page.extract

getWikiSearch = ( text, cb ) ->
  searchUrl = 'http://en.wikipedia.org/w/api.php?format=json&action=query&list=search&srwhat=text&srlimit=3&srsearch='
  needle.get searchUrl + encodeURIComponent( text ), ( err, resp, obj ) ->
    if err or resp.statusCode isnt 200 or not obj.query or not obj.query.search or obj.query.search.length is 0
      cb new Error 'Nothing found'
    else
      for result in obj.query.search
        cb null, result.snippet.replace /<(?:.|\n)*?>/gm, ''

exports.annotateTagEntries = ( entryId, tags ) ->
  arrTags = tags.split( "#" ).slice 1
  for tag in arrTags
    tag = tag.trim().replace /\\"/g, ''
    tag = tag.trim().replace /\\/g, ''
```

```
142     fProcessTitleAnswer = ( loopTag ) ->
143       ( err, result ) ->
144         if not err and result
145           exports.commentEntry entryId, result
146         else
147           getWikiSearch loopTag, ( err, result ) ->
148             exports.commentEntry entryId, result
149
150     getWikiTitle tag, fProcessTitleAnswer tag
151
152 exports.commentEntry = ( entryId, text ) ->
153   log "Adding Commment '#{ text }' to ##{ entryId }"
154   callService
155     service: 'content'
156     method: 'addcomment'
157     data:
158       contentId: entryId
159       text: text
```

A.6 ProBinder WebAPI Testing Trigger

```
###

Tests the ProBinder API. Requires user credentials:

- username
- password

###
url = "https://probinder.com/service/"
arrFailed = []
testLog = ""
testTimeout = 15000
options = {}

exports.testProBinder = ( requestTimeoutMilliSeconds ) ->
  arrFailed = []
  testLog = ""
  testTimeout = parseInt( requestTimeoutMilliSeconds ) || testTimeout
  options =
    username: params.username
    password: params.password
    timeout: testTimeout

  oTestFuncs =
    testLogin: testLogin
    testNotLoggedInUnreadContentCount: testNotLoggedInUnreadContentCount
    testLoggedInUnreadContentCount: testLoggedInUnreadContentCount

  semaphore = 0
  for name, fTest of oTestFuncs
    semaphore++
    log "Testing function '#{ name }'"
    fTest () ->
      if --semaphore is 0
        testSuccess = arrFailed.length is 0
        if testSuccess
          summary = "All tests passed!"
        else
          summary = arrFailed.length + " test(s) failed: " + arrFailed.join ", "
        pushEvent
          success: testSuccess
          log: testLog
          summary: summary
        log summary

testLoggedInUnreadContentCount = ( cb ) ->
  turl = url + "user/unreadcontentcount"
  needle.get turl, options, responseHandler cb, 'testLoggedInUnreadContentCount', 200

testNotLoggedInUnreadContentCount = ( cb ) ->
  turl = url + "user/unreadcontentcount"
  needle.get turl, timeout: testTimeout, responseHandler cb, 'testNotLoggedInUnreadContentCount', 400

testLogin = ( cb ) ->
  turl = url + "auth/login/email/#{ params.username }/password/#{ params.password }"
  needle.get turl, timeout: testTimeout, responseHandler cb, 'testLogin', 200

responseHandler = ( cb, testName, expectedCode ) ->
  ( err, resp, body ) ->
    if err
      testLog += "<b> - FAIL | #{ testName }: Timeout! Server didn't answer within #{ testTimeout / 1000 } seconds</b><br/
      arrFailed.push testName
    else if resp.statusCode isnt expectedCode
      testLog += "<b> - FAIL | #{ testName }: Response
        #{ resp.statusCode }(expected: #{ expectedCode }), #{ body.error.message }</b><br/>"
      arrFailed.push testName
    else
      testLog += " + SUCCESS | #{ testName }<br/>"
    cb?()
```

Appendix B

Prototype System

B.1 Main Application for Startup

```
# ###

# WebAPI-ECA Engine
# =================

# >This is the main module that is used to run the whole application:
# >
# >     node webapi-eca [opt]
# >
# > See below in the optimist CLI preparation for allowed optional parameters `[opt]`.
# ###

# **Loads Modules:**

# - [Logging](logging.html)
logger = require './logging'

# - [Configuration](config.html)
conf = require './config'

# - [Persistence](persistence.html)
db = require './persistence'

# - [ECA Components Manager](components-manager.html)
cm = require './components-manager'

# - [Engine](engine.html)
engine = require './engine'

# - [HTTP Listener](http-listener.html)
http = require './http-listener'

# - [Encryption](encryption.html)
encryption = require './encryption'

# - [Event Poller](event-poller.html) *(will be forked into a child process)*
nameEP = 'event-poller'

# - Node.js Modules: [fs](http://nodejs.org/api/fs.html),
# [path](http://nodejs.org/api/path.html)
# and [child_process](http://nodejs.org/api/child_process.html)
fs = require 'fs'
path = require 'path'
cp = require 'child_process'

# - External Modules: [optimist](https://github.com/substack/node-optimist)
optimist = require 'optimist'
```

```
48
49  procCmds = {}
50
51  # ###
52  # Let's prepare the optimist CLI optional arguments `[opt]`:
53  # ###
54  usage = 'This runs your webapi-based ECA engine'
55  opt =
56  #  `-h`, `--help`: Display the help
57    'h':
58      alias : 'help',
59      describe: 'Display this'
60
61  #  `-c`, `--config-path`: Specify a path to a custom configuration file, other than "config/config.json"
62    'c':
63      alias : 'config-path',
64      describe: 'Specify a path to a custom configuration file, other than "config/config.json"'
65
66  #  `-w`, `--http-port`: Specify a HTTP port for the web server
67    'w':
68      alias : 'http-port',
69      describe: 'Specify a HTTP port for the web server'
70
71  #  `-d`, `--db-port`: Specify a port for the redis DB
72    'd':
73      alias : 'db-port',
74      describe: 'Specify a port for the redis DB'
75
76  #  `-s`, `--db-select`: Specify a database
77    's':
78      alias : 'db-select',
79      describe: 'Specify a database identifier'
80
81  #  `-m`, `--log-mode`: Specify a log mode: [development|productive]
82    'm':
83      alias : 'log-mode',
84      describe: 'Specify a log mode: [development|productive]'
85
86  #  `-i`, `--log-io-level`: Specify the log level for the I/O. in development expensive origin
87  #                          lookups are made and added to the log entries
88    'i':
89      alias : 'log-io-level',
90      describe: 'Specify the log level for the I/O'
91
92  #  `-f`, `--log-file-level`: Specify the log level for the log file
93    'f':
94      alias : 'log-file-level',
95      describe: 'Specify the log level for the log file'
96
97  #  `-p`, `--log-file-path`: Specify the path to the log file within the "logs" folder
98    'p':
99      alias : 'log-file-path',
100     describe: 'Specify the path to the log file within the "logs" folder'
101
102 #  `-n`, `--nolog`: Set this true if no output shall be generated
103   'n':
104     alias : 'nolog',
105     describe: 'Set this if no output shall be generated'
106
107 # now fetch the CLI arguments and exit if the help has been called.
108 argv = optimist.usage( usage ).options( opt ).argv
109 if argv.help
110   console.log optimist.help()
111   process.exit()
112
113 conf argv.c
114 # > Check whether the config file is ready, which is required to start the server.
115 if !conf.isReady()
116   console.error 'FAIL: Config file not ready! Shutting down...'
117   process.exit()
118
119 logconf = conf.getLogConf()
```

```
120
121 if argv.m
122   logconf[ 'mode' ] = argv.m
123 if argv.i
124   logconf[ 'io-level' ] = argv.i
125 if argv.f
126   logconf[ 'file-level' ] = argv.f
127 if argv.p
128   logconf[ 'file-path' ] = argv.p
129 if argv.n
130   logconf[ 'nolog' ] = true
131 try
132   fs.unlinkSync path.resolve __dirname, '..', 'logs', logconf[ 'file-path' ]
133 @log = logger.getLogger logconf
134 @log.info 'RS | STARTING SERVER'
135
136 # ###
137 # This function is invoked right after the module is loaded and starts the server.
138
139 # @private init()
140 # ###
141 init = =>
142
143   args =
144     logger: @log
145     logconf: logconf
146   # > Fetch the `http-port` argument
147   args[ 'http-port' ] = parseInt argv.w || conf.getHttpPort()
148   args[ 'db-port' ] = parseInt argv.d || conf.getDbPort()
149   args[ 'db-select' ] = parseInt argv.s || conf.fetchProp 'db-select'
150
151   #FIXME this has to come from user input for security reasons:
152   args[ 'keygen' ] = conf.getKeygenPassphrase()
153   args[ 'webhooks' ] = conf.fetchProp 'webhooks'
154
155   encryption args
156
157   @log.info 'RS | Initialzing DB'
158   db args
159   # > We only proceed with the initialization if the DB is ready
160   db.isConnected ( err ) =>
161     db.selectDatabase parseInt( args[ 'db-select' ] ) || 0
162     if err
163       @log.error 'RS | No DB connection, shutting down system!'
164       shutDown()
165
166     else
167       # > Initialize all required modules with the args object.
168       @log.info 'RS | Initialzing engine'
169       engine args
170
171       # Start the event poller. The module manager will emit events for it
172       @log.info 'RS | Forking a child process for the event poller'
173       # Grab all required log config fields
174
175       cliArgs = [
176         # - the log mode: [development|productive], in development expensive origin
177         # lookups are made and added to the log entries
178         args.logconf[ 'mode' ]
179         # - the I/O log level, refer to logging.coffee for the different levels
180         args.logconf[ 'io-level' ]
181         # - the file log level, refer to logging.coffee for the different levels
182         args.logconf[ 'file-level' ]
183         # - the optional path to the log file
184         args.logconf[ 'file-path' ]
185         # - whether a log file shall be written at all [true|false]
186         args.logconf[ 'nolog' ]
187         # - The selected database
188         args[ 'db-select' ]
189         # - The keygen phrase, this has to be handled differently in the future!
190         args[ 'keygen' ]
191       ]
```

```
192         # Initialize the event poller with the required CLI arguments
193         poller = cp.fork path.resolve( __dirname, nameEP ), cliArgs
194
195         # after the engine and the event poller have been initialized we can
196         # initialize the module manager and register event listener functions
197         # from engine and event poller
198         @log.info 'RS | Initialzing module manager'
199         cm args
200         cm.addRuleListener engine.internalEvent
201         cm.addRuleListener ( evt ) -> poller.send evt
202
203         @log.info 'RS | Initialzing http listener'
204         # The request handler passes certain requests to the components manager
205         args[ 'request-service' ] = cm.processRequest
206         # We give the HTTP listener the ability to shutdown the whole system
207         args[ 'shutdown-function' ] = shutDown
208         http args
209
210 # ###
211 # Shuts down the server.
212
213 # @private shutDown()
214 # ###
215 shutDown = () =>
216   @log.warn 'RS | Received shut down command!'
217   db?.shutDown()
218   engine.shutDown()
219   # We need to call process.exit() since the express server in the http-listener
220   # can't be stopped gracefully. Why would you stop this system anyways!??
221   process.exit()
222
223 # ###
224 # ## Process Commands
225
226 # When the server is run as a child process, this function handles messages
227 # from the parent process (e.g. the testing suite)
228 # ###
229 process.on 'message', ( cmd ) -> procCmds[cmd]?()
230
231 process.on 'SIGINT', shutDown
232 process.on 'SIGTERM', shutDown
233
234 # The die command redirects to the shutDown function.
235 procCmds.die = shutDown
236
237 # *Start initialization*
238 init()
```

B.2 Rule Engine

```coffee
# ###
# Engine
# ==================
# > The heart of the WebAPI ECA System. The engine loads action invoker modules
# > corresponding to active rules actions and invokes them if an appropriate event
# > is retrieved.
# ###

# **Loads Modules:**

# - [Persistence](persistence.html)
db = require './persistence'
# - [Dynamic Modules](dynamic-modules.html)
dynmod = require './dynamic-modules'

# - External Modules:
#   [js-select](https://github.com/harthur/js-select)
jsonQuery = require 'js-select'

listUserRules = {}
isRunning = false

# ###
# Module call
# -----------
# Initializes the Engine and starts polling the event queue for new events.

# @param {Object} args
# ###
exports = module.exports = ( args ) =>
  if not isRunning
    @log = args.logger
    dynmod args
    setTimeout exports.startEngine, 10 # Very important, this forks a token for the poll task
    module.exports


# ###
# This is a helper function for the unit tests so we can verify that action
# modules are loaded correctly

# @public getListUserRules ()
# ###
exports.getListUserRules = () ->
  listUserRules

# We need this so we can shut it down after the module unit tests
exports.startEngine = () ->
  if not isRunning
    isRunning = true
    pollQueue()

# ###
# An event associated to rules happened and is captured here. Such events
# are basically CRUD on rules.

# @public internalEvent ( *evt* )
# @param {Object} evt
# ###
exports.internalEvent = ( evt ) =>
  if not listUserRules[evt.user] and evt.intevent isnt 'del'
    listUserRules[evt.user] = {}

  oUser = listUserRules[evt.user]
  oRule = evt.rule
  if evt.intevent is 'new' or ( evt.intevent is 'init' and not oUser[oRule.id] )
    oUser[oRule.id] =
      rule: oRule
      actions: {}
```

```
70      updateActionModules oRule.id
71
72    if evt.intevent is 'del' and oUser
73      delete oUser[evt.ruleId]
74
75    # If a user is empty after all the updates above, we remove her from the list
76    if JSON.stringify( oUser ) is "{}"
77      delete listUserRules[evt.user]
78
79  # ###
80  # As soon as changes were made to the rule set we need to ensure that the aprropriate action
81  # invoker modules are loaded, updated or deleted.
82
83  # @private updateActionModules ( *updatedRuleId* )
84  # @param {Object} updatedRuleId
85  # ###
86  updateActionModules = ( updatedRuleId ) =>
87
88    # Remove all action invoker modules that are not required anymore
89    fRemoveNotRequired = ( oUser ) ->
90
91      # Check whether the action is still existing in the rule
92      fRequired = ( actionName ) ->
93        for action in oUser[updatedRuleId].rule.actions
94          # Since the event is in the format 'module -> function' we need to split the string
95          if (action.split ' -> ')[ 0 ] is actionName
96            return true
97        false
98
99      # Go thorugh all loaded action modules and check whether the action is still required
100     if oUser[updatedRuleId]
101       for action of oUser[updatedRuleId].rule.actions
102         delete oUser[updatedRuleId].actions[action] if not fRequired action
103
104   fRemoveNotRequired oUser for name, oUser of listUserRules
105
106   # Add action invoker modules that are not yet loaded
107   fAddRequired = ( userName, oUser ) =>
108
109     # Check whether the action is existing in a rule and load if not
110     fCheckRules = ( oMyRule ) =>
111
112       # Load the action invoker module if it was part of the updated rule or if it's new
113       fAddIfNewOrNotExisting = ( actionName ) =>
114         moduleName = (actionName.split ' -> ')[ 0 ]
115         if not oMyRule.actions[moduleName] or oMyRule.rule.id is updatedRuleId
116           db.actionInvokers.getModule userName, moduleName, ( err, obj ) =>
117             if obj
118               # we compile the module and pass:
119               dynmod.compileString obj.data,  # code
120                 userName,                          # userId
121                 oMyRule.rule,                     # oRule
122                 moduleName,                        # moduleId
123                 obj.lang,                          # script language
124                 "actioninvoker",                   # module type
125                 db.actionInvokers,                 # the DB interface
126                 ( result ) =>
127                   if result.answ.code is 200
128                     @log.info "EN | Module '#{ moduleName }' successfully loaded for userName
129                       '#{ userName }' in rule '#{ oMyRule.rule.id }'"
130                   else
131                     @log.error "EN | Compilation of code failed! #{ userName },
132                       #{ oMyRule.rule.id }, #{ moduleName }: #{ result.answ.message }"
133                   oMyRule.actions[moduleName] = result
134           else
135             @log.warn "EN | #{ moduleName } not found for #{ oMyRule.rule.id }!"
136
137       fAddIfNewOrNotExisting action for action in oMyRule.rule.actions
138
139     # Go thorugh all rules and check whether the action is still required
140     fCheckRules oRl for nmRl, oRl of oUser
141
```

```
142   # load all required modules for all users
143   fAddRequired userName, oUser for userName, oUser of listUserRules
144
145 numExecutingFunctions = 1
146 pollQueue = () ->
147   if isRunning
148     db.popEvent ( err, obj ) ->
149       if not err and obj
150         processEvent obj
151     setTimeout pollQueue, 20 * numExecutingFunctions
152
153 oOperators =
154   '<': ( x, y ) -> x < y
155   '<=': ( x, y ) -> x <= y
156   '>': ( x, y ) -> x > y
157   '>=': ( x, y ) -> x >= y
158   '==': ( x, y ) -> x is y
159   '!=': ( x, y ) -> x isnt y
160   'instr': ( x, y ) -> x.indexOf( y ) > -1
161
162 # ###
163 # Checks whether all conditions of the rule are met by the event.
164
165 # @private validConditions ( *evt, rule* )
166 # @param {Object} evt
167 # @param {Object} rule
168 # ###
169 validConditions = ( evt, rule, userId, ruleId ) ->
170   if rule.conditions.length is 0
171     return true
172   for cond in rule.conditions
173     selectedProperty = jsonQuery( evt, cond.selector ).nodes()
174     if selectedProperty.length is 0
175       db.appendLog userId, ruleId, 'Condition', "Node not found in event: #{ cond.selector }"
176       return false
177
178     op = oOperators[ cond.operator ]
179     if not op
180       db.appendLog userId, ruleId, 'Condition', "Unknown operator: #{ cond.operator }.
181         Use one of #{ Object.keys( oOperators ).join ', ' }"
182       return false
183
184     try
185       # maybe we should only allow certain ops for certain types
186       if cond.type is 'string'
187         val = selectedProperty[ 0 ]
188       else if cond.type is 'bool'
189         val = selectedProperty[ 0 ]
190       else if cond.type is 'value'
191         val = parseFloat( selectedProperty[ 0 ] ) || 0
192
193       if not op val, cond.compare
194         return false
195     catch err
196       db.appendLog userId, ruleId, 'Condition', "Error: Selector '#{ cond.selector }',
197         Operator #{ cond.operator }, Compare: #{ cond.compare }"
198
199   return true
200
201 # ###
202 # Handles retrieved events.
203
204 # @private processEvent ( *evt* )
205 # @param {Object} evt
206 # ###
207 processEvent = ( evt ) =>
208   fSearchAndInvokeAction = ( node, arrPath, funcName, evt, depth ) =>
209     if not node
210       @log.error "EN | Didn't find property in rule list: " + arrPath.join( ', ' ) + " at depth " + depth
211       return
212     if depth is arrPath.length
213       try
```

```
214           numExecutingFunctions++
215           @log.info "EN | #{ funcName } executes..."
216           arrArgs = []
217           if node.funcArgs[ funcName ]
218             for oArg in node.funcArgs[ funcName ]
219               arrSelectors = oArg.value.match /#\{(.*?)\}/g
220               argument = oArg.value
221               if arrSelectors
222                 for sel in arrSelectors
223                   selector = sel.substring 2, sel.length - 1
224                   data = jsonQuery( evt.body, selector ).nodes()[ 0 ]
225                   argument = argument.replace sel, data
226                   if oArg.value is sel
227                     argument = data # if the user wants to pass an object, we allow him to do so
228               arrArgs.push argument # Add arguments to the array
229           else
230             @log.warn "EN | Weird! arguments not loaded for function '#{ funcName }'!"
231
232           # Dispatching the action:
233           node.module[ funcName ].apply this, arrArgs
234
235           @log.info "EN | #{ funcName } finished execution"
236         catch err
237           @log.info "EN | ERROR IN ACTION INVOKER: " + err.message
238           node.logger err.message
239         if numExecutingFunctions-- % 100 is 0
240           @log.warn "EN | The system is producing too many tokens! Currently: #{ numExecutingFunctions }"
241       else
242         fSearchAndInvokeAction node[arrPath[depth]], arrPath, funcName, evt, depth + 1
243
244     @log.info 'EN | Processing event: ' + evt.eventname
245     fCheckEventForUser = ( userName, oUser ) =>
246       for ruleName, oMyRule of oUser
247
248         ruleEvent = oMyRule.rule.eventname
249         if oMyRule.rule.timestamp
250           ruleEvent += '_created:' + oMyRule.rule.timestamp
251         if evt.eventname is ruleEvent and validConditions evt, oMyRule.rule, userName, ruleName
252
253           @log.info 'EN | EVENT FIRED: ' + evt.eventname + ' for rule ' + ruleName
254
255           for action in oMyRule.rule.actions
256             arr = action.split ' -> '
257             fSearchAndInvokeAction listUserRules, [ userName, ruleName, 'actions', arr[0]], arr[1], evt, 0
258
259     # If the event is bound to a user, we only process it for him
260     if evt.username
261       fCheckEventForUser evt.username, listUserRules[ evt.username ]
262
263     # Else we loop through all users
264     else
265       fCheckEventForUser userName, oUser for userName, oUser of listUserRules
266
267 exports.shutDown = () ->
268     isRunning = false
269     listUserRules = {}
```

B.3 User Request Handler

```coffee
# ###

# Components Manager
# ==================
# > The components manager (User Request Handler) is the interface for CRUD on ECA components.
# > It also takes care of the dynamic JS modules and the rules.
# > Event Poller and Action Invoker modules are loaded as strings and stored in the database,
# > then compiled into node modules and rules and used in the engine and event poller.

# ###

# **Loads Modules:**

# - [Persistence](persistence.html)
db = require './persistence'
# - [Dynamic Modules](dynamic-modules.html)
dynmod = require './dynamic-modules'
# - [Encryption](encryption.html)
encryption = require './encryption'
# - [Request Handler](request-handler.html)
rh = require './request-handler'

# - Node.js Modules: [fs](http://nodejs.org/api/fs.html),
#   [path](http://nodejs.org/api/path.html) and
#   [events](http://nodejs.org/api/events.html)
fs = require 'fs'
path = require 'path'
events = require 'events'
eventEmitter = new events.EventEmitter()

# ###
# Module call
# -----------
# Initializes the Components Manager and constructs a new Event Emitter.

# @param {Object} args
# ###
exports = module.exports = ( args ) =>
  @log = args.logger
  db args
  dynmod args
  module.exports


# ###
# Add an event handler (eh) that listens for rules.

# @public addRuleListener ( *eh* )
# @param {function} eh
# ###

exports.addRuleListener = ( eh ) =>
  eventEmitter.addListener 'rule', eh

  # Fetch all active rules per user
  db.getAllActivatedRuleIdsPerUser ( err, objUsers ) =>

    # Go through all rules of each user
    fGoThroughUsers = ( user, rules ) =>

      # Fetch the rules object for each rule in each user
      fFetchRule = ( rule ) =>
          db.getRule user, rule, ( err, strRule ) =>
            try
              oRule = JSON.parse strRule
              db.resetLog user, oRule.id
              eventInfo = ''
              if oRule.eventstart
                eventInfo = "Starting at #{ new Date( oRule.eventstart ) },
```

```
70                    Interval set to #{ oRule.eventinterval } minutes"
71                db.appendLog user, oRule.id, "INIT", "Rule '#{ oRule.id }' initialized. #{ eventInfo }"
72
73                eventEmitter.emit 'rule',
74                  intevent: 'init'
75                  user: user
76                  rule: oRule
77              catch err
78                @log.warn "CM | There's an invalid rule in the system: #{ strRule }"
79
80        # Go through all rules for each user
81        fFetchRule rule for rule in rules
82
83      # Go through each user
84      fGoThroughUsers user, rules for user, rules of objUsers
85
86  # ###
87  # Processes a user request coming through the request-handler.
88
89  # - `user` is the user object as it comes from the DB.
90  # - `oReq` is the request object that contains:
91
92  #   - `command` as a string
93  #   - `body` an optional stringified JSON object
94  # The callback function `callback( obj )` will receive an object
95  # containing the HTTP response code and a corresponding message.
96
97  # @public processRequest ( *user, oReq, callback* )
98  # ###
99  exports.processRequest = ( user, oReq, callback ) ->
100   if not oReq.body
101     oReq.body = '{}'
102   try
103     dat = JSON.parse oReq.body
104   catch err
105     return callback
106       code: 404
107       message: 'You had a strange body in your request!'
108   if commandFunctions[oReq.command]
109
110     # If the command function was registered we invoke it
111     commandFunctions[oReq.command] user, dat, callback
112   else
113     callback
114       code: 404
115       message: 'What do you want from me?'
116
117 # ###
118 # Checks whether all required parameters are present in the body.
119
120 # @private hasRequiredParams ( *arrParams, oBody* )
121 # @param {Array} arrParams
122 # @param {Object} oBody
123 # ###
124 hasRequiredParams = ( arrParams, oBody ) ->
125   answ =
126     code: 400
127     message: "Your request didn't contain all necessary fields! Requires: #{ arrParams.join() }"
128   return answ for param in arrParams when not oBody[param]
129   answ.code = 200
130   answ.message = 'All required properties found'
131   answ
132
133 # ###
134 # Fetches all available modules and return them together with the available functions.
135
136 # @private getModules ( *user, oBody, dbMod, callback* )
137 # @param {Object} user
138 # @param {Object} oBody
139 # @param {Object} dbMod
140 # @param {function} callback
141 # ###
```

```
142 getModules = ( user, oBody, dbMod, callback ) ->
143   fProcessIds = ( userName ) ->
144     ( err, arrNames ) ->
145       oRes = {}
146       answReq = () ->
147         callback
148           code: 200
149           message: JSON.stringify oRes
150       sem = arrNames.length
151       if sem is 0
152         answReq()
153       else
154         fGetFunctions = ( id ) =>
155           dbMod.getModule userName, id, ( err, oModule ) =>
156             if oModule
157               oRes[id] = JSON.parse oModule.functions
158             if --sem is 0
159               answReq()
160         fGetFunctions id for id in arrNames
161
162   dbMod.getAvailableModuleIds user.username, fProcessIds user.username
163
164 getModuleParams = ( user, oBody, dbMod, callback ) ->
165   answ = hasRequiredParams [ 'id' ], oBody
166   if answ.code isnt 200
167     callback answ
168   else
169     dbMod.getModuleField user.username, oBody.id, "params", ( err, oBody ) ->
170       answ.message = oBody
171       callback answ
172
173 getModuleUserParams = ( user, oBody, dbMod, callback ) ->
174   answ = hasRequiredParams [ 'id' ], oBody
175   if answ.code isnt 200
176     callback answ
177   else
178     dbMod.getUserParams oBody.id, user.username, ( err, str ) ->
179       oParams = JSON.parse str
180       for name, oParam of oParams
181         if not oParam.shielded
182           oParam.value = encryption.decrypt oParam.value
183       answ.message = JSON.stringify oParams
184       callback answ
185
186 getModuleUserArguments = ( user, oBody, dbMod, callback ) ->
187   answ = hasRequiredParams [ 'ruleId' ,'moduleId' ], oBody
188   if answ.code isnt 200
189     callback answ
190   else
191     dbMod.getAllModuleUserArguments user.username, oBody.ruleId, oBody.moduleId, ( err, oBody ) ->
192       answ.message = oBody
193       callback answ
194
195 forgeModule = ( user, oBody, modType, dbMod, callback ) =>
196   answ = hasRequiredParams [ 'id', 'params', 'lang', 'data' ], oBody
197   if answ.code isnt 200
198     callback answ
199   else
200     if oBody.overwrite
201       storeModule user, oBody, modType, dbMod, callback
202     else
203       dbMod.getModule user.username, oBody.id, ( err, mod ) =>
204         if mod
205           answ.code = 409
206           answ.message = 'Module name already existing: ' + oBody.id
207           callback answ
208         else
209           storeModule user, oBody, modType, dbMod, callback
210
211 storeModule = ( user, oBody, modType, dbMod, callback ) =>
212   src = oBody.data
213   dynmod.compileString src, user.username, id: 'dummyRule' , oBody.id, oBody.lang, modType, null, ( cm ) =>
```

```
214       answ = cm.answ
215       if answ.code is 200
216         funcs = []
217         funcs.push name for name, id of cm.module
218         @log.info "CM | Storing new module with functions #{ funcs.join( ', ' ) }"
219         answ.message =
220           " Module #{ oBody.id } successfully stored! Found following function(s): #{ funcs }"
221         oBody.functions = JSON.stringify funcs
222         oBody.functionArgs = JSON.stringify cm.funcParams
223         dbMod.storeModule user.username, oBody
224         # if oBody.public is 'true'
225         #   dbMod.publish oBody.id
226       callback answ
227
228   # Store a rule and inform everybody about it
229   # ------------------------------------------
230   storeRule = ( user, oBody, callback ) =>
231     # This is how a rule is stored in the database
232       rule =
233         id: oBody.id
234         eventtype: oBody.eventtype
235         eventname: oBody.eventname
236         eventstart: oBody.eventstart
237         eventinterval: oBody.eventinterval
238         conditions: oBody.conditions
239         actions: oBody.actions
240       if oBody.eventstart
241         rule.timestamp = (new Date()).toISOString()
242       strRule = JSON.stringify rule
243       # store the rule
244       db.storeRule user.username, rule.id, strRule
245       # if event module parameters were sent, store them
246       if oBody.eventparams
247         epModId = rule.eventname.split( ' -> ' )[ 0 ]
248         db.eventPollers.storeUserParams epModId, user.username, JSON.stringify oBody.eventparams
249       oFuncArgs = oBody.eventfunctions
250       # if event function arguments were sent, store them
251       for id, args of oFuncArgs
252         arr = id.split ' -> '
253         db.eventPollers.storeUserArguments user.username, rule.id, arr[ 0 ], arr[ 1 ], JSON.stringify args
254
255       # if action module params were sent, store them
256       oParams = oBody.actionparams
257       for id, params of oParams
258         db.actionInvokers.storeUserParams id, user.username, JSON.stringify params
259       oFuncArgs = oBody.actionfunctions
260       # if action function arguments were sent, store them
261       for id, args of oFuncArgs
262         arr = id.split ' -> '
263         db.actionInvokers.storeUserArguments user.username, rule.id, arr[ 0 ], arr[ 1 ], JSON.stringify args
264
265       eventInfo = ''
266       if rule.eventstart
267         eventInfo = "Starting at #{ new Date( rule.eventstart ) }, Interval set to #{ rule.eventinterval } minutes"
268       # Initialize the rule log
269       db.resetLog user.username, rule.id
270       db.appendLog user.username, rule.id, "INIT", "Rule '#{ rule.id }' initialized. #{ eventInfo }"
271
272       # Inform everbody about the new rule
273       eventEmitter.emit 'rule',
274         intevent: 'new'
275         user: user.username
276         rule: rule
277       callback
278         code: 200
279         message: "Rule '#{ rule.id }' stored and activated!"
280
281
282   #
283   # COMMAND FUNCTIONS
284   # =================
285   #
```

```
286  # Those are the answers to user requests.
287
288  commandFunctions =
289    get_public_key: ( user, oBody, callback ) ->
290      callback
291        code: 200
292        message: encryption.getPublicKey()
293
294  # EVENT POLLERS
295  # -------------
296    get_event_pollers: ( user, oBody, callback ) ->
297      getModules  user, oBody, db.eventPollers, callback
298
299    get_full_event_poller: ( user, oBody, callback ) ->
300      db.eventPollers.getModule user.username, oBody.id, ( err, obj ) ->
301        callback
302          code: 200
303          message: JSON.stringify obj
304
305    get_event_poller_params: ( user, oBody, callback ) ->
306      getModuleParams user, oBody, db.eventPollers, callback
307
308    get_event_poller_user_params: ( user, oBody, callback ) ->
309      getModuleUserParams user, oBody, db.eventPollers, callback
310
311    get_event_poller_user_arguments: ( user, oBody, callback ) ->
312      getModuleUserArguments user, oBody, db.eventPollers, callback
313
314    get_event_poller_function_arguments: ( user, oBody, callback ) ->
315      answ = hasRequiredParams [ 'id' ], oBody
316      if answ.code isnt 200
317        callback answ
318      else
319        db.eventPollers.getModuleField user.username, oBody.id, 'functionArgs', ( err, obj ) ->
320          callback
321            code: 200
322            message: obj
323
324    forge_event_poller: ( user, oBody, callback ) ->
325      forgeModule user, oBody, "eventpoller", db.eventPollers, callback
326
327    delete_event_poller: ( user, oBody, callback ) ->
328      answ = hasRequiredParams [ 'id' ], oBody
329      if answ.code isnt 200
330        callback answ
331      else
332        db.eventPollers.deleteModule user.username, oBody.id
333        callback
334          code: 200
335          message: 'OK!'
336
337  # ACTION INVOKERS
338  # ---------------
339    get_action_invokers: ( user, oBody, callback ) ->
340      getModules  user, oBody, db.actionInvokers, callback
341
342    get_full_action_invoker: ( user, oBody, callback ) ->
343      answ = hasRequiredParams [ 'id' ], oBody
344      if answ.code isnt 200
345        callback answ
346      else
347        db.actionInvokers.getModule user.username, oBody.id, ( err, obj ) ->
348          callback
349            code: 200
350            message: JSON.stringify obj
351
352    get_action_invoker_params: ( user, oBody, callback ) ->
353      getModuleParams user, oBody, db.actionInvokers, callback
354
355    get_action_invoker_user_params: ( user, oBody, callback ) ->
356      getModuleUserParams user, oBody, db.actionInvokers, callback
357
```

```
358   get_action_invoker_user_arguments: ( user, oBody, callback ) ->
359     getModuleUserArguments user, oBody, db.actionInvokers, callback
360
361   get_action_invoker_function_arguments: ( user, oBody, callback ) ->
362     answ = hasRequiredParams [ 'id' ], oBody
363     if answ.code isnt 200
364       callback answ
365     else
366       db.actionInvokers.getModuleField user.username, oBody.id, 'functionArgs', ( err, obj ) ->
367         callback
368           code: 200
369           message: obj
370
371   forge_action_invoker: ( user, oBody, callback ) ->
372     forgeModule user, oBody, "actioninvoker", db.actionInvokers, callback
373
374   delete_action_invoker: ( user, oBody, callback ) ->
375     answ = hasRequiredParams [ 'id' ], oBody
376     if answ.code isnt 200
377       callback answ
378     else
379       db.actionInvokers.deleteModule user.username, oBody.id
380       callback
381         code: 200
382         message: 'OK!'
383
384 # RULES
385 # -----
386   get_rules: ( user, oBody, callback ) ->
387     db.getRuleIds user.username, ( err, obj ) ->
388       callback
389         code: 200
390         message: obj
391
392   get_rule: ( user, oBody, callback ) ->
393     answ = hasRequiredParams [ 'id' ], oBody
394     if answ.code isnt 200
395       callback answ
396     else
397       db.getRule user.username, oBody.id, ( err, obj ) ->
398         callback
399           code: 200
400           message: obj
401
402   get_rule_log: ( user, oBody, callback ) ->
403     answ = hasRequiredParams [ 'id' ], oBody
404     if answ.code isnt 200
405       callback answ
406     else
407       db.getLog user.username, oBody.id, ( err, obj ) ->
408         callback
409           code: 200
410           message: obj
411
412   # Create new rule
413   forge_rule: ( user, oBody, callback ) ->
414     answ = hasRequiredParams [ 'id', 'eventname', 'conditions', 'actions' ], oBody
415     if answ.code isnt 200
416       callback answ
417     else
418       if oBody.overwrite
419         storeRule user, oBody, callback
420       else
421         db.getRule user.username, oBody.id, ( err, mod ) =>
422           if mod
423             answ.code = 409
424             answ.message = 'Rule name already existing: ' + oBody.id
425             callback answ
426           else
427             storeRule user, oBody, callback
428
429   delete_rule: ( user, oBody, callback ) ->
```

```
430       answ = hasRequiredParams [ 'id' ], oBody
431       if answ.code isnt 200
432         callback answ
433       else
434         db.deleteRule user.username, oBody.id
435         eventEmitter.emit 'rule',
436           intevent: 'del'
437           user: user.username
438           rule: null
439           ruleId: oBody.id
440         callback
441           code: 200
442           message: 'OK!'
443
444
445   # WEBHOOKS
446   # --------
447     create_webhook: ( user, oBody, callback ) ->
448       answ = hasRequiredParams [ 'hookname' ], oBody
449       if answ.code isnt 200
450         callback answ
451       else
452         db.getAllUserWebhookNames user.username, ( err, arrHooks ) =>
453           hookExists = false
454           hookExists = true for hookid, hookname of arrHooks when hookname is oBody.hookname
455           if hookExists
456             callback
457               code: 409
458               message: 'Webhook already existing: ' + oBody.hookname
459           else
460             db.getAllWebhookIDs ( err, arrHooks ) ->
461               genHookID = ( arrHooks ) ->
462                 hookid = ''
463                 for i in [ 0..1 ]
464                   hookid += Math.random().toString( 36 ).substring 2
465                 if arrHooks and arrHooks.indexOf( hookid ) > -1
466                   hookid = genHookID arrHooks
467                 hookid
468               hookid = genHookID arrHooks
469               db.createWebhook user.username, hookid, oBody.hookname
470               rh.activateWebhook user.username, hookid, oBody.hookname
471               callback
472                 code: 200
473                 message: JSON.stringify
474                   hookid: hookid
475                   hookname: oBody.hookname
476
477     get_all_webhooks: ( user, oBody, callback ) ->
478       db.getAllUserWebhookNames user.username, ( err, data ) ->
479         if err
480           callback
481             code: 400
482             message: "We didn't like your request!"
483         else
484           data = JSON.stringify( data ) || null
485           callback
486             code: 200
487             message: data
488
489     delete_webhook: ( user, oBody, callback ) ->
490       answ = hasRequiredParams [ 'hookid' ], oBody
491       if answ.code isnt 200
492         callback answ
493       else
494         rh.deactivateWebhook oBody.hookid
495         db.deleteWebhook user.username, oBody.hookid
496         callback
497           code: 200
498           message: 'OK!'
```

B.4 Dynamic Code Loading for Event Trigger and Action Invoker

```coffeescript
# ###

# Dynamic Modules
# ===============
# > Compiles CoffeeScript modules and loads JS modules in a VM, together
# > with only a few allowed node.js modules.
# ###

# **Loads Modules:**

# - [Persistence](persistence.html)
db = require './persistence'
# - [Encryption](encryption.html)
encryption = require './encryption'

# - Node.js Modules: [vm](http://nodejs.org/api/vm.html) and
#   [events](http://nodejs.org/api/events.html)
vm = require 'vm'

# - External Modules: [coffee-script](http://coffeescript.org/),
#       [crypto-js](https://www.npmjs.org/package/crypto-js) and
#       [import-io](https://www.npmjs.org/package/import-io)
cs = require 'coffee-script'


# ###
# Module call
# -----------
# Initializes the dynamic module handler.

# @param {Object} args
# ###
exports = module.exports = ( args ) =>
  @log = args.logger
  module.exports

logFunction = ( uId, rId, mId ) ->
  ( msg ) ->
    db.appendLog uId, rId, mId, msg

regexpComments = /((\/\/.*$)|(\/\*[\s\S]*?\*\/))/mg;
getFunctionParamNames = ( fName, func, oFuncs ) ->
  fnStr = func.toString().replace regexpComments, ''
  result = fnStr.slice( fnStr.indexOf( '(' ) + 1, fnStr.indexOf( ')' ) ).match /([^\s,]+)/g
  if not result
    result = []
  oFuncs[fName] = result

# ###
# Try to run a JS module from a string, together with the
# given parameters. If it is written in CoffeeScript we
# compile it first into JS.
# ###
exports.compileString = ( src, userId, oRule, modId, lang, modType, dbMod, cb ) =>
  if lang is 'CoffeeScript'
    try
      @log.info "DM | Compiling module '#{ modId }' for user '#{ userId }'"
      src = cs.compile src
    catch err
      cb
        answ:
          code: 400
          message: 'Compilation of CoffeeScript failed at line ' +
            err.location.first_line
      return

  @log.info "DM | Trying to fetch user specific module '#{ modId }' paramters for user '#{ userId }'"
  # dbMod is only attached if the module really gets loaded
  if dbMod
```

```
70      dbMod.getUserParams modId, userId, ( err, obj ) =>
71        try
72          oParams = {}
73          for name, oParam of JSON.parse obj
74            oParams[ name ] = encryption.decrypt oParam.value
75          @log.info "DM | Loaded user defined params for #{ userId }, #{ oRule.id }, #{ modId }"
76        catch err
77          @log.warn "DM | Error during parsing of user defined params for #{ userId }, #{ oRule.id }, #{ modId }"
78          @log.warn err
79        fTryToLoadModule userId, oRule, modId, src, modType, dbMod, oParams, cb
80    else
81      fTryToLoadModule userId, oRule, modId, src, modType, dbMod, null, cb
82
83
84  fPushEvent = ( userId, oRule, modType ) ->
85    ( obj ) ->
86      timestamp = ( new Date() ).toISOString()
87      if modType is 'eventpoller'
88        db.pushEvent
89          eventname: oRule.eventname + '_created:' + oRule.timestamp
90          body: obj
91      else
92        db.pushEvent obj
93
94  fTryToLoadModule = ( userId, oRule, modId, src, modType, dbMod, params, cb ) =>
95    if not params
96      params = {}
97
98    answ =
99      code: 200
100     message: 'Successfully compiled'
101
102   @log.info "DM | Running module '#{ modId }' for user '#{ userId }'"
103   # The function used to provide logging mechanisms on a per rule basis
104   logFunc = logFunction userId, oRule.id, modId
105   # The sandbox contains the objects that are accessible to the user
106   sandbox =
107     importio: require( 'import-io' ).client
108     prettydiff: require 'prettydiff'
109     cryptoJS: require 'crypto-js'
110     deepdiff: require 'deep-diff'
111     jsselect: require 'js-select'
112     request: require 'request'
113     needle: require 'needle'
114     jsdom: require 'jsdom'
115     diff: require 'diff'
116     id: "#{ userId }.#{ oRule.id }.#{ modId }.vm"
117     params: params
118     log: logFunc
119     debug: console.log
120     exports: {}
121     setTimeout: setTimeout # This one allows probably too much
122     pushEvent: fPushEvent userId, oRule, modType
123
124   try
125     vm.runInNewContext src, sandbox, sandbox.id
126   catch err
127     answ.code = 400
128     msg = err.message
129     if not msg
130       msg = 'Try to run the script locally to track the error! Sadly we cannot provide the line number'
131     answ.message = 'Loading Module failed: ' + msg
132
133   @log.info "DM | Module '#{ modId }' ran successfully for user '#{ userId }' in rule '#{ oRule.id }'"
134   oFuncParams = {}
135   oFuncArgs = {}
136   for fName, func of sandbox.exports
137     getFunctionParamNames fName, func, oFuncParams
138
139   if dbMod
140     oFuncArgs = {}
141
```

```
142       fRegisterArguments = ( fName ) =>
143         ( err, obj ) =>
144           if obj
145             try
146               oFuncArgs[ fName ] = JSON.parse obj
147               @log.info "DM | Found user-specific arguments to #{ userId }, #{ oRule.id }, #{ modId }: #{ obj }"
148             catch err
149               @log.warn "DM | Error parsing user-specific arguments for #{ userId }, #{ oRule.id }, #{ modId }"
150               @log.warn err
151       for func of oFuncParams
152         dbMod.getUserArguments userId, oRule.id, modId, func, fRegisterArguments func
153     cb
154       answ: answ
155       module: sandbox.exports
156       funcParams: oFuncParams
157       funcArgs: oFuncArgs
158       logger: sandbox.log
```

B.5 Poller for Events, loads Event Triggers

```coffee
# ###

# Event Poller
# ===============
# > Gets notified about new rules and loads event poller (trigger) if it is part of the rule.
# > Polls the Event Triggers according to their predefined intervals and starting times.
# ###

# **Loads Modules:**

# - [Logging](logging.html), [Persistence](persistence.html),
# [Encryption](encryption.html)
# and [Dynamic Modules](dynamic-modules.html)
logger = require './logging'
db = require './persistence'
dynmod = require './dynamic-modules'
encryption = require './encryption'

# If we do not receive all required arguments we shut down immediately
if process.argv.length < 8
  console.error 'Not all arguments have been passed!'
  process.exit()

# Fetch all the command line arguments to the process to init the logger
logconf =
  mode: process.argv[ 2 ]
  nolog: process.argv[ 6 ]
logconf[ 'io-level' ] = process.argv[ 3 ]
logconf[ 'file-level' ] = process.argv[ 4 ]
logconf[ 'file-path' ] = process.argv[ 5 ]
log = logger.getLogger logconf
log.info 'EP | Event Poller starts up'

process.on 'uncaughtException', ( err ) ->
  log.error 'Probably one of the event pollers produced an error!'
  log.error err

# Initialize required modules (should be in cache already)
db logger: log
dynmod
  logger: log

db.selectDatabase parseInt( process.argv[ 7 ] ) || 0

encryption
  logger: log
  keygen: process.argv[ 8 ]

# Initialize module local variables and
listUserModules = {}
isRunning = true

# Register disconnect action. Since no standalone mode is intended
# the event poller will shut down
process.on 'disconnect', () ->
  log.warn 'EP | Shutting down Event Poller'
  isRunning = false
  # very important so the process doesnt linger on when the paren process is killed
  process.exit()

# If the process receives a message it is concerning the rules
process.on 'message', ( msg ) ->
  log.info "EP | Got info about new rule: #{ msg.event }"

  # Let's split the event string to find module and function in an array

  # A initialization notification or a new rule
  if msg.intevent is 'new' or msg.intevent is 'init'
    fLoadModule msg
```

```
70      # We fetch the module also if the rule was updated
71
72    # A rule was deleted
73    if msg.intevent is 'del'
74      delete listUserModules[msg.user][msg.ruleId]
75      if JSON.stringify( listUserModules[msg.user] ) is "{}"
76        delete listUserModules[msg.user]
77
78  # Loads a module if required
79  fLoadModule = ( msg ) ->
80    arrName = msg.rule.eventname.split ' -> '
81    fAnonymous = () ->
82      db.eventPollers.getModule msg.user, arrName[ 0 ], ( err, obj ) ->
83        if not obj
84          log.info "EP | No module retrieved for #{ arrName[0] }, must be a custom event or Webhook"
85        else
86           # we compile the module and pass:
87          dynmod.compileString obj.data,  # code
88            msg.user,                     # userId
89            msg.rule,                     # oRule
90            arrName[0],                   # moduleId
91            obj.lang,                     # script language
92            "eventpoller",                # the module type
93            db.eventPollers,              # the DB interface
94            ( result ) ->
95              if not result.answ is 200
96                log.error "EP | Compilation of code failed! #{ msg.user },
97                  #{ msg.rule.id }, #{ arrName[0] }"
98
99              # If user is not yet stored, we open a new object
100             if not listUserModules[msg.user]
101               listUserModules[msg.user] = {}
102
103             oUser = listUserModules[msg.user]
104             # We open up a new object for the rule it
105             oUser[msg.rule.id] =
106               id: msg.rule.eventname
107               timestamp: msg.rule.timestamp
108               pollfunc: arrName[1]
109               funcArgs: result.funcArgs
110               eventinterval: msg.rule.eventinterval * 60 * 1000
111               module: result.module
112               logger: result.logger
113
114             start = new Date msg.rule.eventstart
115             nd = new Date()
116             now = new Date()
117             if start < nd
118               # If the engine restarts start could be from last year even
119               nd.setMilliseconds 0
120               nd.setSeconds 0
121               nd.setMinutes start.getMinutes()
122               nd.setHours start.getHours()
123               # if it's still smaller we add one day
124               if nd < now
125                 nd.setDate nd.getDate() + 1
126             else
127               nd = start
128
129             log.info "EP | New event module '#{ arrName[0] }' loaded for user #{ msg.user },
130               in rule #{ msg.rule.id }, registered at UTC|#{ msg.rule.timestamp },
131               starting at UTC|#{ start.toISOString() } ( which is in #{ ( nd - now ) / 1000 / 60 } minutes )
132               and polling every #{ msg.rule.eventinterval } minutes"
133             setTimeout fCheckAndRun( msg.user, msg.rule.id, msg.rule.timestamp ), nd - now
134
135   if msg.intevent is 'new' or
136       not listUserModules[msg.user] or
137       not listUserModules[msg.user][msg.rule.id]
138     fAnonymous()
139
140 fCheckAndRun = ( userId, ruleId, timestamp ) ->
141   () ->
```

```
142     log.info "EP | Check and run user #{ userId }, rule #{ ruleId }"
143     if isRunning and
144         listUserModules[userId] and
145         listUserModules[userId][ruleId]
146       # If there was a rule update we only continue the latest setTimeout execution
147       if listUserModules[userId][ruleId].timestamp is timestamp
148         oRule = listUserModules[userId][ruleId]
149         fCallFunction userId, ruleId, oRule
150         setTimeout fCheckAndRun( userId, ruleId, timestamp ), oRule.eventinterval
151       else
152         log.info "EP | We found a newer polling interval and discontinue this one which
153             was created at UTC|#{ timestamp }"
154
155 # We have to register the poll function in belows anonymous function
156 # because we're fast iterating through the listUserModules and references will
157 # eventually not be what they are expected to be
158 fCallFunction = ( userId, ruleId, oRule ) ->
159   try
160     arrArgs = []
161     if oRule.funcArgs and oRule.funcArgs[oRule.pollfunc]
162       for oArg in oRule.funcArgs[oRule.pollfunc]
163         arrArgs.push oArg.value
164     oRule.module[oRule.pollfunc].apply this, arrArgs
165   catch err
166     log.info "EP | ERROR in module when polled: #{ oRule.id } #{ userId }: #{err.message}"
167     throw err
168     oRule.logger err.message
169 # ###
170 # This function will loop infinitely every 10 seconds until isRunning is set to false
171
172 # @private pollLoop()
173 # ###
174 pollLoop = () ->
175   # We only loop if we are running, this is an ugly keep-alive for legacy reasons...
176   if isRunning
177
178     # # Go through all users
179     # for userName, oRules of listUserModules
180
181     #   # Go through each of the users modules
182     #   for ruleName, myRule of oRules
183
184     #     # Call the event poller module function
185     #     fCallFunction myRule, ruleName, userName
186
187     setTimeout pollLoop, 10000
188
189 # Finally if everything initialized we start polling for new events
190 pollLoop()
```

UNIVERSITÄT BASEL

PHILOSOPHISCH-NATURWISSENSCHAFTLICHE FAKULTÄT

Declaration on Scientific Integrity

(including a Declaration on Plagiarism and Fraud)

Bachelor's / Master's Thesis *(Please cross out what does not apply)*

Title of Thesis *(Please print in capital letters)*:

_____

_____

_____

First Name, Surname *(Please print in capital letters):* _____

Matriculation No.: _____

I hereby declare that this submission is my own work and that I have fully acknowledged the assistance received in completing this work and that it contains no material that has not been formally acknowledged.

I have mentioned all source materials used and have cited these in accordance with recognised scientific rules.

In addition to this declaration, I am submitting a separate agreement regarding the publication of or public access to this work.

☐ Yes ☐ No

Place, Date: _____

Signature: _____

Please enclose a completed and signed copy of this declaration in your Bachelor's or Master's thesis .

UNI
BASEL