Towards Reactive Information Systems and their Services

Master Thesis

Author:
Dominic Bosch

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

May 15, 2014

UNI BASEL

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

The Web is an ever growing entity, in all aspects that it covers. It surrounds a rapidly growing number of human beings in their daily life and starts to flood them with functionality and information. One example of this information flood is the one hundred hours of video material that is uploaded to YouTube[33] every single minute. An increasing number of bigger computing centers, but also ever smaller devices provide more data and functionality, both in quantity and complexity. Also, a growing fraction of these devices have access to the Web, which means they are potential data and functionality resources. A white paper[6] estimates that 200 million devices were connected to the internet in the year 2000. They also estimate that this number raised up to approximately 10 billion connected devices in 2013. Further more, they expect this number to grow up to 50 billion connected devices by the year 2020. This alone shows how strong growth of potentially data and functionalty delivering devices is growing currently and in the near future. Other recent research[16][27] has shown that the number of accessible Web APIs follows power law distribution and thus provides an ever growing source of data and functionality. The ever smaller devices that make up the "Internet of Things"[31] today, are also capable of spreading the ubiquitous access to the Web, e.g. through mobile devices. Web of Things [15]

The human being requires tools to get the right information in the right situation at the right place and also to automate tasks user-centric reactions to allow a personalization of the information flood Governing the Web's information flood is getting more difficult and even fairly impossible for human beings with the tools given.

It becomes important to the individual to be able to filter out personally important bits and pieces. Basic filters are often available but they do not allow smart filtering in any way. Also, apart from smart filtering of information, people should have the possibility to aggregate important information in their desired place and in a way it's most useful for themselves. Such an aggregation implies access to services that consume data and produce an output, be it a data answer or storage. This means the user should get access to data and functionality services in a way that she can combine the possibilities in a suitable way to generated the most valuable output for her. Even if the Web service access gets simpler these days, the average user is not able to weild them. The challenge to provide users with ways to handle these already simpler accessible services, called Web API's has received a notable amount of attention over the last few years.

It is a promising research field that leads towards reactivity in the Web through programmability. Currently somebody that want's to program the Web, requires deep knowledge of the required services and their functionality. There are a few possibilities in the Web that go towards easing the programmability of the Web, but they are either complicated to weild themselves or mere data copy or mashup tasks. Our research in this thesis is about easing the programmability of the Web and therefore to achieve the reactive Web.

Chapter 2

Related Work

Research on behalf of the Web attracts a great deal of attention. This is not surprising since modern business and life is already impossible without the internet. Great opportunities arise with it and we cover a part of the Web service orchestration in it. There Service-oriented Architecture (SOA) [23] With the wide adoption of SOA, we see an increasing number of Web accessible services and their compositions. [16] By the end of 2013 this number grew to 10'000 http://www.programmableweb.com/news/programmablewebs-directory-hits-10000-apis.-and-counting./analysis/2013/09/23 At the

2.1 Services in the Web

What are services [22] An important development within the Web are the increasing number of available services. The term service in the Web is not very precise and the Web's short history has already seen a lot of different kinds.

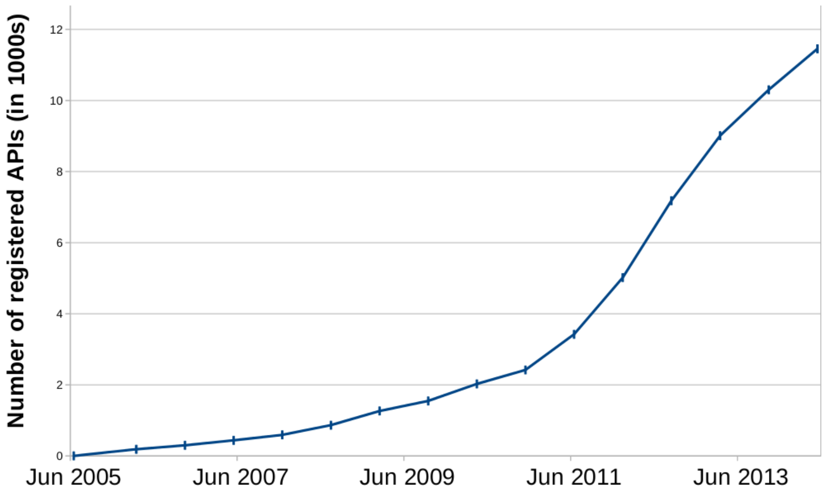


Figure 2.1: Number of registered APIs in the ProgrammableWeb directory by date

A fairly early adoption of the service concept to computers were the Remote Procedure Calls (RPC) [3]. Through RPC a piece of code can be executed on a different machine, than the one which is calling the procedure. It is basically achieved via inter-process communication and doesn't necessarily require the Web. Even more since when RPC was invented, the World-Wide Web [2] didn't even exist. RPC also found its use in grid computing [29] and through this, opened doors into the field of distributed computation. The RPC paradigm isn't bound to certain technologies and thus, has been implemented in a lot of differrent programming languages. These implementations were tightly bound to the respective language that was used, which resulted largely in incomptaiblity among them. It became necessary to enhance RPC's in order to get cross platform compatibility. By abstracting RPC functionality with the Extensible Markup Language (XML) [7], compatibility between services that used different technologies was easier to achieve.

Since XML-RPC was held relatively simple but received a lot of attention, it was further enhanced. Together with additional functionality, XML-RPC turned into Simple Object Access Protocol (SOAP) [5]. SOAP is accompanied by the Web Service Description Language (WSDL) [9] which is used to describe the interfaces to SOAP services. Through SOAP and WSDL a client for 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 incorporated into the existing application as if it is a local function call. The SOAP layer takes care of marshalling the request and unmarshalling the response.

Another initiative that aimed for eased communication between different platform is the Common Object Request Broker Architecture (CORBA) [10]. As the name already suggest it is an object-oriented approach and it allows the exchange of whole objects. CORBA relies on its communication layer, the Object Request Broker (ORB), which forms the basis of its architecture. The platform-specific ORB's provide the communication abstraction, which free the application from platform dependencies. Similar to SOAP's WSDL, CORBA has its Interface Definition Languange (IDL) to provide information about the objects to be offered and accessed. 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. CORBA requires the implementation of object-oriented mechanisms in programming languages which aren't object-oriented. This can be technically difficult and become an eventually tedious task. CORBA allows communication between applications written in different programming languages and which are running on the same physical computer, as well as the communication between different computers in the same network. 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, thought they are shielded by the ORB.

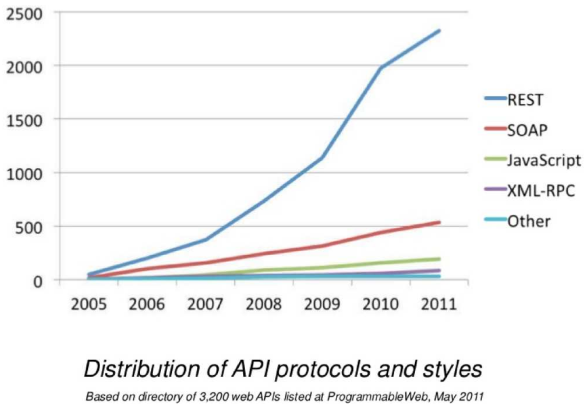
Naturally, as the Web grows quickly in its offered services and also efforts to access them, trends lead toward simple ways to access services in the Web. During our research it turned out that SOAP's overhead compared to REST diminishes its popularity on behalf of the latter one. But still since SOAP services are request responders they have their place in our model, both as event trigger and action invoker.

Several different resources [Figure 2.2b][Figure 2.2a (John Musser, ProgrammableWeb, 2011)] draw the same picture, an increasing popularity of REST over SOAP by an order of magnitude and a exponential growth of the number of accessible Web API's.

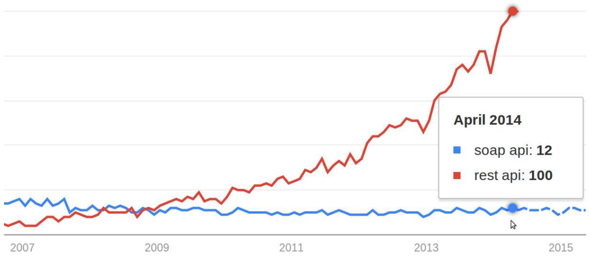
An interesting trend that could be observed was the growing popularity of REST over SOAP. [24]

Web APIs are Web accessible endpoints for users to invoke.

Twelve Theses on Reactive Rules for the Web [8]: This article investigates issues of relevance in designing high-level programming languages dedicated to reactivity on the Web. It presents twelve theses on features desirable for a language of reactive rules tuned to programming Web and Semantic Web applications.



(a) REST vs. SOAP: Simplicity wins again (John Musser, ProgrammableWeb, 2011)



(b) Google Trends: "soap api" vs. "rest api"

Figure 2.2: REST vs. SOAP Comparison

Data and functionalities in the Web were always accessible via Web services, whatever this means. reengineering of services in the beginning with standardised REST API's we got to Web API, easy access and understandable

The fast evolving Web has brought up a trend towards easy to master interfaces to services, the so called Web APIs. They do not only provide access to mere services but whole applications that allow access over Web APIs. These trending Web APIs benefit from a RESTful architecture which predominantly uses HTTP and thus relies on the most basic and powerful operations and the basis of the Web itself, the HTTP protocol.

Mashups combine information and functionality of more than one Web service in a single place. The mashing up of such Web services allows data to be enhanced with new informations, processing / refinement of the information, or even ways to interact with them, e.g. through Google Maps. Simple functions from different sources can be combined into more powerful ones, which influence data and services in a way their founders eventually didn't even think of. Web service mashups have been developped ever since services in the Web started to exist and were accessible in a more or less convenient way. We introduce Paul Rademacher as an example for how recent the invention of Web service mashups are. He's one of the first inventors of such a Web service mashup. In the same year after Google Maps came up in 2005, he invented a site [25, 11] that displayed Craigslist houses on a Google Map. With no Google Maps API at that time, he needed time and skills to reverse engineer Google Map's functionalities.

A large number of such "static" mashups were and are still developped. They are static in the way that they aggregate a fixed (and mostly low) number, of either data or functionality resources, to provide an enhanced resource in a specialized domain. Of course Mashups can be mashed up again, to provide even more sophisticated functionality and data. Some latest example Mashups, taken from the ProgrammableWeb [26] collection, are:

- Wifi and Plugs [13]: MapBox, Google Docs and Import.io API's used to display where Wi-Fi and plugs are available in London.
- MapLight [17]: GovTrack.us and OpenSecrets API's used to combine political results with financial contributions to show how capital contributions affect voting.
- Shared Count [30]: Facebook, LinkedIn, Pinterest and Twitter API's used to display informations about how well spread a URL is on social media sites.

In the past few years, research and development for platforms to allow users to flexibly mashup Web APIs got attention. With IFTTT and Zapier, two platforms have evolved out of this process. Users that register on those platforms are provided with a multitude of Web API functions that act as event triggers and such that are used to execute actions. The user is then free to combine these event triggers and actions in the way it suits best, creating helpful Web API mashups on their own.

2.2 Rule Engines & Rule Languages

It turns out that Web API mashing up is not able to bring reactivity to the Web. They are merely aggregations of services that only provide data or functions but no write possibilities such as Web applications provide.

Thus

Several different rule languages have been developped for different purposes over the last years and they vary grately in their purpose. A compilation of research on different emerging rule-based languages and technologies [19] gives an oveview over such efforts. We examined different existing rule languages with respect to a certain use case to identify its applicability for reactivity in the Web. The use case is defined such that the rule needs to suffice the ECA paradigm:

- Event: Receipt of an Email
- Condition: Check for a certain sender
- Action: Store it remotely via a Web API

We defined an email event which the rule languages need to be able to process. The JSON representation of the given email event as depicted in the appendix.

An early ECA Rule Language for XML repositories [18] was postulated in 2003 and was picked up by many researches afterwards. It was designed to react on insert and delete events within XML repositories and as an action change XML documents.

Now apart from implementing a rules engine, we would also need to add an XML document event manager which interpretes and pushes events into the XML file *inbound_queue.xml*. Then again this instance would interprete the ouptuts of the ECA engine, which would theoretically manifest in other XML documents, and produce meaningful actions on remote hosts. This wouldn't be an architecture which has its focus on the solution of our use case and, as a result, add complexity and create an unnecessary overhead.

To make the lengthy RDF definitions smaller and more readable, Notation 3 [1] was designed and announced in 2005. Through the implies operator(=>) an "event" can be connected to an "action", both expressed in RDF's subject, predicate, object notation, which makes the expression of ECA rules a complicated and not very intuitive task. A solution to our use case would look as follows:

This language is used to express relations between entities and thus not really suitable for our use case, since we would require another interpreter to infer the actions. But concepts and ideas of the work that was done in these consortias could eventually still find influence into our solution.

The rule language XChange [21] was the outcome of the REWERSE ([28], Reasoning on the Web with Rules and Semantics) project, which was funded by the EU and Switzerland. Their work influenced a number of future research. The language was designed to add reactive behaviour to a "static" Web which is represented through XML resources. Thus we have action logics to alter such resources through insertions and deletions. Since we aim to utilize Web API's for our rule language we need a more generic approach which adds flexibility in term of the API provided. But the thorough research done with the language XChange holds valuable concepts, especially in terms of temporal evet composition. This could be a rule according to our use case:

But XChange is designed to access other resources in an action and thus provides powerful tools:

In 2008 *JSON Rules* [14] was introduced as a language to easily react on specific DOM tree compositions. The usage of JavaScript allowed them to provide simple functions which could be called directly by the actions, thus abstracting functionality from the language. This key concept found influence into our language as it allows different layers of abstractions. Through this it is possible to provide generic functions for expert user as well as very limited functions with only few possibilities for parameterization to be used by unexperienced persons. A drawback of this language is its binding to DOM tree events, where we would want to react on any events happening in the world. Also the temporal composition to complex events is not a subject of their work and needs further attention.

A recent open-source development is the Kinetic Rules Engine together with the Kinetics Rule Language [32]. It is built for the purpose of adding reactivity to the cloud. The language is based on declarative syntax, enriched with imparative elements. But it is a tedious task to get into a whole new language and their caveats. *authorization?*

The basis of *RuleML* [4] is datalog, a language in the intersection of SQL and Prolog. In 2012 the *Reaction RuleML* [20] language incorporated several different types of rules into the RuleML syntax, to establish a uniform syntax and interchangability of rules. *Reaction RuleML* is a valuable resource in terms of manifold research that has been done in the domain of rule languages, but the syntax is not user-friendly.

R2ML allows usage for RuleML together with many other dialects. Really!?

Most of the examined rule languages are designed for the interchangability of rules between different service providers. We do not attempt to jump into this domain but we rather pick up important concepts to manifest Web API's as first class citizens of our rule language. This allows the ad-hoc design and implementation of reactive rules between existing Web API's without the need for their cooperation in setting up their endpoint in a special way.

Chapter 3

Conceptual Model for Reactive Information Systems and their Services

In the previous chapter we have shown that services in the Web and reactivity through programmability have received a great deal of attention. Our goal is to combine both research fields in order to achieve reactivity within the Web by orchestrating its information space.

3.1 From Real Events to Events in the Web

Real events are always bound to a spatial location and a point on the time axis. An earthquake for example always has an epicenter and an occurence time. Different points on earth's surface would feel the earthquake, which originates from the same epicentre, at a different point in time with a different intensity. A Web event model of an earthquake would consist of a large number of identical **ground-shake** events that occur at different points in time and places. Therefore they would hold different spatial location informations and intensities. These events can be thought of as emmited into the Web by a seismometer sitting at the corresponding location.

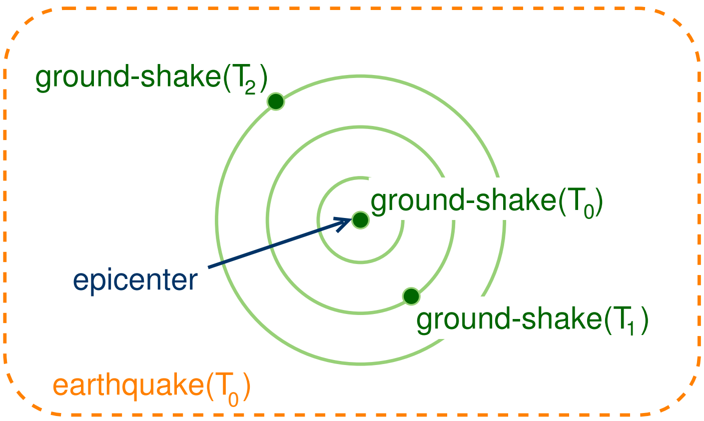


Figure 3.1: Web Event Model of an Earthquake

Within the Web, events lose their tight coupling to locations and retain only a time component. The event instances keep this information as descriptive metadata. A reactive system such as we envision it, could detect these **ground-shake** events and react on behalf of each one of them. Because of the Web's latency these events do most certainly not arrive at systems within the Web in the original order, in which they were triggered. They also do most likely not arrive in the exact same order for all systems. This leaves us with time as the only important factor left, to distinguish events from each other in the first place. To get an earthquake event out of all these ground shaking informations floating through the Web, somebody would need a reactive system that detects these events and assembles them into one earthquake event, together with a computed epicenter and magnitude. Such a system (we call it **earthquake-tracker**) would own an earthquake model that allows it to decide wether a **ground-shake** event belongs to one physical earthquake or to another one, depending on its spatial location information and the intensity at that point in time. It could then emit a more complex **earthquake** event (with epicenter and magnitude) that allows other systems to interpret this physical event and react on behalf of it.

Let's take another system that reacts on a physical earthquake. It is now left with a multitude of different options on how to react. It could only react on the **earthquake** event which is coming from the system above (**earthquake-tracker**) that applies its earthquake model to the incoming **ground-shake** events. But how long will it take for this system to deploy its **earthquake** event? Eventually it waits for one round-trip of a seismic wave around the world, which takes approximately half an hour. What if it waits two or three round-trip times in order to collect more accurate data? And what if our new system wants to react as fast as possible in order to warn people all around the world. It would then need to react on a small subset of the **ground-shake** events in order to quickly identify a real earthquake and take measurements, e.g. immediately send out text messages to people, or to deploy yet another (this time **earthquake-alert**) event into the Web's information space. This relatively simple example discloses the complex nature of event-driven systems, but also their high flexibility and fine grained tuning possibilities.

3.2 The Web's Event & Action Information Space

For a conceptual model, the information space in which the events are triggered and the actions are invoked, needs to be identified. During our research we encountered many different event or action providing subsets of the Web that can be incorporated into our model:

- World-Wide Web
- Services in the Web
- Web of Things

We have already shown in chapter "Related Work", that there are basically two different ways how the Web's information space is accessible, i.e. either functionality and data have to be requested, or data is pushed through Webhooks. All of the above listed information space subsets require at least one of these two access methodologies. And through these access methodoligies are we able to turn the information space of the Web into events and actions. The World-Wide Web [2], as envisioned by Tim Berners-Lee, is an information universe of interlinked documents, that a user can browse through. In our model, we can pull events directly from the World-Wide Web. For example, most documents in the World-Wide Web are subject to changes and such changes can be translated into events.

We gave an introduction into Services in the Web in chapter "Related Work"

Either there is an event producer which proactively pushes events into the Web, or a service is offered whose responses can be turned into events. These events and actions can have a solely virtual nature or, in the case of data from the "Web of Things", also a physical nature. A virtual nature can be anything from a static webpage to offered services, such as a detected change on a webpage can become an event as well as a service answer is interpreted as such for example a new mail arriving. A virtual action could be a babelibu. A physical nature for events could for example be measurements from a rain detector, an action could be a window shutting automatically.

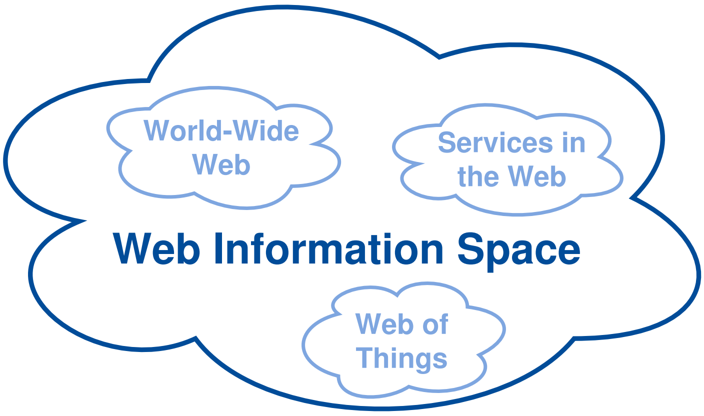


Figure 3.2: The Web's Information Space

There are different categories of events and also different ways how they can get into the Web:

We have seen in the related work chapter that there is a growing number of Web APIs which become accessible.

There are different categories in which we could

Actions

- Event Redirection
- Event Enhancement
- WebApp Actions

3.3 Reactivity through Web Information Space Orchestration

In the last section we showed how mashups create additional value for the Web by combining several WebAPI's. But it turned out, that such mashups are closed systems, which ofen only allow little degree of parametrization. To get past such limitations and define a conceptual model for reactive Web systems, it is necessary to define a

existing rule languages, rule engines,

Existing ECA systems all act on local data. Looking at (Wikipedia...) their definition is actions on local data. This does only add reactivity to these systems and not to the Web per se.

Such systems are merely event sinks which add fairly any value to the Web, except for the individual users and the system itself.

- from Web to events

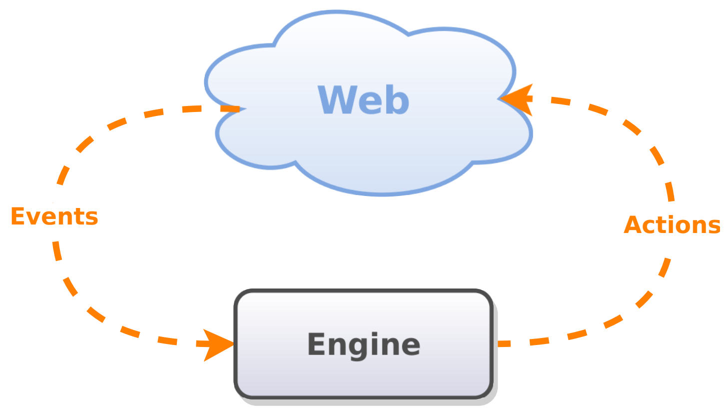


Figure 3.3: Reactor ;) Conceptual model for reactive Web systems

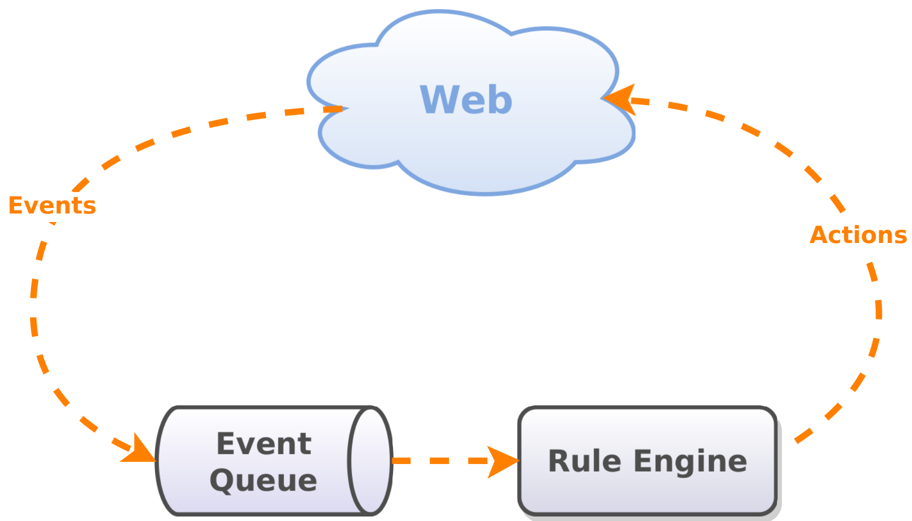


Figure 3.4: Conceptual model for reactive Web Systems

- from events to rules
- from rules to actions
- from actions to the Web
- from concept to engine

3.3.1 Conceptual Rule Language

Describe conceptual rule language ON (existing categories) IF (condition boundaries) DO (call to existing action modules with parameters)

a lot possible, but dangerous.

```
on mail
if sender="sender@mail.com"
do webapi->newcontent(subject)
```

Chapter 4

Applicability

4.1 Capturing the World-Wide Web

4.2 Capturing the Web of Things

4.3 Enhancing existing Web Applications

- Annotations
- Workflows Automation
- Availability and Functionality Testing

Chapter 5

Prototype System

The prototype system is the realisation of a reactive Web system. It was developped during the research for this thesis and acts as a platform for feasibility studies of certain use cases.

Prototype consists of a queue in which all incoming events are pushed, and an engine that picks the events from the end of the queue whenever it is idle. Since Prototype's core functionalty is the communication with resources in the Web, the architecture bases on HTTP protocol in several parts. For example the events are meant to be retrieved completely via HTTP, the user interface is a Webpage which posts requests to the system and most actions are also meant to be HTTP requests, or at least using them to gather information.

Renew figure: (Rule Engine? Reactor?)

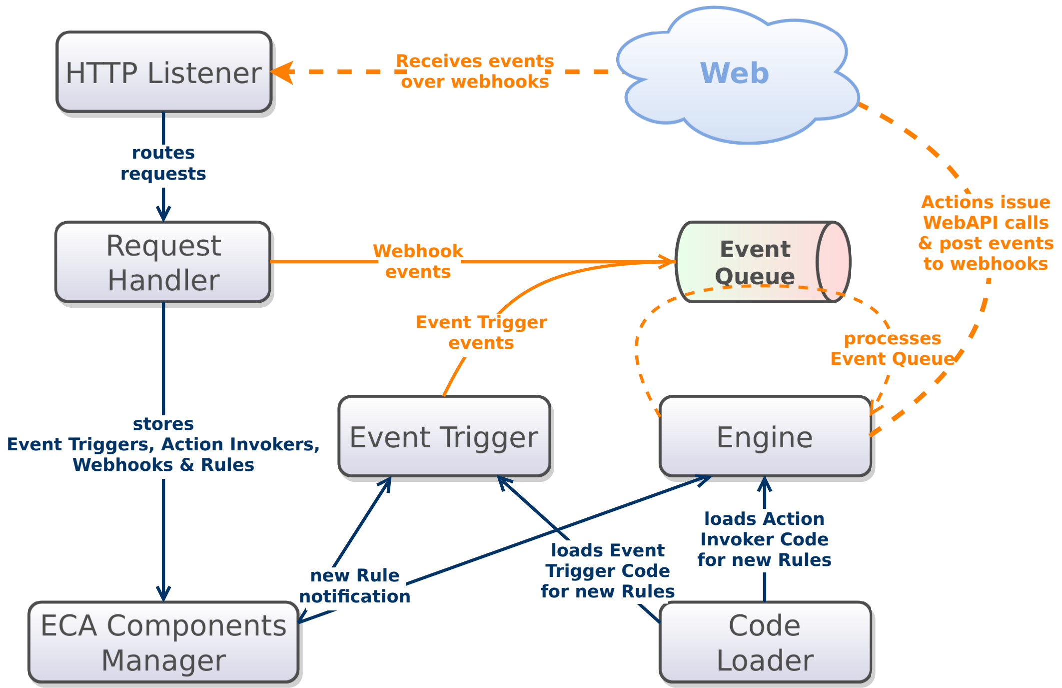


Figure 5.1: Prototype Process diagram

5.1 Event Trigger

Event Gathering is the E in ECA and without one of these letters such a system would not run. It is of utmost importance to find as much as possible ways to get data into a system.

5.1.1 Polling

5.1.2 Webhooks

5.2 Action Dispatcher

5.3 ECA Rules in the Engine

5.4 Web Programming

5.4.1 Node.js

5.4.2 Callback Functions

5.4.3 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 5.2, 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.

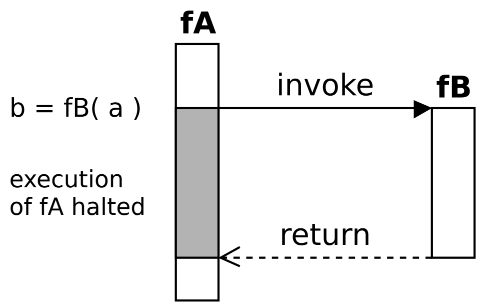


Figure 5.2: Synchronous Function Call

Asynchronous code execution, as shown in Figure 5.3, allows non-blocking and thus scalable applications. Non-blocking operations are a remedy for optimzed 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)

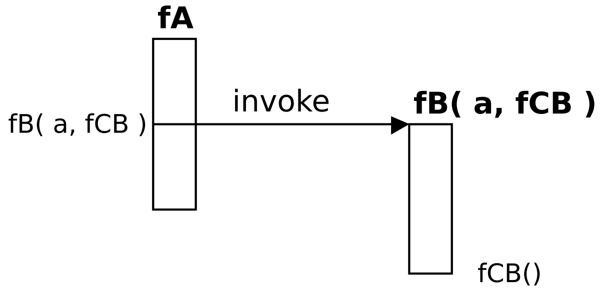


Figure 5.3: Asynchronous Function Call

exploits resources that would otherwise be bound while waiting for completion. Such operations are processed and completed whenever required resources are available.

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 5.3. 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.

Now we take closures into this asynchronous context, as defined in ECMAScript[12], which is the base for widely-spread script languages like JavaScript, JScript and ActionScript. Closures in ECMAScript[12] are defined such as they have access to the context of the function they were created in. This is shown in Figure 5.4 where `c` from `fA`'s context is accessible from within `fB`, assuming that `fB` was created in `fA` and not only invoked from there. 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 5.5. In such a setup `c` will have different values in the same part of different invocations of `fB`.

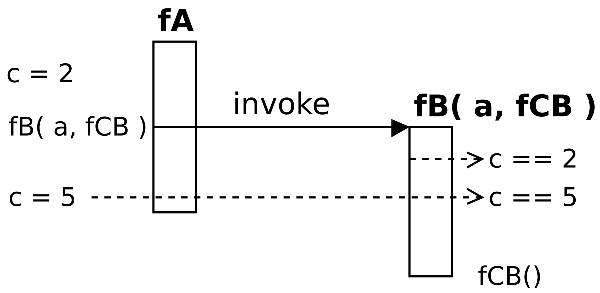


Figure 5.4: Closure Scope and referenced context

Those event-driven context overwrites can be taken care of by shielding the closure from context changes, as shown in Figure 5.6. 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 5.6) that might change but requires to be persistent for one invocation. `fC` has now `c` as a fixed context, which can't 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.

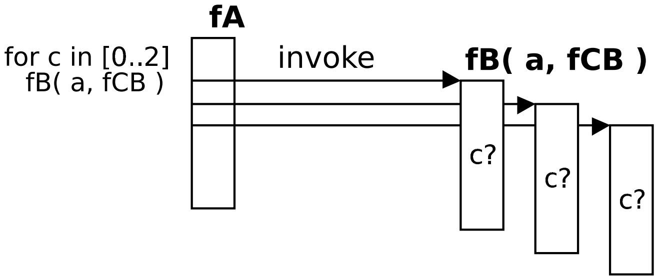


Figure 5.5: Closure context changes in a loop

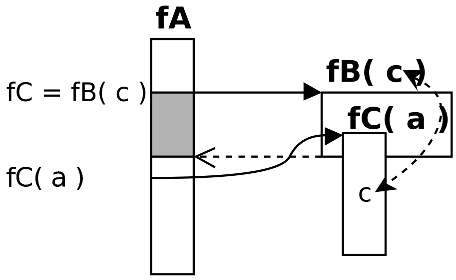


Figure 5.6: Closure context shielding

Benchmarking JavaScript vs. Java

refer to listings

Chapter 6

Discussion & Results

We have seen that the ECA approach is already a powerful one to make the Web reactive. A future improvement of this could be to adopt Complex Event Processing (CEP). This would mean that several events could be stored in a rule and be evaluated in terms of time constraints. Through this more complex events can be created as a result of several atomic events which would lead into semantically more complex events. A change in paradigm will result in an approach where events are not just processed when they are entering the system and evaluated against rules, but these events would need to be stored for quite a long time. Also the rules will not all be checked for each event but they are subject to a scheduler. It can be decided when and how often a rule is evaluated and all events will be checked at these point in times, whether they are candidates for firing the rule. A relational database will be needed in order to search through the timestamps

Bibliography

[1] Tim Berners-Lee. Notation 3 Logic. `http://www.w3.org/DesignIssues/Notation3.html`, 2005. Accessed: 2013-10-21.

[2] 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.

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

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

[5] 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.

[6] Joseph Bradley, Joel Barbier, and Doug Handler. Embracing the Internet of Everything To Capture Your Share of $14.4 Trillion. Technical report, 2013.

[7] 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.

[8] 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.

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

[10] HP DEC, NCR HyperDesk, et al., The common object request broker: architecture and specification. *Object Management Group*, 1991.

[11] Adam DuVander. 5 Years Ago Today the Web Mashup Was Born. `http://blog.programmableweb.com/2010/04/08/the-fifth-anniversary-of-map-mashups-on-the-web`, 2010. Accessed: 2014-05-01.

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

[13] Free Wifi & Plugs — Your comprehensive list of where to work around London. `http://wifiandplugs.co.uk`. Accessed: 2014-05-02.

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

[15] Dominique Guinard, Vlad Trifa, Friedemann Mattern, and Erik Wilde. From the Internet of Things to the Web of Things: Resource-oriented Architecture and Best Practices. In Dieter Uckelmann, Mark Harrison, and Florian Michahelles, editors, *Architecting the Internet of Things*, pages 97–129. Springer Berlin Heidelberg, 2011.

[16] 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.

[17] MapLight - Money and Politics — U.S. Congress Campaign Contributions and Voting Database. `http://maplight.org`. Accessed: 2014-05-02.

[18] George Papamarkos, Alexandra Poulovassilis, Ra Poulovassilis, and Peter T. Wood. Event-Condition-Action Rule Languages for the Semantic Web. In *In: Workshop on Semantic Web and Databases*, pages 309–327, 2003.

[19] Adrian Paschke and Harold Boley. Rules Capturing Events and Reactivity. In Adrian Giurca, Dragan Gasevic, and Kuldar Taveter, editors, *Handbook of Research on Emerging Rule-Based Languages and Technologies: Open Solutions and Approaches*, pages 215–252. IGI Publishing, May 2009.

[20] 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.

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

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

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

[24] Adina Ploscar, XML-RPC vs. SOAP vs. REST web services in Java–uniform using WSWrapper. *Int. J. Comput*, 4:215–223, 2012.

[25] Joshua Porter. Holy Amazing Interface, Batman! Paul Rademachers Brilliant Lodging Finder. `http://bokardo.com/archives/holy-amazing-interface-batman`, 2005. Accessed: 2014-05-01.

[26] ProgrammableWeb: APIs, mashups and code. Because the world's your programmable oyster. `http://www.programmableweb.com`. Accessed: 2014-05-02.

[27] PROGRAMMABLEWEB RESEARCH CENTER. `http://www.programmableweb.com/api-research`. Accessed: 2014-05-26.

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

[29] 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.

[30] Shared Count. `http://lab.neerajkumar.name/sharedcount`. Accessed: 2014-05-02.

[31] RolfH. Weber and Romana Weber. Introduction. In *Internet of Things*, pages 1–22. Springer Berlin Heidelberg, 2010.

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

[33] Statistics - YouTube. `http://www.youtube.com/yt/press/statistics.html`. Accessed: 2014-05-27.

Index

Appendices

Appendix A

Rule Languages

A.1 Example JSON Event for Rule Languages

```
{
  "eventname": "email",
  "body": {
    "sender": "sender@mail.com",
    "subject": "Important subject!",
    "textbody": "Hi User,\n\nThis is a lengthy mail body"
  }
}
```

A.2 E-Mail Example Rule expressed in RDF

```
ON INSERT document("inbound_queue.xml")/mails/mail
IF $delta/sender[.="sender@mail.com"]
DO DELETE document("inbound_queue.xml")/mails/mail;
  LET $api = resource("www.webapi.com") IN
  INSERT ($api, newcontent,
    <content>New mail: {$delta/subject}</content>)
```

A.3 E-Mail Example Rule expressed in Notation 3

```
{ ?x :event "email". ?x :sender "sender@mail.com" }
  => { :webapi :newcontent ?x }
```

```
TRANSACTION
  in {
    resource { "http://www.webapi.com"},
    newcontents {{
      insert newcontent { var Mail }
    }}
  }
ON
  xchange:event {{
    xchange:sender { "http://mailserver.com" },
    var Mail -> email {{
      sender { "sender@mail.com" }
    }}
  }}
END
```

```
TRANSACTION
  [...]
ON
  [...]
FROM
  in {
    resource { "http://www.weather.com"},
    temperatures {{
      var T -> temperature {{
        datetime { "2013-10-20-08:00:00AM" }
      }}
    }}
  }
END
```

A.4 E-Mail Example Rule expressed in XChange/Xcerpt

A.5 XChange/Xcerpt Remote Resource Access

A.6 E-Mail Example Rule expressed in JSON Rules

```
{
    "id": 0,
    "conditions": [
        {
            "type": "email",
            "constraints": [
                {
                    "propertyName": "sender",
                    "operator": "EQ",
                    "restriction": {
                        "type": "String",
                        "value": "sender@mail.com"
                    }
                },
                {
                    "bind": "$S",
                    "propertyName": "subject"
```

```
                }
            ]
        }
    ],
    "actions": [
        "webapi('addcontent', $S)"
    ]
}
```

A.7 E-Mail Example Rule expressed in Kinetics Rule Language (KRL)

```
rule store_mail {
  select when mail newmail
  sender re#sender@mail.com#
  subject re#*# setting(subj)
  http:post("http://www.webapi.com/newcontent")
  with params = {
    "text": subj
  }
}
```

Listing A.1: E-Mail Example rule in KRL

A.8 E-Mail Example Rule expressed in (Reaction) RuleML

```
<Rule style="active">
  <on>
    <Event>
      <Atom>
        <Rel per="value">mail</Rel>
        <Var>sender</Var>
        <Var>subject</Var>
      </Atom>
    </Event>
  </on>
  <if>
    <Atom>
      <op><Rel>equals</Rel></op>
      <Var>sender</Var>
      <Ind>sender@mail.com</Ind>
    </Atom>
  </if>
  <do>
    <Atom>
      <oid><Ind uri="http://webapi.com"/></oid>
      <Rel>newcontent</Rel>
      <Var>subject</Var>
    </Atom>
  </do>
</Rule>
```

A.9 Prototype Rule transformed into JSON


```json
{
  "event": "mail",
  "conditions": [
    { "sender": "sender@mail.com" },
  ],
  "actions": [
    {
      "api": "webapi",
      "method": "newcontent",
      "arguments": {
        "text": "$X.subject"
      }
    }
  ]
}
```

Appendix B

Rules

B.1 Binder Annotations

B.1.1 Binder Annotations

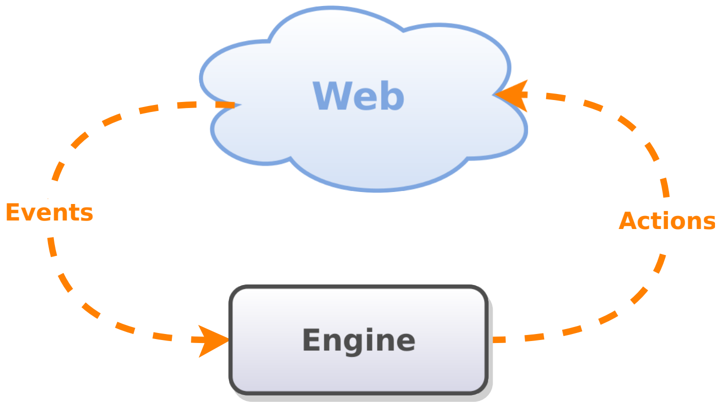


Figure B.1: SHOULD NOT SHOW UP

Appendix C

Benchmarking

C.1 Java

```java
/*
 * BenchmarkingDeferred.java
 */
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;
import java.util.HashMap;

public class BenchmarkingDeferred {

  private static Runtime runtime = Runtime.getRuntime();
  private static final ScheduledExecutorService worker =
    Executors.newSingleThreadScheduledExecutor();

  private static void deferFunctionCall( int numScopeVars, int delay,
      String scopeId ) {
    HashMap<String, String> mapVars = new HashMap<String, String>();
    for( int i = 0; i < numScopeVars; i++ ) {
      mapVars.put( "id" + i, "12345678" ); // 8 bytes per stored scope
          variable
    }
    Object context = new TimeoutContext( "TimeoutFunction" );
    Runnable task = new RunnableCallbackFunction( mapVars, context );
    worker.schedule( task, delay, TimeUnit.SECONDS );
  }

  public static void main( String[] args ) {
    long startTime, stopTime;
    int numVars = 10, firstArg = 0;
    firstArg = Integer.parseInt( args[0] );
    numVars = Integer.parseInt( args[1] );
    int j = 0, numFuncs = 1 << firstArg;

    startTime = System.nanoTime();
    while( j++ < numFuncs) {
      deferFunctionCall( numVars, numFuncs * 10, numFuncs + "(" + j + ")"
          );
    }
    stopTime = System.nanoTime();

```

```java
    // [...] benchmark system out

    worker.shutdownNow();
  }
}

/*
 * RunnableCallbackFunction.java
 */
import java.util.HashMap;

/*
 * The Callback function instance.
 */
public class RunnableCallbackFunction implements Runnable {

  // The hashhmap is used to store variables and their value as the scope
  private HashMap<String, String> mapScope;
  private Object context;

  public RunnableCallbackFunction( HashMap<String, String> scope, Object
      context ) {
    this.mapScope = scope;
    this.context = context;
  }

  // If this is executing, we didn't wait long enough and the
  // benchmark time is compromised
  public void run() {
    System.out.println( mapScope.toString() );
  }

}

/*
 * TimeoutContext.java
 */
public class TimeoutContext {
  private long idleTimeout = 1;
  private long idlePrev;
  private long idleNext;
  private long idleStart = 140000505;
  private String onTimeout = null;
  private boolean repeat = false;

  public TimeoutContext( String cb ) {
    this.onTimeout = cb;
  }
}
```

Listing C.1: Closure Benchmarking: Java Code

C.2 JavaScript

```
/*
The function deferral measurements in node.js
*/

var deferredFunction = function ( numScopeVars, delay, scopeId ) {
  var scope = {};
  for ( var i = 0; i < numScopeVars; i++ ) {
    scope[ "id" + i ] = "12345678"; // 8 bytes per stored scope variable
  }
  setTimeout( function () {
    // If this is executed we didn't wait long enough
    console.log( JSON.stringify( scope, null, '  ' ) );
  }, delay );
}

var numOfFunctions,
    numOfScopeVars = process.argv[ 3 ];

numOfFunctions = Math.pow( 2,  process.argv[ 2 ] );

var time = process.hrtime();
for (var i = 0; i < numOfFunctions; i++) {
  deferredFunction( numOfScopeVars, 1000 * numOfFunctions, numOfFunctions
       + "(" + i + ")" );
};
var diff = process.hrtime( time );

var mem = process.memoryUsage();
// [...] benchmark system out
process.exit( 0 );
```

Listing C.2: Closure Benchmarking: JavaScript Code

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