Department of Informatics
University of Fribourg (Switzerland)

A Model-Driven, Component Generation Approach for the Web of Things

THESIS
presented to the Faculty of Science of the University of Fribourg (Switzerland)
in consideration for the award of the academic grade of *Doctor scientiarum informaticarum*

BY
ANDREAS RUPPEN
from
Saas-Grund (VS)

Thesis No. 1905
UniPrint, Fribourg
2015

Accepted by the Faculty of Science of the University of Fribourg (Switzerland) upon the recommendation of:
Prof. Dr. Ulrich Ultes-Nitsche, University of Fribourg (Jury president)
Prof. Dr. Jacques Pasquier-Rocha, University of Fribourg (Supervisor and first reporter)
Prof. Dr. Beat Hirsbrunner, University of Fribourg (Second reporter)
Prof. Dr. Jacques Savoy, University of Neuchâtel (Third reporter)

Fribourg, June 19., 2015

Thesis supervisor	Dean
Prof. Dr. Jacques Pasquier-Rocha	Prof. Dr. Fritz Müller

Acknowledgements

A Ph.D is a journey that can only be completed with the aid and support of professors, colleagues and family. I would like to take this opportunity to thank a few of those who supported me throughout this thesis.
First, and foremost my deepest gratitude goes to my thesis supervisor, Dr. Jacques Pasquier-Rocha. I wanted to be part of his research group ever since my undergraduate days. He gave me the opportunity to work for his research group as an undergraduate teaching assistant, while teaching me the fine art of writing *good* code. As my supervisor he introduced me to the field of scientific research and throughout my thesis I benefited from his experience and knowledge. I would like to thank him for the countless hours spent with me, discussing, motivating and pushing me further in my work.
Earning a Ph.D also involves cooperating with other researchers. This sometimes opens up new perspectives and leads to the unexpected. I would like to thank Dr. Sonja Meyer, a former Ph.D. student in our lab. Our discussions not only lead to common research results and publications but, more importantly, also to a deep friendship. She kept my motivation up when I needed some emotional backup.
I would also like to thank Dr. Dominique Guinard, a visionary and great researcher, for introducing me to my field of research. Through his advice, I not only learned a lot about the Web of Things but also about life as a researcher. My thanks also go to Simon Mayer for the time spent philosophizing about our lives as Ph.D. students and for the more engaging discussions about our respective research.
Working in the Ph.D. program also gave me the opportunity to supervise students' bachelor's and master's theses. I would like to express my gratitudes to all of them who followed me and helped to realize my vision of the Web of Things. Special thanks goes to Jean-Daniel Mathieu. In his Bachelor thesis he not only contributed to build my Web of Things ecosystem but also became a great friend outside work and university.
On a more personal note, I would like to express my deepest gratitude to my family and beloved ones for their unconditional support during this long journey; to my sister and my brothers-in-law for always reminding me there is life outside university; to my mother for always believing in me and motivating me throughout my journey; to my father for his scientific curiosity which impressed me as a child; to my partner, Denise, for her endless patience and support. It would have been an impossible endeavor without such an understanding partner.

Abstract

Since its beginning in the late 80's, the Web has evolved from a simple infrastructure built to serve files from remote machines to a participatory Web, where clients (people and machines) actively participate to create the content of web sites. About the same time, the Internet of Things took its first steps at the Auto ID Center. In parallel the automobile industry started tagging components in their assembly lines with QR codes for quality control. Adding tags to parts allows them to be directly identifiable by a machine. Since then, the Internet of Things has evolved to integrate smart devices (sensors and actuators) in pervasive systems able to sense the environment or to act on it. Finally, the adoption of HTTP as a fully-fledged application protocol for the Internet of Things led to the Web of Things. This thesis pushes the research on in three directions: (1) an extension of the Web of Things, (2) a meta-model and (3) a component based methodology.

First, the xWoT is an extension of the classic Web of Things. As many applications built on top of the Web of Things depend on additional services and algorithms, the xWoT considers physical devices, virtual services and algorithms as first class citizens. From a client's perspective, it is impossible to tell virtual and physical services apart, which supports this vision.

Second, a meta-model tailored for the xWoT (extended WoT) formally defines the different actors and their relationships. It considers the aspects related to smart devices and those important for creating services. The meta-model also takes care of an event mechanism so that applications can build on notifications to launch specific actions.

Third, instead of thinking of how to create mashup applications, this thesis takes a step back and defines a methodology based on components to create new smart devices. The methodology is based on the meta-model and supported by specialized tools for translating models into code skeletons.

To conclude, in our vision we consider such a component based approach leading to reusable pieces of soft and hardware as the first step towards a world where it is easy to develop new smart devices, and where creating mashup applications tailored to a given situation become technically feasible for the average user.

Kurzfassung

Seit den Anfängen des Webs in den achtziger Jahren hat es sich stark verändert. War es zu Beginn nur eine einfache Infrastruktur welche Klienten Dokumente auf Servern zur Verfügung zu stellen, sind die Klienten (Personen und Maschinen) heute die Hauptakteure im Web und stellen selbst Inhalte zur Verfügung. Ungefähr zur selben Zeit entstand das Internet der Dinge am Auto ID Zentrum. Parallel dazu, entwickelte die Autoindustrie QR codes. Diese wurden dann auf alle Autoteile angebracht und erlaubten die Einführung einer einfachen Qualitätskontrolle direkt auf den Produktionslinien. Das Internet der Dinge hat sich seither weiter entwickelt und umfasst heute auch intelligente Objekte (Aktuatoren und Sensoren) welche in pervasiven Systemen die Umwelt erfassen und verändern. Die Kombination von HTTP mit dem Internet der Dinge führte schlussendlich zum Web der Dinge. Diese Doktorarbeit fokussiert sich auf die drei Folgenden Ziele: (1) eine Erweiterung des Web der Dinge, (2) ein Meta-Model und (3) ein Komponentenbasierte Methodik.

Das xWoT is eine Erweiterung des Web der Dinge. Viele Applikationen welche auf dem Web der Dinge aufbauen hängen von zusätzlichen Services and Algorithmen ab. Deshalb behandelt das extended WoT nicht nur intelligente Objekte sondern auch rein virtuelle Objekte und Algorithmen as Bürger erster Klasse. Aus der Perspektive eines Benutzers besteht zwischen diesen Kategorien sowieso kein Unterschied. Deshalb macht es Sinn, alle gleich zu behandeln.

Ein auf das extended WoT zugeschnittenes Meta-Model bennent und definiert die verschiedenen Akteure und ihre Relationen. Dazu betrachtet das Meta-Model Aspekte welche für intelligente Objekte wichtig sind aber auch solche welche für rein virtuelle Objekte ein Interesse darstellen. Das Meta-Model bildet zusätzliche ein Infrastruktur ab um Informationen von intelligenten Objekten zurück zum Kunden zu schieben. Diese Benachrichtigungen erlauben es dem Kuden bestimmte Aktionen in Abhängigkeit der empfangenen Nachricht auszuführen.

Statt, wie viele andere Arbeiten, sich mit der Umsetzung von Mashup Applikationen zu befassen, konzentriert sich diese Doktorarbeit auf Struktur der Komponenten auf welchen solche Mashups aufbauen. Dazu wird eine Komponentenbasierte Methodik eingeführt welche auf dem Meta-Model basiert und zu neuen intelligenten Objekten führt. Die Methodik wird dabei von einer Anzahl Hilfsprogrammen unterstützt welche Instanzen des Meta-Models in Code Artefakte umwandeln.

Wir glauben dass solch eine Kompontenbasierte Methodik der erste Schritt in Richtung von wiederverwendbaren intelligenten Objekten ist. Diese führt einerseits zu einem

schnelleren Entwicklungszyklus für intelligente Objekte und macht andererseits die Herstellung und Anpassung von Mashup Applikationen einem breiteren Publikum zugänglich

Keywords: Web of Things, REST, Meta-Model, Model-Driven, Software Component, Internet of Things,

Table of Contents

List of Figures

Listings

List of Tables

1 Introduction

1.1. Motivation

Following Moore's law [55], the transistor count per square inch doubles each year. As a side effect, costs of a given device (like a CPU) become cheaper each year, even if the decrease in price is not as much as the increase in transistors. These two factors lead to the appearance of low-cost hardware with limited communication capabilities. The Internet of Things is an Internet where the participants are Things [42]. A Thing is mainly a piece of hardware, like a sensor, with some communication capabilities. Thus, Moore's law is an enabling factor for the Internet of Things (IoT). Technological advances over the past decade have lead to really cheap devices, full of sensors, having reasonable autonomy and connectivity like the Arduino boards [WEB5], the Sun SPOT [WEB58] or more recently the UDOO [WEB66].

Over the past few years, researchers have shown great interest in connected Things. Projects like Cooltown [39] have emerged. All these research efforts have led to different approaches on how to build smart devices, the foundation of the IoT. More recently, the Web of Things (WoT) appeared [19]. The WoT stands on top of the IoT by adding a standardized layer, connecting individual smart devices together. Instead of defining new architectures for each scenario, which leads to a fragmented IoT, it uses the web as a fully fledged application protocol. An enabling architecture consists of using RESTful web services. Indeed, on the one hand they present several advantages over their WS-* counterparts [20] and, on the other hand, they represent today an acceptable way of designing architectures for smart things [19, 27]. Another trend to take into consideration is the shift from using single raw services towards integrating them with computationally complex processes [50, 70]. This shift is motivated by the observation that the number of Thing participants in the Web will exceed that of human ones by a wide margin [WEB63]. These Things will produce a quantity of information impossible to treat directly. To manage this flood of information, it needs to be aggregated and "distilled" somehow.

Producing smart devices is a necessity to build the IoT. Today, producers of smart devices find themselves in a similar situation to software architects and engineers in the past. For each new smart device that hits the market, the producers start from scratch, designing the hardware part, and later, add on an arbitrary Application Programming Interface (API). Whereas the dark period of software engineering was called the `software crisis`, the situation faced today could be termed the `things crisis`. Each vendors creates his own eco system, most of which are incompatible with each other. Back in the 80's Cox [46] and others proposed a component based approach to solve the problem of software. Instead of starting over and over again, software engineers need to produce reusable and deployable components. The same pragmatic approach could be the solution to the `things crisis`. Visions like smart homes and smart environments only have a chance of becoming true if their complexity is broken down into components, thus giving the user the chance to build an environment based on blocks coming from different vendors.

1.2. Contribution

Section 1.1 introduced and discussed the enabling factors of the IoT. One of the problems identified is the `things crisis` faced today. Although media claims that the year 2015 will be the year of the IoT [WEB60], this will only come true if vendors manage to escape the things crisis. Implementing closed environments in which proprietary smart objects from only one vendor can evolve will definitely lead to failure. In order to avoid failure, this thesis suggests an approach which will avoid this situation. Given this foundation, several open questions can be identified. The following issues will be addressed in this thesis and present its main contribution.

- This thesis proposes an extension of the Web of Things to take into consideration how the generated data can be distilled so that clients can use it. This extension revolves around two main factors: (1) With the predicted increase of non-human participants in the Web, approaches like combining data from different sensors, actuators and tags into a nice application will not be sufficient anymore. Such combinations, also called *mashup applications*, will overwhelm the user with the quantity of information they deliver. Instead, information needs to be filtered, distilled and aggregated to suit a user needs. This thesis presents an integrated approach where Things, aggregations and other sorts of algorithms are all equivalent and treated as first class citizens. (2) Following a given Thing, is just as realistic as following a blog or a Twitter feed today. However, due to its underlying architecture, the web was not intended to push information from servers to client. This thesis exposes a novel way for information to be pushed from a Thing to a client, without violating the fundamental principles of the Web of Things (WoT). Adding these two factors to the WoT leads to an extension of the latter, that we shall refer to as extended WoT (xWoT).
- Today, the WoT lacks a clear vocabulary and structure. This shortcoming is problematic when different people with different horizons try to team up. It also makes it difficult to provide recipes for how to build applications embedded in the Web of Things. To overcome these limitations, a meta-model for the WoT is discussed in detail. Together with the meta-model comes a tool set composed of an IDE

integrating the meta-model and tools to compile models into code. The IDE gives developers the possibility to create instances of this meta-model. These instances can later be translated into executable code.

- Whereas researchers have already spend a lot of time investigating how to build mashup applications [23, 5, 76], less work has been undertaken to define the building blocks of the latter. In order to overcome this gap, this thesis proposes a component oriented approach to structure the building blocks of mashup applications. These blocks have a well defined outer interface exploited by the various mashup application but also a clearly defined inner structure. Together with the meta-model defining the inner guts of these components, a model-driven component generation approach is sketched out. This approach helps developers to start new applications by giving them conventions to follow and the necessary tools to support these conventions.

1.3. Organization

This thesis has three parts. Part I outlines the historical background of the key architectures used throughout this thesis. These include the evolution of the web in Chapter 2, the evolution of web-services in Chapter 3 and the evolution of Things in Chapter 4. These chapters provide an understanding of the beginnings and development approaches in the domain of the Web of Things.

Part II introduces the theoretical background necessary for the remainder of the thesis. In order to come up with computer assisted WoT (Web of Things) compliant component generation, a few theoretical foundations are necessary. Chapter 5 defines the concept of Representational State Transfer, RESTful web services and Resource Oriented Architecture. It also includes a discussion of HTTP, which is fundamental to REST and Resource Oriented Architecture. Chapter 6 introduces the notion of software component, a ready to use piece of code which can be deployed in a container. Although this is quite an old principle, it is still widely used in computer science and also applies to the WoT. Chapter 7 explains how meta modeling works and its usages. With the aid of a simple example, this chapter illustrates how meta-models are created and used to create instances.

Finally, Part III contains the main contribution of this thesis. The extended WoT is defined as an extension of the WoT and Chapter 8 formally introduces it. Chapter 9 contains the xWoT meta model and a discussion of how it is applied to structure the xWoT and create ready to deploy xWoT components. Finally, Section 10 closes Part III with a few example use cases to showcase different aspects of the xWoT meta model and how it can be applied in different situations to semi automatically build xWoT components.

1.4. Notations and Conventions

- Formatting conventions:
  - Abbreviations and acronyms are as follows:

    - When an acronym in common use appears for the first time, its full unabbreviated form follows in brackets, as in IBM (International Business Machines Corporation).
    - If both, the acronym and the full name are equally common, when it is used for the first time, it is written, for example: Java Persistence API (JPI)
    - For all subsequent references, only the acronym is used: JPA
  - `http://localhost:9090/eHealthServer` is used for web addresses;
  - Code is formatted as follows:

```
1 public double division(int _x, int _y) {
2   double result;
3   result = _x / _y;
4   return result;
5 }
```

- This work is divided into eleven chapters, each with sections and subsections. Every section or subsection is organized into paragraphs signaling topic changes.
- **Fig.**s, **Tab.**s and **List.**s are numbered with the initial digit of the chapter. For example, a reference to **Fig.** *j* of Chapter *i* will be labeled ***Fig.*** *i.j*.
- While aware of gender concerns, I systematically use the masculine form for simplicity to refer to both genders.
- If figures are not the authors work, or if a figure is highly inspired by some other figure, the reference is given in the caption but not in the *List of Figures*. Furthermore, the following convention holds for these citations:
  - Figures copied from other sources have a caption like:
    Decorator Pattern (from [B6, p. 177])
  - Adapted figures where the original one is clearly identifiable have a caption like:
    The Semiotic Triangle merged with the Seidewitz' terminology (after [B10, p. 4])
  - Figures modified from an original figure have a caption like:
    Another view on meta-models (modified from [B9, p. 28])

Part I.

Historical Background

Foreword

In order to understand where we go we need to understand where we come from or as George Santayana put it: — „Those who are unaware of history are destined to repeat it.“

2

Evolution of the Internet and the Web

2.1. Introduction

The Web of Things (WoT) is highly dependent on the web, the concept of services and smart objects. Therefore, to define the WoT, its supporting architectures, concepts and technologies have to be discussed first. As the name says, the Web of Things is related to the Web, but also to Things. So, what exactly does the *Web* mean, what *Web* the acronym references and what are these *Things*?

Through the following three sections, this chapter lays the foundation for giving a clear definition of the Web of Things. To do this, Section 2 first discusses the Web, the types of web, and their evolution. Section 4 defines Things and thus the WoT after Section 3 has formally introduced web services, characterized the different types of web services and outlined their differences.

The web as we know it today has a long history. The problem is, that unlike other technologies which have revolutionized modern life, the Internet has no inventor. This makes it more difficult to trace its evolution from the early beginnings to modern times. Historically, networks like phone lines, tend to be *circuit-switched*. Yet, in the early 60's, several technological innovations started to change the way telephone land-lines worked. *Packet-switched* lines have several advantages over the circuit-switched "pendant". This is one of the big inventions supporting the Internet [B12, 40, B2] and has many advantages over circuit switching. Not only can one line be used by several users simultaneously, the system is also more robust. If one link breaks, the packets can be rerouted over

another link and still reach their destination. This approach proved to be successful, since modern networks still use it for routing their traffic between peers. The research around this topic has been quite robust: besides the PhD. thesis of Kleinrock [40], and Paul Baran's work *On Distributed Communication* [B2], several other researchers around the world have worked on this topic. One of them was Donald W. Davies at the National Physical Laboratory in London. Since he only got funding for one node but needed several to do his research, he connected a handful of terminals to the latter, thus creating the first Local Area Network (LAN).

The military founded the ARPA (Advanced Research Projects Agency, which later became DARPA) in the late 1950's with the goal of ensuring the U.S military's technology was more sophisticated than the enemies'. Nonetheless, this goal rapidly began to push the frontiers of science and technology beyond military requirements. Ten years later, in the late 60's, the ARPANET (Advanced Research Projects Agency Network) was funded by the ARPA. It became one of the first working *packet-switched* networks. The motivation for the development of a robust communication system can clearly be found in the cold war between the U.S and Russia. When Paul Baran, one of the inventors of the modern Internet, designed a packet-switched network for voice, his aim was to give it the capacity to continue working in case a node disappeared through a nuclear strike.

At first the Advanced Research Projects Agency Network (ARPANET) only had four participants through the Interface Message Processor (IMP). These four research institutes were: Stanford, Utah, California (UCLA) and Santa Barbara (UCSB). Eventually, in 1972, ARPANET reached a stable and usable state. Rapidly the ARPANET gained momentum and ten years later, in the early 80's, the number of connected nodes was already over 200. However, development did not stop with the success of that system. Bob Kahn, another father of the Internet was visionary in many ways. He had the idea that the Internet was not one big network but merely connections between several corporate networks. This vision, opened new perspectives and challenges like addressing networks and individual machines within those networks. Furthermore, he identified the problem of reliable connections with unreliable hardware. The outcome of Kahn's research, in the mid 70's, is the TCP/IP stack, which took several years and many iterations to develop. By the mid 80's, the stack was sufficiently stable to be an enabling factor of the future Internet revolution.

The Internet as it is known and used today is not an invention which can be precisely dated in history. It is more of a continuous development and research process over 20 years involving some of the best computer scientists around the world. More historical background about ARPANET and the beginning of the modern Internet era can be found in [B3] and [74]. The TCP/IP stack ushered in a new era in computer science. Soon ARPANET was replaced by the National Science Foundation Network (NSFNet) and eventually shut down in 1989. By this time, Tim Berners-Lee was working at CERN (Conseil Européen pour la Recherche Nucléaire) in Geneva. In an effort to rapidly exchange research results between different offices around the world, Tim Berners-Lee and his colleague, Robert Cailliau, invented, in 1989, the World Wide Web (WWW), also called the Web. Their idea was that machines can serve documents containing hyperlinks to other documents, possibly hosted on other machines. Such documents are formatted in HTML (Hypertext Markup Language) [rfc1866]. In order to connect with a server, a client needs a web-browser capable of requesting and interpreting Hypertext Markup Language (HTML) documents. HTML as well as the first web-browser and, of course, the

first web-server were all developed by Tim Berners-Lee. Often the Web and the Internet are used to designate the same thing which, of course, is not true. While the Internet is a network architecture, the Web is a set of protocols using the Internet to transport packets. However, the Web is not the only protocol using the Internet as its underlying infrastructure; protocols like Internet Relay Chat (IRC) and email existed well before the invention of the Web.

Yet, the WWW (World Wide Web) is not the first attempt to create a document oriented network. At almost the same time, the university of Minnesota developed the *Gopher* protocol [rfc1436]. It aimed to replace the rather fussy to use FTP by a more elegant way of browsing documents. Yet another approach is *HyperCard* developed by Apple. Neither system had the success of the Web for two reasons: (1) The Web allows unidirectional links between documents [WEB21] and (2) the Web is a royalty free protocol. Everybody can read the specification and write clients for this protocol. An example is in 1992 when Marc Andreessen presented his web-browser *Mosaic for X* (which later evolved into *Netscape*). Both, Gopher and HyperCard have a licensed model for commercial usages which limits their application. For non-technical users, the Web made the Internet available to the masses. The increasing density of nodes connected to the Internet, the advent of commercial Internet Service Providers (ISPs) and the availability of a modern web-browser like *Netscape*, allowed users for the first time to take advantage of this technology. As will be discussed in the remainder of this section, the Web has steadily evolved from its protype to its current version. Yet, at its basis it still shares some important aspects with the Web of 1991. This shows that the choices made in the early version proved to be sound and the different iterations are merely evolutions rather than revolutions.

When Tim Berners-Lee makes the first public demonstration of the WWW, he envisaged of a document oriented network for researchers. Along with HTML to mark up the published documents, he also presented a new protocol allowing documents to be browsed: the Hypertext Transfer Protocol (HTTP) [rfc1945]. It is simple to implement in browsers and servers, yet sufficiently powerful for the intended purpose. Twenty years later, it is still the standard for the Web and beyond. In fact, many use the protocol to provide services of all kinds. The most prominent example is RESTful web services which use HTTP as a fully fledged application protocol to provide a service over the web. By proposing a way to execute the four standard CRUD (Create, Read, Update, Delete) operations, the protocol is also available for service delivery.

2.2. Web 1.0

Back in 1991, when Tim Berners-Lee and Robert Cailliau presented to the public the first version of a working World Wide Web, it was only composed of static documents and hyperlinks pointing to other, possibly related, locations or documents. Its purpose was to transmit static content over a network. Documents were formatted using an early version of HTML. The concept was to annotate the text with tags. These tags are defined as SGML and mostly used in pairs, surrounding portions of text. All HTML documents share a common structure shown in Listing 2.1. Therefore, a valid HTML document is made of three major elements: the *document type declaration* in line 1, the header containing meta-information about the current document, shown between lines 3 and 6

and finally the body containing the visible elements later presented in a web-browser from line 7 to 11.

```
1  <!DOCTYPE html PUBLIC "-//IETF//DTD HTML 2.0//EN">
2  <html>
3    <head>
4      <title>World Wide Web</title>
5      ...
6    </head>
7    <body>
8      ...
9      <em>Here goes the content<em>
10     ....
11   </body>
12 </html>
```

List. 2.1: Structure of an HTML Document

Besides HTML, HTTP (Hypertext Transfer Protocol) and HTTPS (Hypertext Transfer Protocol Secure) are two enabling factors for the birth of the Web. Figure 2.1 show the first document published on the WWW. CERN displayed it again on the occasion of the 20th anniversary of the Web. These technologies made information exchange, browsing and reading much easier than before, nonetheless the Web 1.0 was invented by researchers and targeted scientists. The available content envisaged a research community. This fact however quickly changed so changing our daily lives.

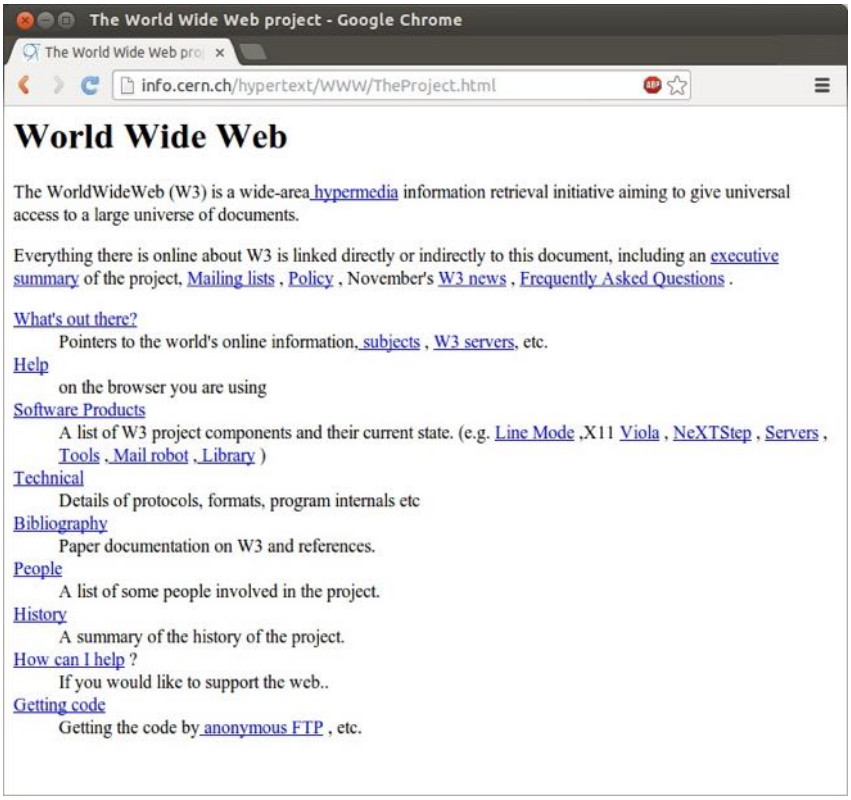


Fig. 2.1.: First Web-Site hosted at CERN

Since this early version of the Web, many things have changed and evolved to allow for a more modern Web. Changes were made to all aspects of the Web. Whereas, there were

only a few iterations of HTTP, the HTML standard drastically evolved from its primary (numberless) version to HTML5 by adopting different intermediary steps like the advent of XHTML where, instead of the antique SGML (Standard Generalized Markup Language) , the newer XML is used as the underlying document type definition [WEB2].

2.3. Web 1.5

The Web 1.5 is not a revolution from the previous version; it is merely a small evolution. The content has evolved from static HTML pages to dynamic ones. This means the content can change for each request, or it can adapt to other events (dates etc.). Whereas, until then, the administrator of a weather-related website had to manually adapt the HTML coding of the web-page to reflect changes, he could now achieve the same result dynamically, for example by using a database. This dynamic content approach was quickly picked up by the first search engines. Since HTML pages were no longer statically defined, the results of a search were now presented individually without having to multiply the number of HTML pages. Figure 2.2 shows the web-site of one of the first search companies *altavista.com* back on May 9th 1997.

Fig. 2.2.: Web-page using some scripting

Two major steps leading towards the web as we know it today were already observable in the screenshot of the Alta Vista web-site back in 1997: (1) the introduction of advertising, although not overwhelming and (2) the availability of business logic behind a website. In the case of altavista.com, this business logic is some kind of search algorithm. But for other websites, this could be something else. The introduction of forms in HTML2 [3] is key to the success of Web 1.5. Forms provide a convenient way for users to input data, which is then processed by the web-server. Yet, on their own, they cannot make work Web 1.5. Even though form data arrive on the server side, they have to be processed somehow. To do this, Web 1.5 introduced the Common Gateway Interface (CGI). This technology allows communication between a web-server and some third-party script or

application. Such applications are independent of the service and can be written in any language supported by the operating system running the web-server. The best known implementation of such a CGI binding is probably PHP (Hypertext Processor, formerly known as Personal Home Page Tools). Whereas in the early days it was just meant as a replacement for a bunch of Perl scripts, it quickly becomes popular among professionals and amateurs alike to create personal homepages. It became so popular that modern web-servers like Apache directly integrate a dedicated PHP bridge and so eliminate the need for a classical CGI binding. Today, PHP is still popular for creating web-sites, be they personal web-sites, business presentations sites, online shopping sites etc. According to *w3tech.com* 81.2% of the sites for which the server side language is known are based on PHP. Knowing this, many popular CMS (Content Management System) [1] use PHP as their foundation framework, which is not surprising.

Another technology which contributed to this change, is Javascript, released in 1996. Although this was not the first client side scripting language, it is the first that was adopted by all major players. Javascript was born of the alliance between Netscape and Sun Microsystems. At that time, Netscape's browser *Navigator* already supported the *LiveScript* language. With LiveScript being a de facto standard, Sun Microsystems could not ignore it, thus, they developed an API (Application Programming Interface) to connect LiveScript with Java objects and called it Javascript.

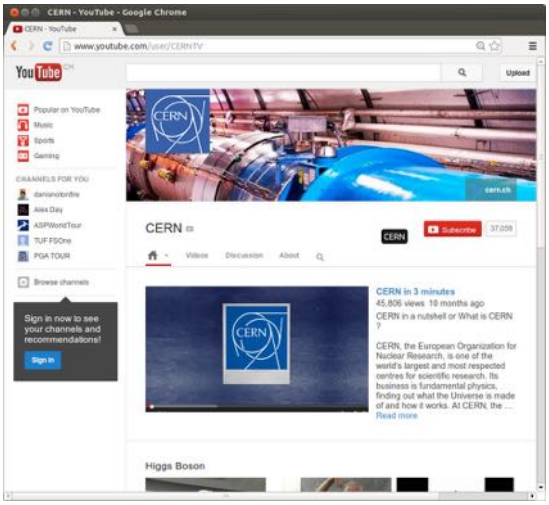
2.4. Web 2.0

Whereas the Web 1.5 was merely a technical evolution of the initial Web, Web 2.0 is a cultural evolution. The underlying frameworks, protocols and technologies have not changed much. Everything was already out there to support this new era of the Web. The Web supports forms to capture inputs generated by users. Additionally, it is possible to process data using several server-side programming languages and databases as storage. Finally, Javascript enhanced with some powerful libraries like JQuery allows for rich user interfaces, making web-sites resemble fully fledged desktop applications.

What is changing, are the users. Whereas users were condemned to play a passive role and only consume content made available by the administrators, they can now play an active role and contribute their own content. This also involves a shift from "content-generating" web-masters to platform providers, where the users can express themselves. Maybe one of the most successful concepts arising out of this shift, are blogs (aka. web-log), which are literally diaries for the World Wide Web. There are blogs targeting almost every aspect of life, be it cooking (`http://notwithoutsalt.com/`), music (`http://www.symphonicandgothicmetal.net/`), technology (`http://www.engadget.com/`), personal blogs (`https://myspace.com/`) or fully customized and self-hosted ones (`http://modx.com/`). Users become eager to participate and create content. This openness also leads to another type of user-generated content, the social platform. Whereas in blogs and forums, people interact around a given topic, on social platforms, they share aspects of their own social lives. Today, social media platforms are part of our lives to the point where even international companies and institutions like CERN can no longer ignore them. Instead, they hire specialists to post content on these platforms to collect the maximum number of followers (which in turn can generate some financial benefits). Figure 2.3

[1] *Wordpress*, *Joomla*, *Drupal* and *ModX* to cite a few of the more popular ones.

(a) Twitter



(b) Youtube

(c) Facebook

Fig. 2.3.: CERN and Social Media

shows CERN's presence on the most popular (according to various sources [WEB73]) social-media websites today: Twitter 2.3a, Facebook 2.3c and Youtube 2.3b.

Besides social-media platforms, where users contribute by sharing details of their private lives, other popular platforms depending on user-contributed content have emerged. What started back in 2000, as a side project to *Nupedia*, a free encyclopedia whose articles were written by experts and reviewed like journal articles, is today the biggest freely available encyclopedia. It seems that many people like to share their knowledge in a particular domain with others. This has lead to a new type of platform which is neither a blog nor a social media one: *Wikipedia* is its most prominent example. To a certain degree, this phenomenon can also be observed on other social-media platforms like *Youtube*, where numerous contributors propose video tutorials on various topics.

2.5. Web 3.0

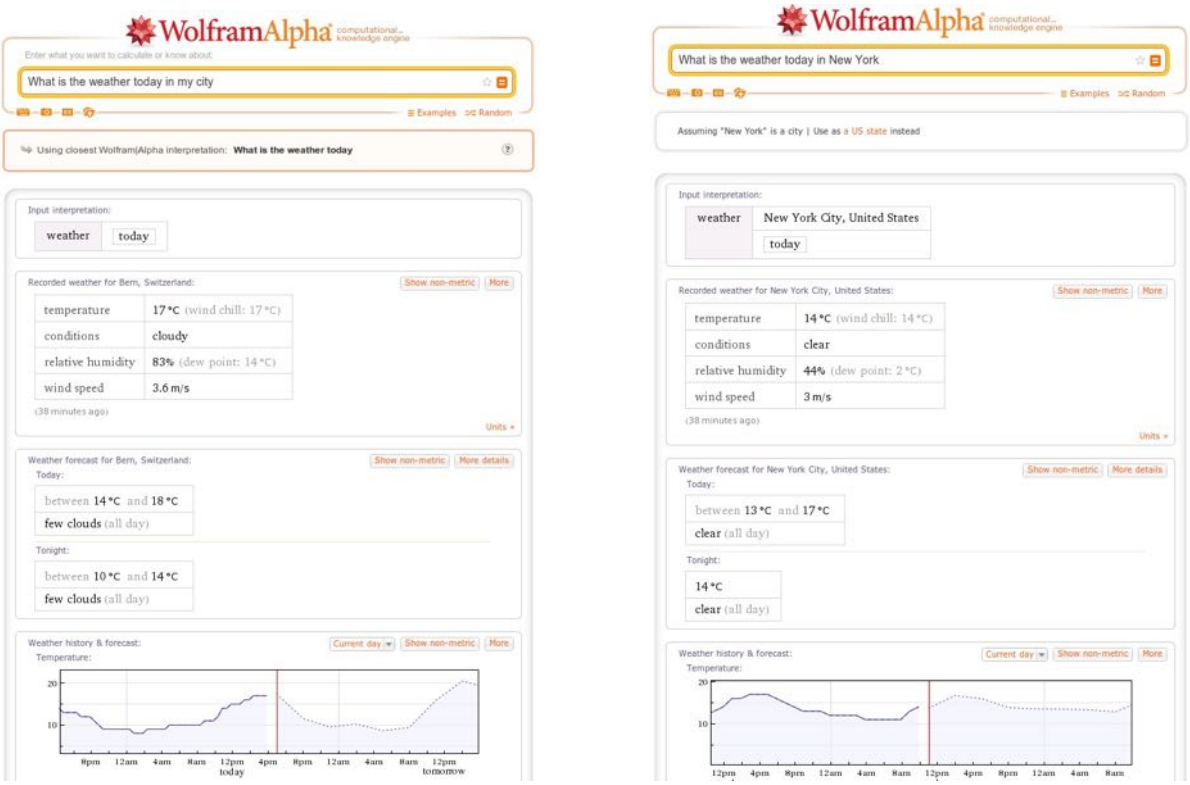
To roughly date the evolution of the Web, we would say that Web 1 and 1.5 appeared before Youtube, Wikipedia, Facebook and Twitter, while Web 2.0 came after and the Web 3.0 is in the future. After the success of the Web 2.0, its evolution forked into several directions. One is the semantic web. Web 1.0 and 1.5 brought the necessary underlying technologies, the Web 2.0 tied the users as a source of information to the web, but there remains at least one considerable problem: searching the web. As the Web grows, it also becomes more difficult to find the information we need. Companies like *Google* try hard to invent and develop increasingly complex algorithms allowing them to classify, order and tag content on the Web. Whereas this seems to work well in most situations, sometimes the Web is just unable to find what a user is looking for.

Here, semantics could greatly enhance the search results. Semantics allow words to be interpreted and give them a meaning beyond the simple order of the letters composing a word. As such, semantics allow a machine to understand queries like "Will it rain tomorrow in my hometown?" More and more products are taking advantage of semantics to enhance the user experience. Siri[2] and Google Now[3] for example both propose an intuitive way to communicate with a digital personal assistant through natural speech. Search engines are also interested in this new approach. Some have already adopted some kind of semantics like *Wolfram Alpha*. Figure 2.4 shows the results after asking Wolfram Alpha for the weather in my hometown 2.4a and after asking about the weather for New York 2.4b.

Yet, the semantic web is only one fork of the current Web. Another trend is the Internet of Things. Integrating sensors as content providers has become more and more common. Projects like the various "Smart Grids" in different countries, try to lower our energy consumption and produce it in smarter ways. This is only possible through real time consumption statistics. At this point sensors become content providers for the Web. Section 4 will give a more in-depth review of the evolution of smart objects in the Internet and the Web.

Only the future will tell what Web 3.0 will be like. The IoT can only benefit from advances in the semantic web. Bringing semantics to the IoT allows for a new range of

[2]http://support.apple.com/kb/HT4992
[3]http://www.google.com/landing/now/



(a) My City (b) New York

Fig. 2.4.: Wolfram Alpha giving the weather for different (fuzzy) locations

scenarios when combining different sensors and actuators to mashup applications. Instead of binding them during the programming of such a mashup, the necessary smart devices could be found on the fly. The reverse is also true. The semantic web benefits from the IoT. Today working examples are mostly of an academic nature like Wolfram Alpha. Furthermore, semantics is often used and cited in the context of smart homes and other ubiquitous applications. These scenario depend on the advances of the IoT to be realized as facts and pass from laboratory essays to commercial products. At this point, there is insufficient experience to decide whether something like Web 3.0 will come into being or what it will be like.

2.6. Key Concepts introduced in this Chapter

This chapter retraced the history of the Web from its beginning, where only primitive exchanges were possible, to a participatory web as we know and use it today. Although, the Web has not changed that much during the past few years, its popularity has greatly increased. Every year fewer households remain without an Internet connection and the rise of mobile devices like smart phones and tablets have participated in this success story.

The key concepts of this chapters are the evolution of the Web from its early beginnings to its actual state. Even though it has become more and more ubiquitous [WEB14] it is still a building block for many modern data exchanges. Whereas for most of its existence, content on the web has been consumed with a browser, more and more clients

prefer native applications built for one content provider (NZZ[4] website as opposed to NZZ applications on Android and iOS). Nonetheless, these applications need some way to bring the data from the service provider to the application running on the clients smart phones. Although, developers have a choice between many different approaches, (see Chapter 3 for an overview of different approaches), one particular approach is becoming more and more popular: REST and RESTful web services - see the market shares of different web service technologies/architectures in Figure 3.4, REST takes the ingredients of the Web and translates them into web services. Additionally, there are other arguments in favor of REST [64].

Since REST architectural style is an abstraction of the Web as we know it, understanding HTTP is a prerequisite to fully understanding REST. REST as an architectural style is not a technology and R. Fielding in his doctoral thesis. [14] only describes its architectural properties, not any concrete implementation. Nonetheless, HTTP remains its most prominent implementation. The fact that Fielding was also actively participating in the development of the HTTP/1.1 standard helps explain why the Web is so important for REST.

Although, there has been a shift from web-pages to mobile applications, the Web will not become obsolete in the near future. Instead, with the growth of ubiquitous computing and more precisely, the massive forthcoming of the WoT will make it more popular [WEB60].

[4]http://nzz.ch

3
Evolution of Web Services

3.1. Introduction

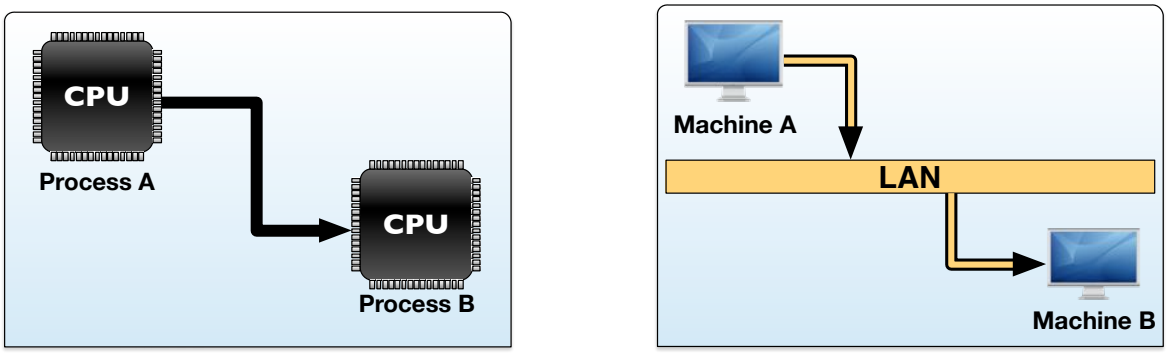
Chapter 2 discussed the evolution of the web and how it changed the way we use it. The web was certainly made for humans. Yet, since the early days of computing, processes and later machines have also needed to communicate in order to complete some tasks. Figure 3.1a shows the situation in the early days of computing. On a given machine there are two processes which need to exchange data. The most simple form of data exchange would be a situation where the output of the first process is the input of the second. Such situations are still very common. For example, how can the number of words in the `main.tex` file of the present thesis be counted. Listing 3.1 shows that this task can easily be achieved using two separate applications, where the output of the first is used as input for the second. In fact, the word counting process (`wc`) needs a document on which to operate as input. On the other hand, `cat` outputs a given file on the command line. Thus, to achieve the desired effect, it is sufficient to send the output of `cat` as input to `wc`. Passing the output from one process as input for another is solved by pipes. Listing 3.1 uses an anonymous pipe to achieve the desired effect. Whereas pipes are still common today, other approaches to sharing common information between processes are less used nowadays. Among them, shared memory, where process A writes somewhere in the memory space also accessible to process B. Later, when process B is launched, the memory space contains all the necessary information for process B to run. Yet another approach having multiple processes running on the same machine to communicate with each other is COM[1] developed by Microsoft. The advantage of this approach is the

[1] `http://www.microsoft.com/com/default.mspx` (last visited 28 oct. 2013)

```
ruppena@tungdil:~$ Thesis: cat main.tex |wc -w
755
```

List. 3.1: Unix Pipes in Application

company behind and supporting it. Microsoft, by delivering simultaneously an operating system and the COM (Component Object Model) interface, can implement this interface in all major parts of the OS (Operating System). This allows developers to quickly join two processes together without struggling with incompatibilities or strange APIs.



(a) Process Pipes (b) Over a LAN

Fig. 3.1.: From Processes to distributed Machines

3.2. DCOM, Services over a LAN

The increasing number of machines connected to a LAN has lead to new ways on how they process data. Previously, only processes running on the same physical machine could coordinate. However, once several machines are connected over a LAN, tasks can be distributed over several machines. This vision is the foundation of any modern n-tier architecture. Microsoft has proposed a standard for how distributed processes should communicate over a LAN. Of course, their approach is based on the inter-process communication interface, Component Object Model (COM), and is called DCOM (Distributed Component Object Model). Yet, many other approaches have arisen, like Remote Procedure Call (RPC). These approaches are the first web services in that they deliver some functionality, that is, process some input and return some output over a network.

Although the concept sounds simple and not very far away from what inter process communication looks like on the same machine, distributed processes bring new challenges. The biggest is the problem of different OSs hosting the different processes. This leads to interpretation problems when reading the messages sent from one processes to another. Whereas, for example, the first process might use big endian to encode values, the second process might work with little endian encoding and interprets the received messages as such, leading to erroneous computations. To overcome this problem, Sun Microsystems together with a handful of other companies defined the XDR (External Data Representation) communication standard [rfc1014]. It is the first OS-independent communication

standard. External Data Representation (XDR) defines the most common datatypes like *Integer*, *String*, *Float* and *Array*. However, the developers are still responsible for dealing with network related tasks, like sockets and protocols, to make the two processes communicate. For each new project, a new communication layer connecting the machines involved is developed. Besides being error prone, this task is also time-consuming and, as a result, projects get more expensive. Yet, since all these implementations share some common structures, rapidly middlewares taking care of the underlying network aspects became popular.

3.3. Middlewares

The simplest way to send information from a process running on machine A to another process sitting on machine B is to open a socket on each machine [rfc147]. This approach is straightforward and very easy to implement. However, as simple as it might seem at first glance, writing such code is complex and leads to all sorts of problems for developer to deal with. One of these problems is the port number on which the service will run. Different developers naturally choose different ports when they are tasked with implementing the same service. Therefore, collisions in port number attribution are programmed. From the large quantity of Request for Comments (RFC), it seems obvious that a considerable effort has been made to standardize port number usage [rfc617, rfc204, rfc433, rfc349, rfc503], to cite only the attempts made in the early 70s. Once the port number to use has been decided and fixed, and the channel opened, the developer needs to worry about how to use this channel. Therefore, he needs to choose a suitable protocol or, if none is found, develop one. Developing such a protocol implies writing code taking care to anticipate the various errors which might occur during a session, like erroneous message or lost messages and so on. Up to this point, not one line of the final application and of its business logic is written. Only the underlying base structure has been implemented so far.

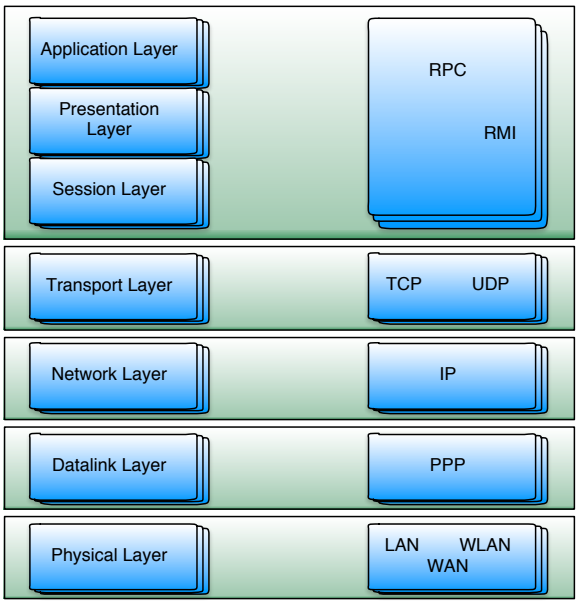


Fig. 3.2.: RPC in the OSI Stack (modified from [B21])

Most of these problems can be avoided when using a middleware. The middleware takes care of defining sockets, protocols and message formats. Additionally, the middleware is responsible for handling protocol and network related errors, shielding them by the final application from the underlying transport layer. They do this by providing the developers with a simple generic Application Programming Interface (API). This approach follows the recommendations of the OSI (Open Systems Interconnection) model [B21, 36]. Figure 3.2 shows where to place service-oriented middlewares with regard to the Open Systems Interconnection (OSI) model. By placing the middleware just above the transport layer, it is responsible for all connection related work letting the developer concentrate on the application.

Over time, several middlewares were developed. Alonso et al. [B1] classify them into the following six categories: RPC-based systems, TP monitors, Object brokers, Object monitors, message-oriented middlewares and message brokers.

- *RPC*-style middlewares are the most basic type. They aim to convert a procedure call into a Remote Procedure Call to take care of how the necessary information is transported to the remote procedure provider and the generated outputs back to the client. Many modern middlewares use RPC as their foundation. Clearly, implementations like Java RMI and SOAP, can be identified as RPC-style middlewares.

- *TP monitors* are the oldest type of middleware. Roughly, TP monitors are RPC like systems with transactions. For this reason they are often used to provide databases with RPC query interfaces.

- *Object Brokers*. With the generalization of object-oriented paradigms and the wide adoption of object oriented programming languages, developers asked for RPC-like middlewares capable of handling objects instead of raw procedure calls. The most prominent implementation of such an Object Broker middleware is CORBA.

- *Object monitors* are the convergence of TP monitors and Object brokers. Soon it became apparent that TP monitor functionality, originally only developed for procedural language, also needed to cover object-oriented languages. On the other hand, it also became apparent that object brokers needed the functionality provided by TP monitors.

- *Message-oriented* middleware takes special care of message queues. Developers rapidly saw that some exchanges are not necessary synchronous and added support for asynchronous interactions to RPC. This approach was also ported to TP monitors. However, developers rapidly discovered the usefulness of message queues as middlewares themselves. Thus, message-oriented middlewares provide transactional access to message queues.

- *Message brokers* have the ability to transform the messages in the queue before they are forwarded to the real business logic for further processing.

Although they are all different, most middleware are based on RPC and add more functionality to it, be they transactions, asynchronous interactions or the smart handling of message queues.

3.4. RPC and the first Services over the Internet

Thus far, processes were either running on the same physical machine and exchanged information using approaches like shared memory or pipes, or they were deployed on different machines sitting on the same LAN. Whereas approaches like DCOM worked quite well for processes sitting either on the same machine or at least the same LAN, it seemed that this approach did not work anymore when service components are deployed on machines reachable over the Internet. This merely comes from the fact, that services then had to address firewall issues, NATs (Network Address Translations) and other network critical hardware and protocols not necessarily present in a LAN. Thus, traversing the Internet instead of a LAN (Local Area Network) increases the complexity of the connection layer. As discussed in Section 3.3, middlewares have to handle such aspects and provide developers with a nice, and easy to use interface. This section describes such middlewares and how they overcome the limits of LAN to deliver services over the Internet.

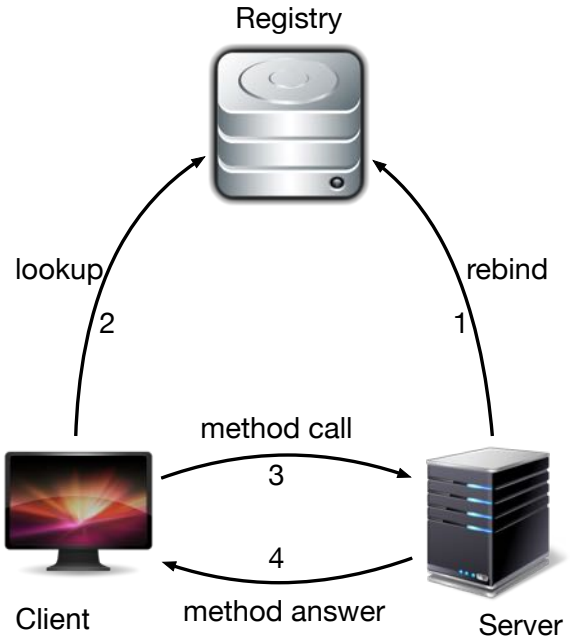


Fig. 3.3.: RMI function call (after [WEB50])

One of the most successful middlewares is Remote Procedure Call (RPC), developed in the 1980s by Birell and Nelson [4]. At that time, most programming languages were procedural (see Andrew Freguson [WEB1] and as pointed out by him, Wikipedia [2] for a more recent list) and so is RPC. RPC allows developers to call procedures located on physically separated machines, just as if they were local. By that, RPC makes calls to distant procedures transparent to developers (it is controversial whether such middlewares should be transparent to the developer or not) and lets them work with them as they would work with any other procedure. Having processes running on different machines communicating with each other is a requirement for n-tier applications. Thus, RPC is an enabling factor for the first 2-tier systems. Additionally, RPC defines concepts still used in any remote service environment like the Interface Definition Language (IDL),

[2] `http://en.wikipedia.org/wiki/History_of_programming_languages` (last accessed Feb 4, 2015)

a directory and a naming service to find appropriate services available on the Internet; dynamic binding, which allows for selecting a concrete service only at runtime. Figure 3.3 shows the necessary elements for calling a procedure on a remote machine. Even though the figure shows the implicated elements in a RMI method call, the sequence is similar in any RPC-based system. First, the service provider registers itself to some directory service (called *registry* in the case of RMI). From this point on clients can find this service by issuing look-up commands to the registry. Once a suitable service is found, client and service provider can exchange directly.

RPC was very popular and is still widely used, for many reasons. It is built for distributed architectures, that is, it allows projects to embrace such architectures without any hassle. Further, it is built to handle all network related tasks and does so transparently with the help of an IDL leaving the developers with just the core business logic to take care of. In their utilisation, remote procedures are no different from local ones, therefore, developers don't have to learn new programming paradigms, they can continue to write applications as they would normally write them. Another success factor of RPC is its availability for many programming languages. For example, Java calls its RPC implementation Java RMI. Since Java is an object oriented language, calls to methods, whether they are local method or distant ones are handled in an object oriented way. Thus, RMI does a little more than bare RPC but, basically, both are very similar.

3.5. Web Services

The *W3C* describes a Web service as [WEB71]:

> "A Web service is a software system designed to support interoperable machine-to-machine interaction over a network."

Thus, all the required elements to create web services are in place: different processes run on separate machines, allowing for machine-to-machine communication. Furthermore, these machines are connected to some network, which might just be a LAN or equally the Internet. Additionally, some middleware like RPC (Remote Procedure Call) handles the communication between them. Lastly, standards like XDR in the past and eXtensible Markup Language (XML) for modern web services guarantee their interoperability. To fulfill these requirements, middlewares need to introduce some standards to define protocols and select sockets and port numbers, thus increasing the interoperability between services issued from different sources.

At roughly the same time as Java proposed RMI as the solution to remote calls, the OMG introduced its own solution, Common Object Request Broker Architecture (CORBA). Yet, neither of these two approaches could win the competition and a third approach became the standard for web services. The *W3C* definition of a web service also reflects this situation [WEB71]:

> "A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards."

Therefore, a web service has to describe its interface in WSDL (Web Service Description Language) and exchange XML serialized SOAP (Simple Object Access Protocol) messages over HTTP. Clearly, the major components of RPC have influenced this definition. Web Service Description Language (WSDL) is the chosen IDL for web services, UDDI for the directory and naming service and SOAP for the message format. Additionally, SOAP web services can be bound dynamically. This feature is heavily used in business processes and exploited by BPM.

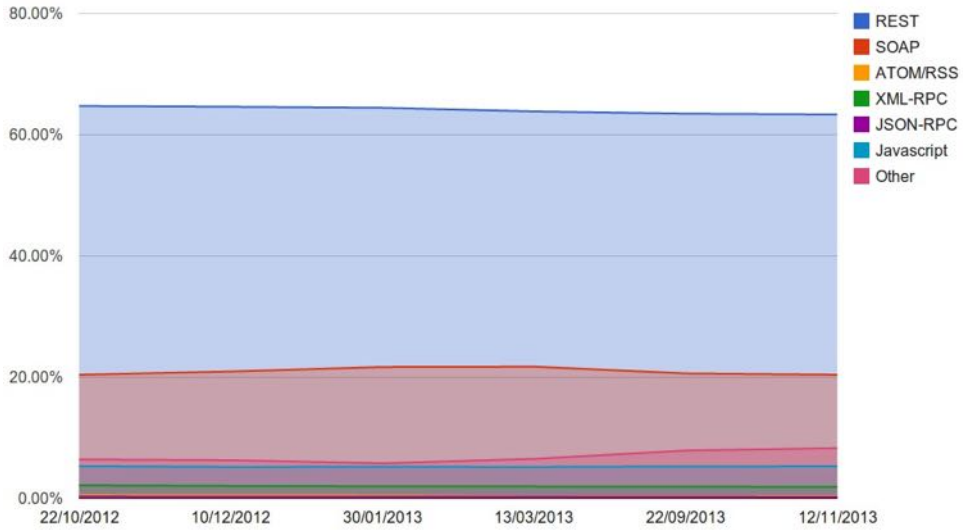


Fig. 3.4.: Market-Share of different web service technologies

For a long time this was the only definition of a web service. Whereas it has the merit of being very precise, it also excludes all other architectures from being called a web service. However, today, several approaches to delivering services over the Internet can be identified. Figure 3.4 shows the evolution of the market-share of selected web services technologies over twelve months [WEB48]. Clearly, the two main competitors in this domain are SOAP and RESTful web services. All others are either derivates of these two or have a negligible market share. RESTful and SOAP in top position is not surprising, and whether REST or SOAP takes the first place depends on the source. Whereas, the programmableweb gives a detailed overview of the current landscape of web services, it only takes into account publicly available services. However, the strength of SOAP web services are business environments where many business process frameworks strongly depend on them. Yet, the programmableweb does not take into consideration private and enterprise web services. Nonetheless, whether private and enterprise web services are considered or not does not change the fact that SOAP and REST are the two most popular families of web services. At least in the public domain, it seems that RESTfull web services are the most popular architecture for designing web services. Over more than a twelve month period, its market-share was always more than 60%. Going back to 2011, its market share was even of 73%. The second biggest family of web services is SOAP [17].

Today, according to programmableweb [WEB48], most web services belong to one of the two big families of web services: (1) WS-* services, following the *W3C*s definition and (2) RESTful and RESTlike web services. However, plenty of other families exist like XML/JSON-RPC or Javascript. All come with their own architecture, tools and conventions. Therefore, a more suitable definition of a web service would be:

Definition 1 (*Web Service*)

> *A web service is a software component supporting interoperable machine-to-machine interactions over a network. Each component is deployed in a supporting container, giving it a life-cycle and ensuring communication with the outer world. It has a well defined API and produces and consumes standardized messages over a well defined protocol.*

While this definition is less specific than the *W3C* one, it has the merit of not favoring any particular technology or architecture. However, it still outlines the 4 most important principles a service must fulfill to be a web services: (1) Addressability and Connectivity, (2) a lifecycle bound to a container, (3) a well defined API and (4) a standardized protocol. The above definition furthermore introduces the concept of components and containers, which will be discussed later (see Section 6). The next two sections illustrate how these principles are applied in the two biggest families of web services.

WS-*

The WS-* family of web services consists of several technologies and architectures to provide a service layer over the Internet, targeted for machine-to-machine communication. In the late 90's, the computer science world was enthusiastic about the web and the new possibilities regarding how to structure applications and provide machine-to-machine services. Since Microsoft's DCOM only worked well in LAN environments, but not the Internet, a new way of delivering services needed to be found.

Dave Winter, developed at that time the XML-RPC (Extensible Markup Language Remote Procedure Call) specification, a new way to refer to methods living on other machines over a network. One of the strengths of Extensible Markup Language Remote Procedure Call (XML-RPC) is the independence of programming languages and operating systems. The specification allows for easy implementation in various programming languages. Over the years, this specification evolved and finally became the WS-* services based on SOAP.

SOAP web services are still widely used principally in enterprise environments and ESB (Enterprise Service Bus). The architectures and the tools evolving in this universe have become so specialized over the past decade that they only understand SOAP. This is, for example, the case of BPM (Business Process Modeling) where the specification (although not explicitly) foresees that a service call is automatically a SOAP call.

Regarding Definition 1, SOAP web services fulfill all four principles. A SOAP web service is addressable. Generally, each SOAP web service has one URL (Uniform Resource Locator) serving as entry point. This URL holds the web service description file (WSDL) and also serves as the target URL for client requests. Since these web services generally don't depend (at least externally) on other web services, their connectivity is very weak. At most, since the WSDL contains links to each available method, this can be interpreted as some sort of connectivity. SOAP web services are generally deployed in a container, for example, services written with Java and C#. Each container has its own method of component deployment and the steps necessary from uploading a component to having the application server routing requests to this component. These steps are commonly called lifecycle and depend on the application server but also on the component to deploy. WSDL ensures that each SOAP service has a well-defined, documented API. This

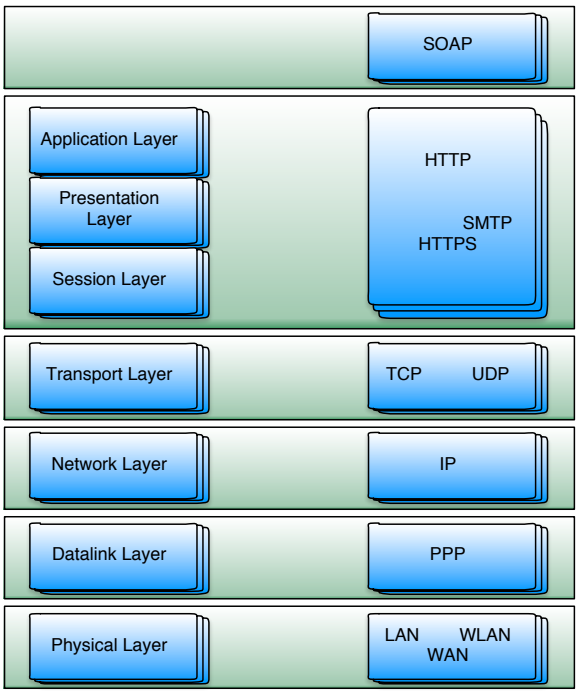


Fig. 3.5.: SOAP in the OSI Stack (modified form [B21])

documentation is sufficient to build simple clients automatically (using SOAP UI, for example). Since this description file is the result of a compilation, it is always up-to-date and reflects the current state of the API. Lastly, SOAP web services use a well-defined protocol to consume web services, called SOAP [17] . This protocol lies on top of HTTP and uses the latter as a transport protocol. It is also this point that most people criticize. In fact, SOAP is an application protocol sitting on top of the protocols composing the application layer (see Figure 3.5), which clearly violates the OSI stack. Additionally, SOAP web services use other protocols for secondary tasks. For example, the web service discovery is handled by UDDI (Universal Description Discovery and Integration) [8].

REST

The second largest family of web services is Representational State Transfer (REST). However, unlike RPC, REST is an architecture not a technology nor an implementation. Its foundations goes back to Roy Fielding's thesis in 2000 [14] where he describes an architectural approach to delivering services over the web. During his Ph.D. studies, he worked together with Tim Berners-Lee and a few other researchers on the HTTP/1.1 [rfc2616] specification. The architectural implications of this specification and a few more reflections are the basis of Fielding's doctorate. The four necessary principles according to Definition 1 are all verified by the architectural constraints enumerated by Fielding. The ins and outs of RESTful web services are an important point in this thesis and discussed in detail in Chapter 5.

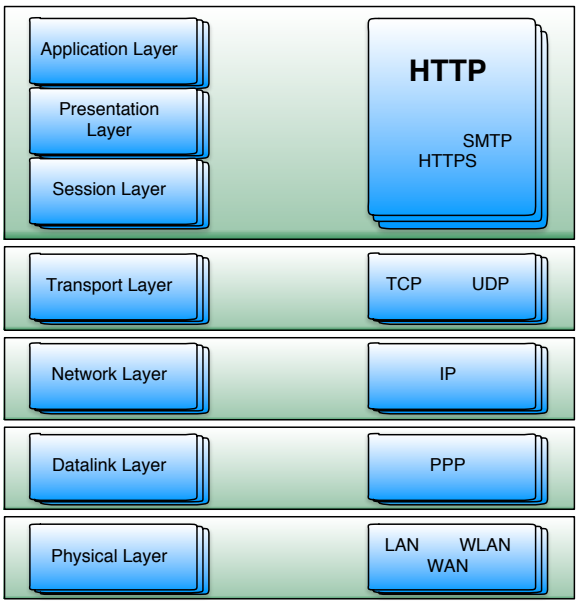


Fig. 3.6.: HTTP in the OSI Stack (modified from [B21])

3.6. Key Concepts introduced in this Chapter

Connecting multiple computers together was the next logical step. These networks, be they ARPANET, the Internet, a LAN or any other network of machines, opened new perspectives. Whereas one of the first needs is personal communication (email, IRC), remote computation quickly becomes just as important. Today there are many different types of services delivered over a web. The most prominent ones are based on the WS-* stack and RESTful (and also RESTlike) services.

Both have their advantages and disadvantages, Pautasso et al. [64] compared lightweight REST services with their big counterpart WS-* and concluded that there is no winner. The best approach to choose should be use-case dependent. Unfortunately, big companies tend to choose "big web services" without asking if there might be a better approach.

RESTful web services are a key architecture for the WoT and the xWoT as will be introduced in Section 8.2. The reasons for useing REST as a requirement for the WoT are manifold. Usually the smart devices composing the WoT are quite simple and don't have much CPU (Central Processing Unit) power, therefore, they usually can't run fully blown WS-* services. Additionally, REST web services are loosely coupled in many aspects [63] and in general, loose coupling is a positive aspect of any architecture and software. REST web services always perform better in terms of coupling, whether in the discovery, the model or the generated code. Finally, REST is built on the success of widely accepted and used standards. The Web became popular because of its properties and the same applies to web services. Whereas many are afraid of consuming a WS-* service and non tech-savvy users are unable to exploit them, everybody can consume a RESTful web service.

4
Evolution of Things

4.1. Introduction

Chapters 2 and 3 describe how the Web evolved through its iteration from the very simple HTTP 0.9 to today. Furthermore, the notion of services over a network and the different approaches to delivering them have been outlined. This chapter focuses on a special kind of service: services for physical objects.

In most offices the digital revolution has already taken place. Writing reports on a computer using a text processing engine, making online searches or simply writing emails to inform co-workers about a new youtube film about cats has become the standard. The digital revolution has also modified large parts of people's lives outside the offices. Today, it is almost more common to communicate over some messaging application (facebook chat, Hangouts, XMPP, Whatsapp to name a few) than telephoning somebody directly. Comparing prices and reading through reviews before buying a new product is normal. Booking flights, hotels and cars online is no longer a dream but a reality [WEB70]. This means that large parts of our lives are already virtual.

However, there is still a big gap between the virtual and the physical world when it comes to everyday duties like shopping, home automation and so on. This gap is mainly due to the simple fact that shopping does not easily translate into a virtual action. The same holds true for home automation. Whereas opening the roller blind might be an easy task for most humans, it is quite complicated to have this task executed by a device. Complicated does not necessary mean that the electronics composing such a device is complicated; it can also mean there is an excessive cost to install a suitable device or designate a noticeable amount of knowledge to use the device.

The smart fridge is maybe the most prominent example of a simple task difficult to carry out in the virtual world. For years, vendors have tried to build a self-refilling fridge based

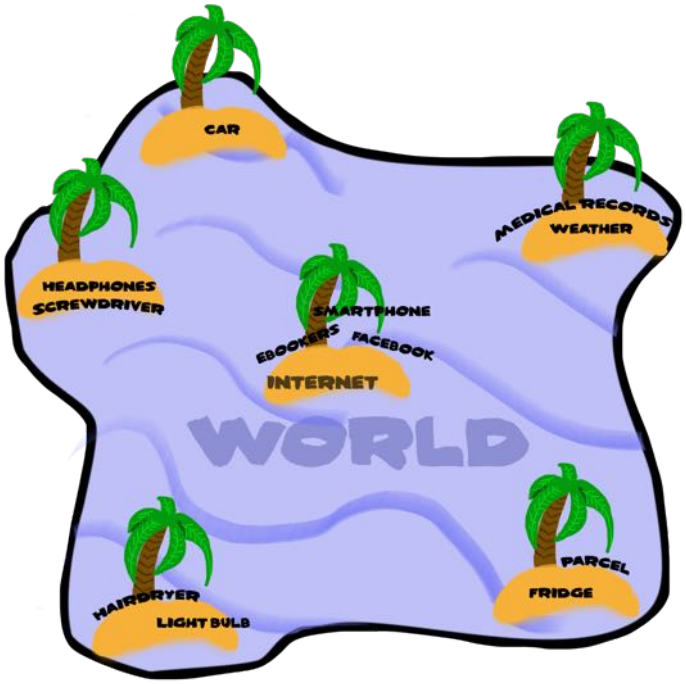


Fig. 4.1.: The world is composed of the Internet plus everything else. Yet, both are disconnected

on several facts like the actual content, user preferences, personalized shopping lists etc. So far, none of the presented systems is really viable to install in a house. The problem here is less the fridge than its surrounding. Shopping centers, for example, would need to provide some sort of communication interface through which this fridge could order new food. Hence, as long as these external dependencies are not met, the fridge will never really work. Figure 4.1 describes this situation: the *INTERNET* island is where services live and can communicate. In this part of the world, everything is connected and can interact with everything else. However, the parts not living on this island are excluded from this communication. There is simply no way of having these things talk to each other

The domain of ubiquitous computing tries to find novel approaches on how to make the computer disappear from everyday life and bring such objects to virtual life. In the past, we shifted from one computer for a city, to one computer for a person (the Personal Computer) to many computers (for a person). Although this last pragma can already be considered true through the abundance of smart phones, tablets, phablets and other portable devices, they are usually not ubiquitous and not specialized for one task. The industry proposes specialized hardware for home automation systems. However, today these systems often evolve in a closed ecosystem making it impossible to integrate new or already established devices into them. This plus the high price tag reserve these systems to only a handful of persons in the world.

The Internet of Things and the Web of Things are trying to come up with new solutions to such problems. To achieve this goal, they take a radically different approach. Instead of focusing on the process (like home automation systems) or one particular object (like the fridge), they enhance any object living in the physical world with a virtual representation which a user can interact with in the virtual world. The physical and virtual world are

linked together so that a change in the physical world is reflected on the virtual side and vice-versa. Following this paradigm still does not solve the smart fridge problem discussed earlier but it allows the building of millions of smart devices delivering information to the world. Therefore the IoT and the WoT is a twofold vision where things not only have a physical manifestation but also an equivalent virtual one.

Definition 2 (*IoT - WoT*)

The IoT and the WoT both enhance physical objects by adding a virtual counterpart. These enhanced objects become smart objects and are also called Things. From this point on, it does not matter whether the Thing is manipulated physically or virtually, both have the same result.

In the vision of the IoT and WoT, millions of smart devices enhance physical objects and make them available to the virtual world. It it is then possible to take advantage of the delivered capabilities to build novel applications at almost no cost. Already today many offices have electronic roller blinds. However, implementing a mechanism where they open and close depending on several factors like the ambient luminosity outdoors, date and time, weather and so on, would be quite cumbersome and very expensive. Additionally, it would be quite challenging to adapt such a system to changing needs. Here the IoT/WoT can help out. Instead of building custom software and closed systems, each thing in the physical world becomes smart. Implementing the automatic roller blinds can then be done with only a few web services calls.

Several sources state that the number of smart phones outnumber humans [WEB35, WEB63, WEB62]. The same statement will become true in a not too distant future for smart devices taking an active part in the web and exchanging data. Most of the time, these smart devices have only limited capabilities (be they battery life, bandwidth, CPU cycles etc...) influencing their implementation and behaviour. Regardless of these limitations, smart devices need to be small and robust. Whereas this is quite easily achieved for the electronic part, it is already more difficult for the API they are providing. This chapter discusses two approaches to building smart devices. The first one, the IoT is historically interesting as it is the foundation of any smart device. The second approach, the WoT is interesting for the scope of this thesis.

4.2. The Internet of Things

As the name suggests and as discussed in the introduction, the Internet of Things is an Internet where Things play a major role. However, this does not indicate which internet is meant nor which objects can participate. Commonly, it is agreed that although there are several separate networks, sometimes joined together, by *Internet* we always mean the "Network of Networks". Therefore, the IoT is the same Internet as the one we use every day to write emails and book flights. The *Things* in IoT designate any physical manifestation. Therefore, anything can be a Thing, a book, a chair, a car, the weather, the temperature, etc. The aim of the IoT is to provide these Things living in the physical world with a virtual counterpart and thus embed them into the virtual world.

Back in 1999, Kevin Ashton mentioned the term Internet of Things for the first time in a presentation about supply chain management at P&G [WEB59]. At that time, tagging objects and following them through a whole process was a brand new approach in supply

chain management. The automobile industry in Japan, for example, invented Quick-Response codes and attached them to each part of a car. These allowed machines to assist the workers during the assembly of cars and also introduced some sort of quality control to ensure that all the required parts are attached together. Another fundamental technology to support this vision of supply chain management is RFID tags. These passive electronic circuits can store a few bytes (usually an ID, but it might be any string, like an URI). With the emitted energy of the reader, the RFID tag has sufficient energy to send back the content it holds. Whereas both approaches are still used today, QR (Quick-Response) codes have gained momentum over the past few years, principally through mobile phones, where they are a convenient way of sharing information. In the consumer world, RFID tags are almost non-existent. Their fabrication, and also the necessary infrastructure to read them, is simply too expensive. Another similar technology gained popularity in the consumer world: Near Field Communication (NFC). Nonetheless, RFID is still widely used (for example, in the clothing industry to prevent theft and identify counterfeits).

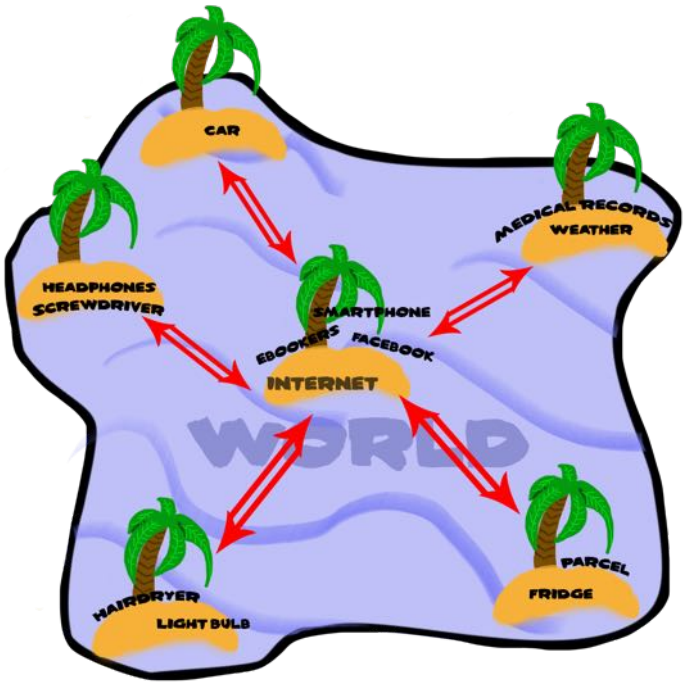


Fig. 4.2.: The IoT world, where everything is connected

These RFID tags, QR-codes, NFC (Near Field Communication) tags and the underlying idea of identifying objects are the foundation of the IoT as we know it today. Based on the work presented at P&G, Kevin Ashton founded the Auto-ID center at MIT (Massachusetts Institute of Technology) in 1999, later renamed the Auto-ID labs, and now has 6 labs around the world. This also marked the birth of the IoT. In the beginning, the Auto-ID center researched the implications of technologies like RFID and objects tagging for supply chain management. The goal was to propose a standard for how objects should be tagged. The Electronic Product Code (EPC) and its supporting infrastructure EPCIS (Electronics Product Code Information Services) , later ratified by the GS1[1], a non-profit organization active in supply chain management, is the result of these efforts. EPC (Electronic Product Code) and the EPCIS are also the most notable outcomes of

[1]http://www.gs1.org/

the early period of the Auto-ID center. By adding a tag to an object, it is possible to identify it. In its simplest form, the tag only contains a unique identification. Upon entering this ID into a system, the user can gain some (virtual) information about the physical product. This is also the approach chosen by the EPC systems. Whenever a tagged product enters or leaves a control point in the supply chain, a reader scans it and saves this event in the global EPC database. In the end, when the client finally holds the product in his hands, he can check where in the world this product has been simply by scanning its tag.

Since that time, the Auto-ID labs have evolved and are now an active promoter of the IoT. Amongst others, the Auto-ID labs organises a bi-annual gathering for researchers interested in this topic. Yet, the IoT has never managed to lose this image of RFID tracking architecture. Still, this early tracking did introduce the notion of sensing and opened the door to other applications. Instead of just identifying objects and tracking them, the sensor itself can be of interest. In the case of a temperature sensor, the same reasoning applies to actuators enabling devices to act on their physical environment. Enabling things to sense and manipulate their environment is the first step to building smart devices. According to Gérald Santucci [35], such smart devices are not only meant to interact with humans but also — and more likely — to interact with each other. The IoT is a prolific research topic of research around the world. Europe is interested in the IoT and has started several extensive research projects like IoT-A [WEB23] . The European project "Internet of Things Initiative" has released, in the form of a comic book, a collection of example use cases which could benefit from smart devices and the IoT.

Fig. 4.3.: Different commercial IoT products

Today there is already a large palette of IoT consumer products and there will be even more in 2015. The IoT was one of the hot topics of this year's CES edition in Las Vegas [WEB60]. Many vendors presented their solutions and products for the IoT. Home automation seems one pushing factor for the IoT. Google has its Nest[2], Apple has HomeKit[3], Samsung has tried to start their eco-systems with smart TVs and so on. Regarding the number of available products, connected light bulbs seem to be for many companies the entry point to the IoT universe. Philips, for example, proposed with Hue, a whole ecosystems of connected lamps, light bulbs and televisions. Whereas this

[2]https://nest.com/
[3]https://developer.apple.com/homekit/

trend is not new, what has changed from previous years is the openness of the presented solutions. Vendors understand that the DIY (Do-it-yourself) community can make a product a success. Therefore, vendors have shifted from a closed ecosystem to an open one, giving the users the power to adapt and tweak the system.

The IoT is similar to the way the Internet works; the difference lies in the participants. Whereas the participants of the classic Internet are humans, participants of the IoT are smart devices, also called Things. Regarding the way two parties communicate, the IoT does not give any guidelines, so, two participants can use just any communication protocol. Being indifferent to the way a smart device communicates allows users to choose the most suitable one for each situation. If power consumption, for example, is a considerable problem, adapted protocols like ZigBee, 6LoWPAN [34, 7] or CoAP [75, rfc7252] can be used. On the one hand, this approach allows a maximum of flexibility and also opens the door for WSN (Wireless Sensor Networks). On the other hand, it leads to the fragmentation of the different participants and incompatibilities between them.

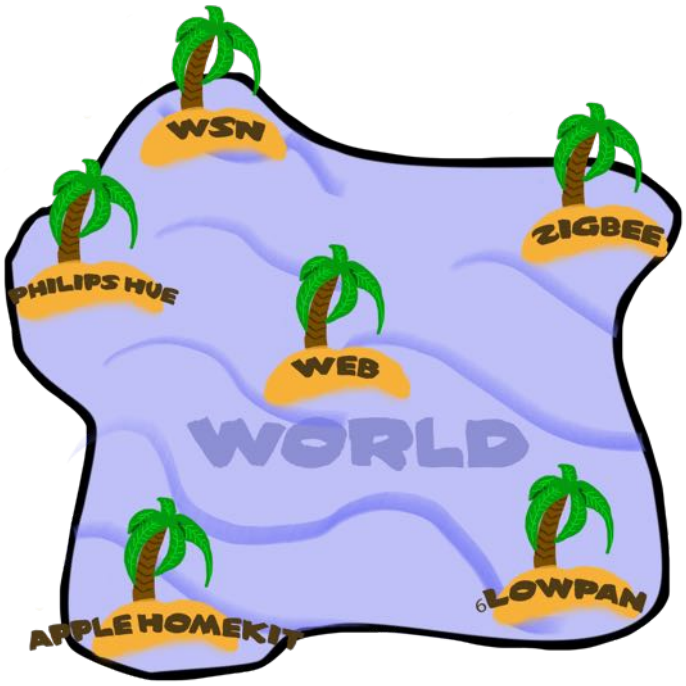


Fig. 4.4.: Fragmentation as a result of the no standards approach

Figures 4.4 and 4.3 illustrate the biggest problem of the IoT. Each vendor mandates his own approach on how smart devices are connected and communicate with each other. Each approach considered in complete isolation seems to work will, although some approaches work better than others. However, on a global, vendor independent scale, the problem becomes clear: smart devices from vendor *A* are not compatible with smart devices from vendor *B*. This leads to several disconnected islands and many frustrated consumers. Such a fragmented situation is not new in computer science (see wireless charging standards for example) but fragmentation is also a challenge for vendors of home automation boxes (like those from Vera [WEB68]). It is almost impossible to create a box that is compatible with 99% of smart devices at an attractive price. Additionally, these boxes rapidly become obsolete since they would need additional or upgraded hardware to keep up with new products hitting the market. Generally, such fragmented situations

are bad and consumer are not willing to invest in one system or another due to concerns about making a bad choice.

4.3. The Web of Things

The WoT is similar to the IoT in the sense that both deal with connected smart devices. In the news media, these terms are many times mixed up and references to the IoT often mean the WoT. This is understandable, as the differences between the two might seem small at first glance. Yet, their implications are big enough to allow the WoT and the IoT to be told apart. Compared to the latter, the WoT takes a radically different approach how devices communicate with each other and imposes some standards. These constraints are necessary to address and solve the problems facing the IoT. In order to allow the disconnected islands of Figure 4.4 to communicate, the Web of Things mandates for RESTful web services. This means that regardless of the kind of smart device or its physical limitations, to participate in the WoT it has to offer its API over a RESTful web service. This new situation is depicted in Figure 4.5 where the different smart devices of Figure 4.4 are now able to communicate.

Definition 3 (*WoT*)

The WoT enhances physical objects with a virtual counterpart representing the latter in the virtual world. In contrast with the IoT, the WoT mandates the strict application of RESTful principles to its APIs.

Fig. 4.5.: Heterogeneous interactions in the Web of Things

The obvious advantage of this approach is that WoT smart devices are embedded into the Web like any other webpage, therefore, they benefit from the information and services already available on the Web. When a user browses the Web he is unable to tell whether

the current response comes from a smart device or an ordinary webpage. Although different in their inner guts, they appear the same to the user. As a side effect, this also brings devices from different vendors to a common denominator and allows them to seamlessly exchange information. Since the whole Web content is accessible to smart devices, having a connected toaster posting messages on twitter whenever the toast is done becomes an easy task. Almost no special knowledge is needed to create such mashup applications. Pautasso et al. [64] discuss the advantages of RESTful web services over the traditional WS-* and conclude that both have their advantages. However, in the field of constrained smart devices, REST is certainly the better choice of two.

Mashup is the art of mixing different sources together to create new content. Mashups exists in the music and video industry but also for books, as they do in the digital world. Yahoo released the Pipes[4] editor, a graphical frontend to blend information from different sources in 2007, as explained in Figure 4.6. The underlying idea is to allow users to create websites tailored to their interests by mixing the content of different RSS (Really Simple Syndication) feeds. This idea is still valid for the WoT. Although, raw sensors and actuators are the building blocks of the WoT, taken in isolation they don't bring any real added value for customers expect in combination with some logic do they create appealing scenarios. A weather forecast service mashed up with a smart alarm clock, for example, can wake the user up 10 minutes earlier if the temperature is below 0° C. Although, such scenarios appeal to clients, the real added value of the WoT lies not in the scenario but in the ease of creating and adapting them to the needs of every customer.

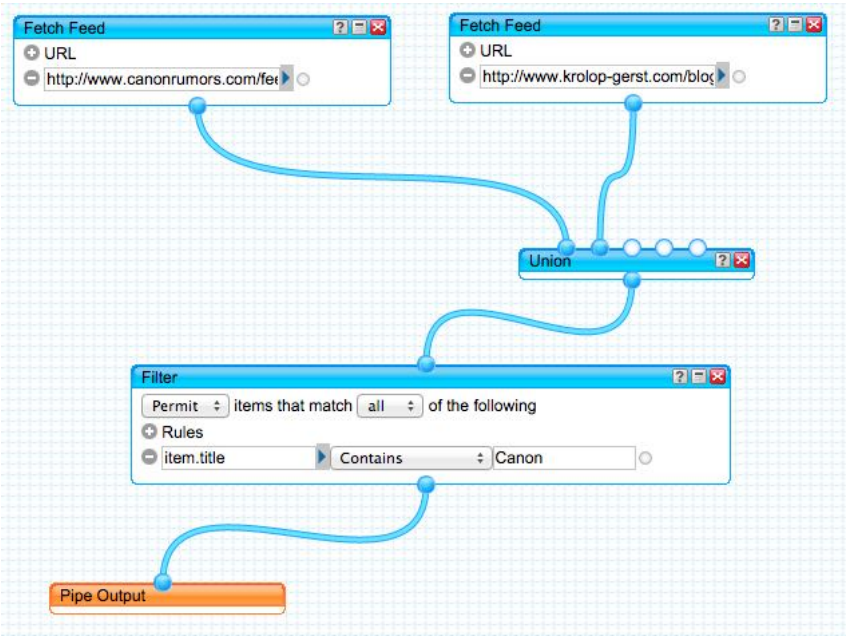


Fig. 4.6.: Yahoo Pipes example flow

The ecosystem of Philips Hue smart lightbulbs is also a great success story of the WoT. Although the individual smart lightbulb communicates over a non-disclosed ZigBee protocol and is therefore outside the scope of the WoT, Philips was smart enough to provide a gateway offering a RESTful façade to communicate with the lightbulbs. This makes the product very appealing to users and many projects are built on top of Philips Hue.

[4]http://pipes.yahoo.com

These projects range from alarm clocks simulating the sunrise [WEB46] to immersive multimedia systems [WEB45]. These projects show that: (1) there is a huge interest in smart devices and (2) users are not necessarily power-users. The WoT also allows new players to enter the game. The Koubachi plant sensor [WEB29] for example, started as a research project at the ETH Zürich and is now an internal company selling WoT compliant plant sensor. Although, it is not always advertised, vendors of smart device solutions have seized the advantages offered by RESTful APIs and often provide such an interface for their products.

Considering the reference implementation of REST for Java, JAX-RS [WEB26], it is quite easy to create RESTful APIs. Take the example of a smart thermometer. On the physical side, the thermometer is composed of a thermistor sensor like a DHT11 measuring the ambient temperature but also the humidity. Furthermore, assuming that there is a class `Thermistor.java` responsible for talking to the raw hardware, then it is just a matter of having one other Java class to create a RESTful interface and embed this thermometer into the WoT. Listing 4.1 shows that it is a matter of a few Java functions, each one handling one type of HTTP request. Additionally, the listing shows three defined REST resources: the first `hvac` represents the smart thermometer (lines 8 and 12-22), whereas the second and the third represent the temperature (lines 24-30) and the humidity (lines 32-38) readings respectively.

```
1  package ch.unifr.diuf.hvac.service;
2
3  import javax.ws.rs.GET;
4  import javax.ws.rs.Path;
5  import javax.ws.rs.Produces;
6  import javax.ws.rs.core.Response;
7
8  @Path("hvac")
9  public class ThermistorResource {
10     private static Thermistor dht = new Thermistor();
11
12     @GET
13     @Produces({"application/xml", "application/json", "text/xml"})
14     public Response getHVACInformation() {
15         ...
16     }
17
18     @GET
19     @Produces({"text/html"})
20     public Response getHVACInformationHTML() {
21         ....
22     }
23
24     @Path("temperature")
25     @GET
26     @Produces({"application/xml", "application/json", "text/xml"})
27     public Response getTemperature() {
28         int temperature = dht.getTemperature();
29         ...
30     }
31
32     @Path("humidity")
33     @GET
34     @Produces({"application/xml", "application/json", "text/xml"})
```

```
35     public Response getHumidity() {
36         int humidity = dht.getHumidity();
37         ....
38     }
39 }
```

List. 4.1: RESTful interface for a smart thermometer

What is true for the creation of RESTful web services is also true for their consumption. Whereas for any other type of service, the corresponding client infrastructure is necessary to use it, users already have the necessary tools to consume RESTful web services installed on almost any modern computer (PC, smartphones, tablets etc.). The easiest way to consume a RESTful web service, and thus communicate with WoT smart devices, is by using a browser. For most gadgets living in the WoT, almost any browser will fit as long as it supports HTTP/1.1. Figure 4.7 shows what a response from the code snippet of Listing 4.1 could look like. Upon requesting the top level resource, the user gets a nice looking HTML page containing some information about the smart device including the sensor readings. This shows that browsing smart devices is roughly the same as browsing any other web-page and therefore easy. However, since smart devices are often built for machine-to-machine interactions, the results can vary. Often, a smart device just sends back some raw XML or JSON data and no fancy HTML form to manipulate the device. Additionally, these interactions are mostly limited to *GET* information.

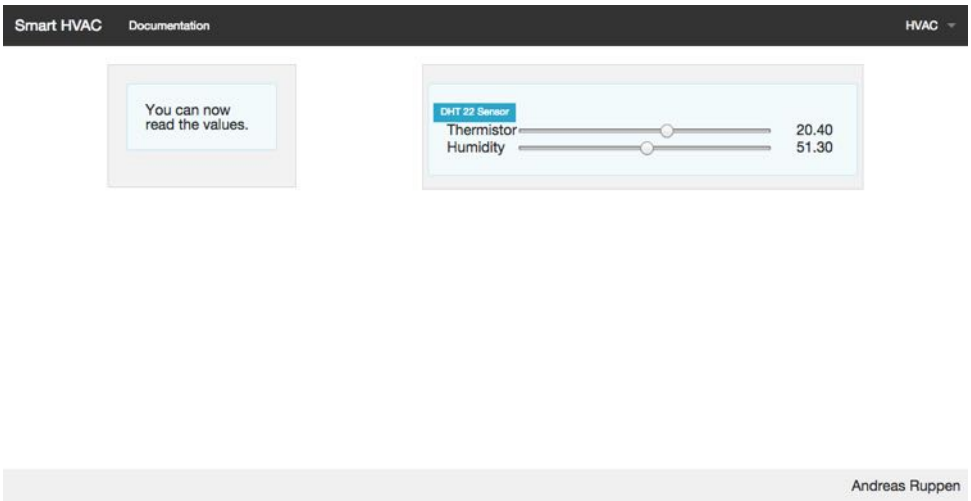


Fig. 4.7.: Browsing the smart thermometer with Google Chrome

To send information back to the server with a browser (for example, to change the state of an actuator), the requested HTML pages have to include forms. Whether this is the case or not, depends on the actual smart device. Yet, the HTML representations are often quite basic and don't include any fancy parts like HTML forms. One way to overcome this limitation is to use browser plugins like *Advanced REST client* for the Google Chrome browser or to rely on common command line tools `cURL`. These tools are available on most platforms and have been around for a long time (1997 for cURL). Choosing one over the other is often a question of personal taste (see emacs vs. vi). Whereas the Advanced REST client comes as a nice looking GUI (Graphical User Interface) with many features, cURL is only a command line tool. Both have their advantages and depending on the situation, one tool might be easier to use than the other. Listing 4.2 shows on

lines 1 and 2 a `GET` request to retrieve the same resource as on Figure 4.7. The only difference lies in the fact that cURL retrieves a JSON representation instead of a HTML webpage. Additionally, in lines 3 and 4, Listing 4.2 also shows an example of an actuator manipulation. With a `PUT`, request a smart socket is switched off. In both cases, the server responds with a representation of the requested/modified resource. Furthermore, the reader can note that the command issued on line 3 has an effect on the smart device and therefore also on the representation. Whereas on line 2, the `electricity` attribute is `1`, meaning that the socket is switched on, the same attribute is `0` on line 4 after the `PUT` request.

```
1 ruppena@tungdil:~: curl -H "Accept: application/json" http://192.168.1.10:9000/hvac/
2 {"temperature": {"@units": "celsisus","@precision": "2","#text": "21.70"},"humidity":
    ↪ {"@units": "celsisus","@precision": "2","#text": "49.70"}, "electricity":"1", "
    ↪ timestamp": "1420793672"}
3 ruppena@tungdil:~: curl -H "Accept: application/json" -H "Content-Type: application/
    ↪ json" -X PUT -d '{"electricity":"0"}' http://192.168.1.10:9000/hvac/
4 {"temperature": {"@units": "celsisus","@precision": "2","#text": "21.70"},"humidity":
    ↪ {"@units": "celsisus","@precision": "2","#text": "49.50"}, "electricity":"0", "
    ↪ timestamp": "1420793751"}
```

List. 4.2: Using cURL to interact with a REST service

4.4. Key Concepts introduced in this Chapter

The evolution of objects towards smart objects is a reality. Ubiquitous systems, home automation, self-driving cars are only a few examples where the technological advances have been huge in the past few years. However, these technologies remain expensive and not ready for mass consumption. Although many car vendors have a working prototype for a self-driving car (Google is still a pioneer in this domain), none are generally available. The same applies to home automation and other ubiquitous systems. They are often monolithic and need to be permanently installed by professionals. This is definitely only an option for (rich) property owners. Yet, the IoT and the WoT make these fields accessible for the masses. The WoT, in particular with its mandate for a RESTful interface makes it very easy to combine smart devices from different sources and vendors and create novel and innovative applications.

Whereas in the beginning the IoT was more about supply chain management and tagging objects with RFID tags or QR codes, it has since evolved and its primary application is today in the field of actuators and sensors making ordinary objects smart. From this perspective, the WoT diverged from the IoT and imposed its own standards, which are adopted here for the remainder of this thesis. Nowadays, even big companies introduce products ready for the WoT (Philips Hue [WEB44], Koubachi Plant Sensor [WEB29], Apple's HomeKit [WEB20]) and use it as a marketing tool.

Platforms like Arduino [WEB5], the Raspberry Pi [WEB49] and their corresponding community websites offer the necessary tools and knowledge for advanced users to build their own smart devices and thus their own vision of the WoT. Sometimes, prototype projects based on these platforms can succeed through fundraising to develop a mass-

market ready product. It seems this is just the beginning; 2015 will be the year of the IoT, where everything will be connected (even pacifiers can become smart [WEB41]).

Although the WoT and RESTful web services simplify the integration of smart devices issued from different sources, vendors are still challenged when it comes to designing such smart devices. Compared to the IoT, the WoT already restricts certain design choices, but there is still a high degree of flexibility when it comes to design a smart device's RESTful API. One of the core contributions of this thesis is a meta-model to help smart device creators design these APIs. In comparison with other models and reference architectures for smart devices, the xWoT meta-model can work in a very efficient way since its outcome is a REST hierarchy representing the physical device.

Part II.

Theoretical Background

Foreword

The aim of this thesis is to propose a way to *automatically* generate WoT *components* living in the *Internet* with a *RESTful* interface.

Model Driven Architecture is a popular approach in software engineering to automatically generate code from models. Starting with a model, generators are able to create project skeletons which can already implement some core functionality or at least presenting the core structure. However, these generators are not transcendent; they can only work with a given number or a family of models. A common way to create models respecting guidelines and sharing some properties is meta-modeling. Therefore, to accomplish the goal of *automatically* generating WoT components it is necessary to introduce a meta-model for the latter (and more precisely, for the xWoT as mentioned in Chapter 8). This meta-model guides the creation of models for any imaginable scenario within the scope of the xWoT.

The WoT, although homogeneous regarding the outer interface (REST) is heterogeneous regarding the deployment of the different parts composing it. Introducing a *component-based architecture* gives these different parts a common structure. Additionally, this component structure later allow different basic components to combine and form new and bigger components. Thus, a scenario managing all smart light bulbs in a house can be built on top of a scenario for one particular light bulb. This visions, introduces the notion of deployable and reusable components which can be combined into new components.

The outer interface of any of these components is RESTful. This is a requirement of the WoT and at the same time greatly simplifies the semi-automatic component generation from xWoT models. Yet, to define the common structures of this outer interface in the meta-model and to accurately translate into code in a later step requires a deep understanding of the underlying principles is necessary.

This part covers the three above mentioned points and discusses them in detail. Chapter 5 introduces the concepts of REST, RESTful architectures and Resource Oriented Architectures (ROAs). Chapter 6 introduces the notion of software component as defined in the literature. The work presented in this chapter serves as a basis for the xWoT components. Finally, Chapter 7 formally introduces meta-models and meta-modeling from two different approaches: 1. with the semiotic triangle and 2. with languages.

5

REST and Resource Oriented Architecture

5.1. Introduction

In contrast to the IoT, the Web of Things imposes REST as an architectural style for all smart devices. According to Guinard [19] this permits the easy combination of smart devices produced by different vendors. Thus, designing WoT compatible smart devices always requires a deep understanding of REST and the underlying HTTP. REST is a successful choice when designing new interfaces. Yet, not all interfaces that claim to be REST are truly REST. This merely comes from the fact that not all agree on the scope of terms like REST, RESTful and ROA.

This chapter introduces each of these terms accurately and within their contexts. The first part, discusses HTTP, its evolution and how it works. This also covers a brief introduction to the different authentication mechanisms. In the second part, this chapter then defines a RESTful architecture according to R. Fielding. Based on these definitions,

this section also gives some insights into how others interpret the REST architectural style. Finally, the third part introduces Resource Oriented Architecture, Richardson and Ruby's view of the REST architectural style applied as to services.

5.2. HTTP

HTTP (Hypertext Transfer Protocol) is a protocol to exchange data over the Internet. It is based on a client-server architecture, where the client requests content and the server delivers it. This is also called the request/response paradigm. Commonly, the combination of the Internet with the HTTP protocol is called the Web. The term Web is commonly used to designate that something uses HTTP as a fully fledged application protocol as is the case for the WoT, which makes it different from the IoT. Whereas the latter connects Things over the Internet without imposing any protocols, the former imposes HTTP as its underlying application protocol. HTTP is mostly used by web-browsers to fetch content (mostly HTML, CSS (Cascading Style Sheet) , Javascript and Images) from remote servers and display it. With respect to the OSI network layers [B21, 36], HTTP sits in the top-most layer and so, is considered as an *application*, although some applications miss-use it as a simple *transport* protocol (see Section 3.5).

By its nature, HTTP is a stateless protocol. If an application on top of HTTP needs to track a state, like that of a shopping cart in e-commerce applications, then it needs to implement its own state tracking mechanism. Today this is commonly done by storing a long unique identifier as a cookie in the browser identifying a session on the server. Using this mechanism, the application logic can uniquely assign each request to a session. Nonetheless, sessions greatly increase the complexity of applications and the supporting infrastructure. Whereas simple HTTP applications do scale up well, load-balancers need additional application-level logic when the applications implement some (application)-state (see also Section 5.4.2).

5.2.1. Evolution of HTTP

HTTP/0.9

HTTP was first introduced by Tim Berners-Lee, Roy Fielding and other researchers at CERN in 1991 as set of protocols for the WWW (World Wide Web) project started two years earlier. This set of protocols also contained the first version of HTML and is known as HTTP/0.9. In the beginning, only simple text documents could be transmitted since any other content would have taken too much time to transmit. Additionally, the only implemented action is `GET`, allowing a client to retrieve documents from a server. To request a document, the client first opens a connection with the server. For now it does not matter whether this connection is Transmission Control Protocol (TCP) or User Datagram Protocol (UDP). HTTP/0.9 is so simple it can handle both types of protocol. Once the connection is open (line 1 to 3 of Listing 5.1), the client sends a document like the one in Listing 5.1 (lines 5 to 6). The document contains the keyword *GET* followed by the URI (Unified Resource Identifier) of the requested document. The server responds by sending back the requested document (line 7).

```
ruppena@tungdil:~$ telnet www.example.com 80
Trying XXX.XXX.XXX.XX...
Connected to www.example.com.
Escape character is '^]'.
GET www.example.com

Hello World!
```

List. 5.1: HTTP/0.9 GET request

Since `GET` is the only valid method, it is also used to send data to the server. Listing 5.2 shows an example where the string "Andreas" is assigned to the variable "name" and sent to the server. Parameters are sent as key-value pairs appended to the URI.

```
ruppena@tungdil:~$ telnet www.example.com 80
Trying XXX.XXX.XXX.XX...
Connected to www.example.com.
Escape character is '^]'.
GET www.example.com/hello_world.txt?name=Andreas

Hello Andreas
```

List. 5.2: HTTP/0.9 GET request sending data

Such key-value pairs are separated from the rest of the URI by the `?` sign. Even though this approach is not suitable for sending large amounts of data to the server, it is still widely used to further specify what a client is interested in. Therefore, this approach is often used for pagination mechanisms, where the client specifies which page he needs. In addition, some web-sites, like Google, use it to submit the user input in the search form which, makes it possible to bookmark and share given requests. Figure 5.1 shows the key elements of an URI as defined in [rfc3986] where these key-value pairs are called *query*. This key-value approach is still valid today.

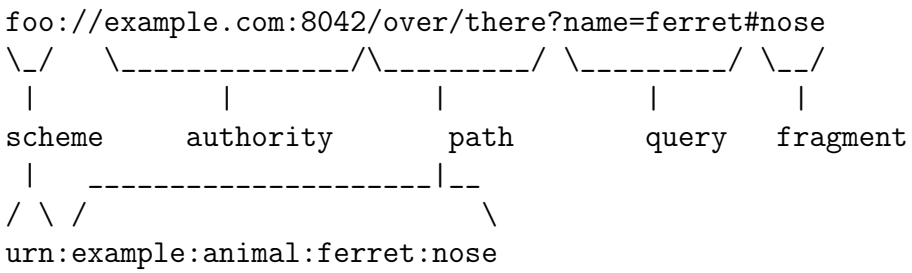


Fig. 5.1.: URI definition (after [rfc3986, p. 15])

HTTP/1.0

After the publication of HTTP/0.9, five more years and several long discussions passed before HTTP/1.0 [rfc1945] was completed and released in 1996. It is also the first officially released version of HTTP and HTTP/0.9 got its version number only after the 1.0 version was released. HTTP/1.0 is a big improvement over its predecessor. Whereas HTTP/0.9

was merely a proof-of-concept implementation, its successor, HTTP/1.0, could already implement all the basic properties of the Web as we know it today. The biggest difference between the two is how documents are requested from the server. For HTTP/0.9 it is sufficient to open a connection with the server and send a one line statement to the latter, which then responds accordingly (see Listing 5.1). HTTP/1.0 keeps the same mechanism, however, the request contains more detail.

Listing 5.3 shows how a user requests a document. The first line still starts with a *verb* indicating what action a client intends to execute. However, instead of only accepting `GET` requests, HTTP/1.0 introduces two additional methods. The `POST` request indicates that the request contains a body which is to be saved by the server, and the `HEAD` request asks for the retrieval of meta-information about the document, which the server delivers without retrieving the actual document. Additionally, this first line is terminated with the string `HTTP/1.0` to distinguish between the old HTTP/0.9 style requests and the newer ones. Furthermore, the request contains an optional header section where the client can specify preferences regarding server behavior. It is important to note that this section is optional and that, even when specified, a server does not have to follow these preferences and can just ignore them.

```
1  ruppena@tungdil:~$ telnet www.example.com 80
2  Trying XXX.XXX.XXX.XX...
3  Connected to www.example.com.
4  Escape character is '^]'.
5  GET hello.txt HTTP/1.0
6  User-Agent: Mozilla/3.0 (X11; I; AIX 2)
7
8  HTTP/1.0 200 OK
9  Date: Thu, 14 Nov 2013 14:27:56 GMT
10 Server: CERN/3.0 libwww/2.17
11 Content-Type: text/html; charset=ISO-8859-1
12
13 Hello World!
```

List. 5.3: HTTP/1.0 GET request

Compared to HTTP/0.9, the server's response also contains more information. Instead of sending back the raw document, the response first contains the string *HTTP/1.0* plus a status code. The string is interpreted by browsers to identify the protocol version used by the server. The status code indicates whether or not a problem has occurred during the request. After this first line, the server sends its response-headers (lines 9 to 11). They contain information about the server and information about the content to follow. Lastly, the server sends the requested document (line 13), which is then interpreted by the client.

HTTP/1.1

In 1999, after another 3 years of successful HTTP usage, the research group around Roy Fielding and Tim Berners-Lee, released the HTTP/1.1 specification [rfc2616]. This signaled another milestone in the success story of the protocol. Whereas HTTP/0.9 was merely a draft and HTTP/1.0 the first really usable version, with HTTP/1.1, the protocol

was ready for modern web applications. HTTP/1.1 still uses the same mechanism for requesting a document: after a connection is established between the client and the server, the client tells the server which document he is interested in and further specifies some headers. In the same way, the server sends back some headers containing meta-information about the exchange and documents before sending back the latter.

Listing 5.4 shows what a full HTTP/1.1 request looks like. Again, a handful of new headers were introduced, compared to the previous version of the protocol. For example, the 6th line of Listing 5.4 shows the new *Host* header telling the server on which host the requested document is available. With the growth of the web, the number of services has exploded. Not every service requires a full server on its own; several applications can share the same physical machine. This implies that they also share the same IP address. Thus, web-servers need a mechanism beyond IP addresses to decide to which application a given request is to be routed.

```
1  ruppena@tungdil:~$ telnet www.example.com 80
2  Trying XXX.XXX.XXX.XX...
3  Connected to www.example.com.
4  Escape character is '^]'.
5  GET / HTTP/1.1
6  Host: www.example.com
7  User-Agent: Mozilla/5.0 (X11; Linux x86_64)

9  HTTP/1.1 200 OK
10 Date: Wed, 20 Nov 2013 14:08:02 GMT
11 Server: Apache/2.2.22 (Ubuntu)
12 Last-Modified: Tue, 29 Jan 2013 11:54:41 GMT
13 ETag: "141e70-b1-4d46c0f0f31a2"
14 Accept-Ranges: bytes
15 Content-Length: 177
16 Vary: Accept-Encoding
17 Content-Type: text/html

19 <html><body><h1>It works!</h1>
20 <p>This is the default web page for this server.</p>
21 <p>The web server software is running but no content has been added, yet.</p>
22 </body></html>
```

List. 5.4: HTTP/1.1 GET request

Most of the other new headers control the cache behavior of the client. Listing 5.4 contains the request-headers on lines 6 and 7 and the server headers on lines 10 to 18. Whereas HTTP/1.0 defines a cache as either valid or not, with HTTP/1.1 it can be *fresh* or *stale*. When a cached value reaches its end-of-life, it no longer needs to be discarded. Instead, it can be refreshed from the server and become *fresh* again. Additionally, clients can override this rule and still use stale values. Another mechanisms to further improve cache utilization is the server header *ETag* combined with the request header *If-None-Match*. If the client already has a copy of a document, but is not sure whether it was modified since his last visit he can ask the server to respond with the document only if it has been modified. In earlier versions of HTTP, the only means to achieve the same effect was by using the *If-Modified-Since* header. Whereas this works reliably, handling timestamps is not very user friendly. First, clients have to remember the time at which the document was

requested and second, all the clocks needs to be completely synchronized. To circumvent these problems, HTTP/1.1 introduced the *If-None-Match* header as shown in line 7 of Listing 5.5. Upon requesting a document, one of the server headers is the *ETag* (visible in Listing 5.4 on line 13). This tag identifies the current version of the document. A client can use this information when requesting the same document again. If he specifies the *If-None-Match* header with the value contained in the *ETag* of a previous request (Listin 5.4 line 13) then, the server will only send back the document if it has been modified since. In this case, the document was not modified and thus, as depicted in Listing 5.5, the server does not send back the document. Additionally, HTTP/1.1 introduced some new headers to facilitate content negotiation regarding a client's preferred languages, encodings etc (*accept-language*, *accept-encoding*, *accept-**).

```
1  ruppena@tungdil:~$ telnet www.example.com 80
2  Trying XXX.XXX.XXX.XX...
3  Connected to www.example.com.
4  Escape character is '^]'.
5  GET / HTTP/1.1
6  Host: www.example.com
7  If-None-Match: "141e70-b1-4d46c0f0f31a2"

9  HTTP/1.1 304 Not Modified
10 Date: Wed, 20 Nov 2013 14:10:37 GMT
11 Server: Apache/2.2.22 (Ubuntu)
12 ETag: "141e70-b1-4d46c0f0f31a2"
13 Vary: Accept-Encoding
```

List. 5.5: HTTP/1.1 ETag usage

Another big improvement over its predecessors is the re-use of an open TCP (Transmission Control Protocol) connection. In previous versions, the connection was closed at the end of each request meaning, that before each request, a new connection needed to be established. This was appropriate in the early days when the web was only used to transmit small, static text documents. However, web-pages became bigger and started to include images, javascripts and other embedded content included by links to the corresponding source, meaning that a client could fetch them by issuing a `GET` request. Therefore, if a web-page has a lot of embedded content, before HTTP/1.1 the client had to open a new TCP connection for each element. HTTP/1.1 allows the re-use of an open TCP connection, which drastically improves the overall performance.

Already HTTP/1.0 had defined more methods than its predecessor. This is also true, of HTTP/1.1. Whereas, the motivation to introduce the `POST` method in HTTP/1.0 was the lack of a proper definition of how to send data from a client to the server, HTTP/1.1 introduced a real discussion about what types of exchange are possible and how they should be handled. The methods are classified into *safe* and *idempotent* methods and each gets a clearly defined semantics. From this point onwards, there were 8 methods, the most common being `GET`, `PUT`, `POST` and `DELETE`.

Since all client-server interactions follow a clearly defined semantics, 25 new status codes were introduced with HTTP/1.1, raising the total to 40 possible status codes. In addition, a new authentication scheme, Digest auth, was added to replace the un-secure Basic

authentication. Yet, history shows that Basic authentication is still popular, mainly due to its easy setup compared to other authentication mechanisms.

5.2.2. HTTP Components

Chapter 5.2.1 discussed some key elements of the HTTP protocol and how they have evolved together with the HTTP standard. This chapter discusses its four key elements more in depth: (1) Headers and how they work, (2) Status Codes, (3) Methods and their semantics and (4) URIs

Headers

Both, the client and the server exchange meta-information before transmitting the real content. This meta-information is called the *header* and is formed of multiple lines of key-value pairs separated by a comma. Headers are sent before the entity but after the status line for responses and after the request line for requests. They have existed since version 1.0 of HTTP and their aim is to influence the behavior of the server/client. Although most headers are not mandatory or strictly applied, they greatly improve the web-experience. For example, in a situation where a web-page is available in several languages, it would be great if the server could choose the right language for each client without having to ask first. Here, the header information can help to select the best matching web-page for each client. However, headers are not only intended for servers but also for clients.

Client-headers can influence the behavior of the server. For example, the cache mechanism introduced with HTTP/1.0, allows a user to specify the *If-Modified-Since* header. If set, this header tells the server to respond with the document only if its content has changed since the specified date. In a similar way, the server can send meta-information back to the client. On example would be the *ETag*, which later allows a client to ask the server whether or not it has a new version of a document, in which case it should be delivered. Thus, client- and server headers go hand-in-hand. Globally, all these headers can be grouped into four categories:

- *Generic-headers* are of general interest to the request. Both clients and servers use these headers since they are not related to the transferred entity. The *Date* header is an example of a generic-header.
- *Request-headers* further specify what the client is requesting, so they can be seen as modifiers, tweaking the request. For example, the *User-Agent* header, also visible in Listing 5.3 and the *Accept-** headers are in this category.
- *Response-headers* are used by the server to send additional information back to the client. They contain information about the server and about later access. One example of such a header is *ETag*.
- *Entity-headers* further describe the entity. Clients and servers can send entities, if not restricted by either the request method or one of the other headers. As such, these headers can be interpreted as meta-information associated with the entity. This includes information about the size (*content-length:*), the type (*content-type*) and the encoding (*content-encoding*) of the document.

According to both, [rfc1945] and [rfc2616], unrecognized headers are treated as *Entity-headers*. As such they are forwarded by proxies and will probably be ignored by the receiver.

Status Codes

Another key element of HTTP is status codes. These codes are like signals sent from the server back to the client. Through them, a client can easily discover what happened on the server side; a client can see whether everything went smoothly or whether an error occurred during the treatment of the request, and if so, what type of error.

Status codes have two parts: a numerical part and a *reason-phrase*, where the numerical part is made up of a 3 digit number and the reason-phrase is a string. Both parts can be taken independently and have the same meaning. Hence, the numerical status code *404* has the same meaning as the reason-phrase *"Not Found"*. The status line of each request always contains both. The numerical part is intended for web-browser or similar software whereas the reason-phrase is intended for human users.

Since not all outcomes are always covered by a concrete status code, they are divided into five categories, each representing one family of possible outcomes:

- `1xx`: denotes general information like `101 Continue`.
- `2xx`: denotes accepted and successfully executed requests.
- `3xx`: denotes all types of redirections. To fulfill the request, the client has to take further actions.
- `4xx`: denotes client errors like missing privileges or erroneous URIs.
- `5xx`: denotes server errors. They have more symbolic value and the client has often no influence on them and must abandon.

Just as for the headers, the number of available status codes has grown with each major version of HTTP. While the first version, HTTP/0.9, had no status code at all, the latest version HTTP/1.1 has 40 and 25 were introduced with HTTP/1.1, all defined in [rfc2616]. However, protocols extending [rfc2616] may add new status codes. This is the case of the TLS (Transport Layer Security) extension of HTTP defined in [rfc2817, p. 2] adding a new status code `426 Upgrade Required`. Other protocol extensions defining additional status codes are: [rfc2616, rfc2518, rfc2817, rfc2518].

Methods

HTTP/1.1 defines a set of 8 methods and their respective semantics regarding client-server interactions. Of course, this set includes methods for the four CRUD operations but also some targeted at debugging a connection or identifying a server's capabilities. The first four methods in Table 5.1 implement the CRUD interactions with a web-server, allowing it to create, read, modify and delete content. The other methods are less used and more intended for debugging.

Further methods exist for derived protocols like WebDAV [rfc2518]. Although WebDAV relies on HTTP as its underlying protocol, it extends the latter with additional capabilities to allow direct file manipulation. To achieve this goal, the WebDAV protocol needs a few additional methods like `MKCOL` to create a new collection on the server side. Of

Method	Description
POST	Creates a new sub-element.
GET	Requests a document.
PUT	Modifies a given element. If the element at the specified URI does not exist, it is created.
DELETE	Deletes the element at the specified URI.
HEAD	Requests a document. The server only sends back the associated header information without the actual content.
OPTIONS	Checks the server communication options. In the case of a RESTful service the server should answer with the WADL (Web Application Description Language) file, describing its resources.
TRACE	Is mainly used to debug client-server interactions. It allows a client to see what the server on the other side is receiving.
CONNECT	Is used for dynamically switching to a tunneled connection.

Tab. 5.1.: Available HTTP methods

course, most of these new methods require additional status codes and also supplementary headers.

5.2.3. URI: Uniform Resource Identifier

In order to request documents from a server, the client needs to uniquely identify the latter. One way to address a computer on the web is to use its IP address. IP (Internet Protocol) addresses are dot separated triples each made of digits. As such, one of Google's IP addresses is `173.194.40.83`. Remembering such complicated numbers is not only cumbersome but also has other drawbacks: (1) Often, one physical server hosts more than one service or more than one domain, so the server needs a mean to know which domain a client's request needs to be forwarded to. If the only information provided by the client is the server's IP address it would be impossible to route the traffic to the right virtual host, since they all share the same public IP. Besides, humans are very bad at remembering numbers. This gets worse as the numbers get bigger. Additionally, the IP address space is divided into separated groups, each managed by a registrar, which in turn allocates sub-groups to providers. Thus, if a server changes its provider, it also changes its IP address, which needs to be communicated to all past and future clients. (2) Additionally, the quantity of devices connected to the Internet has steadily grown over the past decades to a point where almost no more IPv4 [rfc791] addresses are available. Of course the solution to this problem is to extend the address space by making them longer. This is called IPv6 [rfc2460] imposing 32-bit addresses formatted like `fd7b:50c8:4645::c12:c5af:2a53:29be`. The IPv6 address space is so huge that in a foreseeable time-frame there will be more than enough addresses. Yet, this new abundance of addresses raises other questions like the privacy of users. However, these topics are outside the scope of this thesis. Yet, IP addresses are only half of the solution.

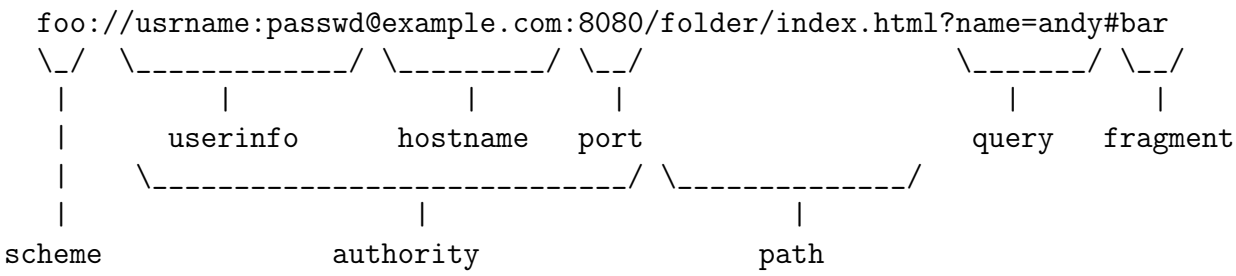


Fig. 5.2.: Full URI definition (after [rfc3986, rfc1738])

Whereas the second problem is solved with the upcoming IPv6, the first one remains open, also for IPv6 addresses. To overcome these limitations, the Internet has an abstraction layer above IP addresses, called the URI. Basically, a URI is just a string standing for a server. Thus, it can be seen as a name for servers on the web, each server with its own name. Figure 5.2 gives an overview of the main elements of a URI. The left part, separated from the rest by `://` is called the scheme and identifies the protocol used. Commonly, `http` is used here, however, other schemes are also possible like `ftp`, `dav`, `smb` or `nfs`. More recently these schemes are also used to call different handlers on a browser. The `ubuntuusers.de` wiki, for example, uses the `apt` scheme to launch APT (Advanced Packaging Tool) which handles the installation of new software in Debian based systems.

To reach the server, the URI still needs to be translated into an IP address, which is the only means to address a server. This translation, however, is transparent to the client and handled by the DNS (Domain Name System).

5.2.4. User Authentication

Identifying users not only opens new perspectives regarding the kind of content delivered to a given client but also allows building applications where privacy and security are a fundamental requirement like *e-Banking* and *e-Commerce* applications. Automatic identification goes from cookies to more sophisticated stochastic analyses based on fused information which can be gathered from the client (like browser history, user-agent, preferred language, operating system). Even though such approaches can more or less successfully identify a given user, they have some serious drawbacks regarding security. The information on which the analysis is based is public and so can be gathered by anyone. Later, the gathered information can be used to fake someone's identity. Cookie stealing is a popular technique to impersonate a user [WEB10, WEB9, WEB55]. Therefore, identifying a user is not enough to ensure that only a particular user has access to certain information. Instead, the server needs to truly authenticate him. The client needs to present some token that only he and the server knows. The server then decides whether this token is valid or not and by that, whether the user can enter the protected area or not. The holy grail for authentication would be zero knowledge proofs and more precisely, zero knowledge password proof. Yet, for now this remains in the domain of science-fiction, so this subsection focuses on the most common authentication systems used on the Web today.

Basic Authentication

The simplest and also the oldest way to authenticate clients is called *Basic Authentication*. It has been part of the HTTP standard since version 1.0 and defined in [rfc2617]. To be authenticated, a user has to: (1) identify himself and (2) present a token. The identification is usually done through a username, unique per authentication domain (basically for each domain). This allows matching the client to a given user account where some client-related information is stored. The authentication is a simple token which only one client should know. All this information is sent in the `Authorization` client-header introduced for this purpose with HTTP/1.0. Its value is defined as the base64 encoded string composed of the username, the ":" character followed by the password. Listing 5.6 shows an authenticated GET request for the user *foo* with the password *bar*. Line 7 shows the `Authorization` with the base64 encoded authorisation string. In addition, the server may ask a client to authenticate with the `WWW-Authenticate` header.

```
1  ruppena@tungdil:~ telnet www.example.com 80
2  Trying XXX.XXX.XXX.XX...
3  Connected to www.example.com.
4  Escape character is '^]'.
5  GET hello.txt HTTP/1.0
6  User-Agent: Mozilla/3.0 (X11; I; AIX 2)
7  Authorization: Basic Zm9vOmJhcg==

9  HTTP/1.0 200 OK
10 Date: Thu, 29 Nov 2013 13:58:34 GMT
11 Server: CERN/3.0 libwww/2.17
12 Content-Type: text/html; charset=ISO-8859-1

14 Hello World!
```

List. 5.6: Basic authenticated GET request

Although basic auth is the oldest authentication mechanism, it is still widely used due to its simplicity of implementation on both the server and the client side. However, basic auth also has some severe drawbacks, the biggest being security considerations. If the connection between the client and the server is not secured by additional means (like SSL or TLS) an attacker can easily sniff the traffic and save the `Authorization` header for later use.

Digest Authentication

To overcome the limitations of basic auth, a new authentication scheme was introduced with HTTP/1.1, called *Digest Authentication* and specified in [rfc2617]. This protocol is similar to the *Message Authentication Code* principle. Instead of sending a static string, upon the first request, the server sends back a nonce. The client uses this information to compute a hash based on the nonce, the username, the password and the requested URI. This hash is then sent back to the server to authenticate the user. According to [rfc2069, pp. 5,6] The computation of this final hash is done in three steps:

1. $HA1 = \text{md5}(username : realm : password)$

2. $HA2 = \text{md5}(HTTPMethod : URI)$
3. $response = \text{md5}(HA1 : NONCE : HA2)$

Listing 5.7 shows in lines 4-9 the necessary HTTP header to successfully authenticate. From the enumeration above and the description of the client header in Listing 5.7, it seems that this authentication scheme is much more complicated to implement, on both the server and the client sides. This mainly comes from the fact that digest auth is a compromise between insecure basic auth and the very secure public key or Kerberos authentication.

```
1  GET / HTTP/1.1
2  Host: www.example.com
3  User-Agent: Mozilla/5.0 (X11; Linux x86_64)
4  Authorization: Digest username="foo"
5                       realm=security@example.com
6                       nonce= dcd98b7102dd2f0e8b11d0f600bfb0c093
7                       uri="/"
8                       response="9d078500e404d2789620909c56c26497"
9                       opaque="5ccc069c403ebaf9f0171e9517f40e41"
10 HTTP/1.1 200 OK
11 Date: Wed, 20 Nov 2013 14:08:02 GMT
12 Server: Apache/2.2.22 (Ubuntu)
13 Last-Modified: Tue, 29 Jan 2013 11:54:41 GMT
14 ETag: "141e70-b1-4d46c0f0f31a2"
15 Accept-Ranges: bytes
16 Content-Length: 177
17 Vary: Accept-Encoding
18 Content-Type: text/html

20 <html><body><h1>It works!</h1>
21 <p>This is the default web page for this server.</p>
22 <p>The web server software is running but no content has been added, yet.</p>
23 </body></html>
```

List. 5.7: Digest authenticated GET request

The introduction of the nonce disarms many attack vectors known to work for basic auth. As such, an attacker cannot compute the username and password, since the user never sends them directly, only combined with other, context dependent information. Although [rfc2617] does not specify how the server has to compute the nonce, nor impose a given generator algorithm, it gives some wise advise about what a good generator algorithm should be based on. Server time is one factor as it can allow disarming replay attacks, where an attacker saves a message for later replay. Since the nonce is based, among other things, on a time construct, the server can judge whether the given nonce is a valid one, and based on that, reject a message, even though the authentication header is correct. Later, the standard was slightly modified and adapted to introduce a few security extensions. However, they remain optional and basically introduce a client-generated nonce, called `cnone` to complete the server generated one.

Today, one of the biggest weaknesses of digest auth is password storage. To compute the hash, the client needs the password to compute part $HA1$ of the hash. However, the server also needs the clear-text password to make the exact same computation server side. Therefore, the passwords need to be stored in a recoverable format on the server

side. As learned from recent break-ins[1], storing passwords in a secure way is just as important as protecting the authentication [56]. Additionally, `md5` is considered to have been broken in recent years. Soon after the introduction of `md5` in 1991, some researches proved that collisions were possible to create. In recent years, this has proved more and more to be true, and with tools like the rainbow tables developed by Philippe Oechslin at EPFL (École Polytechnique Fédérale de Lausanne) or the availability of GPU calculus, has become a real problem. However, it is believed that these attacks do not harm `md5`, since the clear-text password is never known.

OAuth

Although OAuth is not part of the HTTP authentication family like basic or digest auth, it has some irrefutable advantages over the other authentication schemes. Services like Twitter have greatly pushed the deployment of OAuth over the past few years to a point where their APIs are only accessible over OAuth. This is also true for many other RESTful applications, where OAuth is widely used [2].

Compared to other authentication schemes, OAuth is quite young since it was first presented back in 2007 as an alternative to the OpenID initiative started by Twitter. The challenges remain the same as for other authentication schemes: identifying a user without revealing credentials to a possible attacker. OAuth even goes further, once authorized, different applications are mutually authorized and can exchange data without knowing the user's username and password. OAuth does this by generating an application token which authorizes a given application, or the user himself. However, OAuth does not really improve the security of the first login or authorization process. Whereas OpenID is all about the Question "Is this really *Mister X*", and how to prevent attacks on the account of *Mister X*, OAuth answers another question: "Is application *X* allowed to talk to application *Y*?". This is also the reason OAuth is so popular in RESTful applications: it allows authorizing a given application to use the content of a different one.

In short, OAuth defines four roles:

1. The *Resource Owner*: is the entity which decides whether access to a protected resource is granted or not. If the resource owner is a human, this is the user.
2. The *Resource Server* hosts the protected resources.
3. The *Client* is an application requesting a protected resource on behalf of the *Resource owner*.
4. The *Authorization Server* issues an access token to the client if the resource owner is successfully authenticated.

Based on these four roles, to consume a protected resource, the *Client* first needs to ask the *Resource Owner* for an authorization (A). With the authorization granted, the client authenticates with the *Authorization Server* (C). If both, the authentication and the authorization grant from (B) are valid, the *Authorization Server* issues an access token. In the final step, the client can access the protected resource by presenting the access token issued in (D). Discussing the details of OAuth is outside the scope of this thesis however, Figure 5.3 gives a rough overview of the main participants in an OAuth

[1] Adobe, juste to cite the last incident
[2] Twitter, Facebook, Dropbox, Flickr, Google to cite a few popular examples

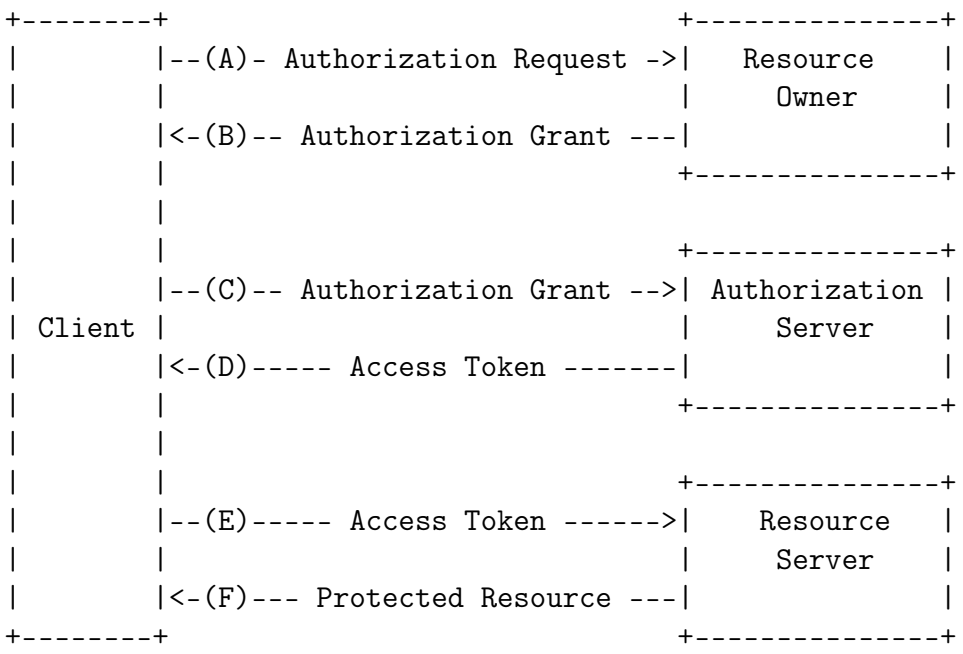


Fig. 5.3.: OAuth information flow (after [rfc6749, p. 6])

authorization. The OAuth specification [rfc6749] on its own is double the size of the specifications of basic and digest auth together.

5.3. REST: Representational State Transfer

Section 3.5 briefly discussed RESTful web services comparing them to classical WS-* services and showing their differences. This chapter introduces the architectural style behind this type of web service and its implications as a basis for Section 5.4 where REST architectures are applied to realize RESTful web services. For a better understanding the first part of this chapter introduces the concepts of *REST* and *RESTful* and also how they differ. In the second half, Roy Fieldings criteria for a REST architecture are discussed.

5.3.1. REST vs RESTful

Roy Fielding introduced back in 2000 the term REST(Representational State Transfer) as an architectural style. His PhD thesis [14] and in particular, REST architectural style, are the outcomes of his contribution to HTTP/1.1 and various concepts strongly coupled with it, like URI design. Roy Fielding is also a contributor to many RFC (Request for Comments) in the domain of web architectures [rfc2616, rfc1945, rfc2396, rfc1808]. While all these contributions are of a concrete nature and they all have concrete implementations, his thesis resumes all these efforts in a common architecture. According to Fielding [14, p. 86]:

> The Representational State Transfer (REST) style is an abstraction of the architectural elements within a distributed hypermedia system

According to this, REST is not bound to the web nor to HTTP or any other protocol. In his thesis, Roy Fieldings keeps the discussion at an abstract level. Today, this is

maybe the most noticeable facet of his work. Whereas HTTP/1.1 might be outdated one day and be replaced by its successor, the concepts behind it remain valid and so does Roy Fieldings' thesis. REST architectural style imposes a clear separation of concerns as outlined in Subsection 5.2.1, about the *Data Elements*. Fielding clearly separates the resources from their representation and from their URIs. Furthermore, the way a client communicates with a server is standardized and remains the same throughout all the different realizations of this architectural style.

The architectural style as described by Roy Fielding in his Ph.D. is called *REST*; referring to it implies the compliance with the 5 concepts discussed in Subsection 5.3.2. As a consequence of REST, several researchers have defined sets of criteria a web service has to fulfill to be compliant with REST. Web Services falling into this category are called *RESTful web services* [B18, B5] — a web service is called *RESTful* if it complies with all REST concepts [B5, p. 4].

5.3.2. Roy Fielding's REST Principles

After deep reflection about HTTP/1.1 and the release of the associated specification [rfc2616] in 1999, Roy Fielding was ready to publish his Ph.D. describing the underlying architectures of systems like HTTP/1.1 insofar as HTTP as a reference implementation of the concepts outlined in his thesis. The core of this work is presented in [Ch 5 14, pp. 76-85]. The following discussion covers each of the five criteria and their impact on the system. Fielding starts with a *Null Style* hypothesis. At the beginning the system already contained all it needs, but had no structure at all. Through an iterative process, boundaries were defined and the system becames more and more structured.

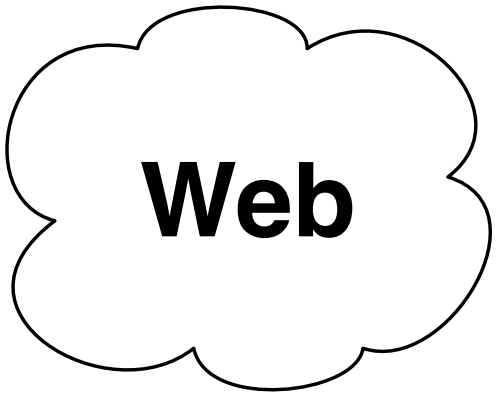


Fig. 5.4.: Situation before adding constraints: The *Null Style*.

Figure 5.4 shows the WWW before adding any constraints. Although the system worked and all components were there, they were indistinguishable at that time. Only by adding one or more constraints did they become visible. Defining these components, the boundaries between them and how they interact with each other is the aim of any architectural style.

Client-Server

The first constraint is the *Client-Server architecture*. Figure 5.5 shows the new state of the system after adding this constraint. Already some components have become visible and identifiable. The user-interface is separated from any data-storage and processing. This separation of concerns allows each tier to evolve independently. Such an evolution does not only concern implementation and features but also scalability and portability. Another benefit of this architecture is that several clients can use the same server.

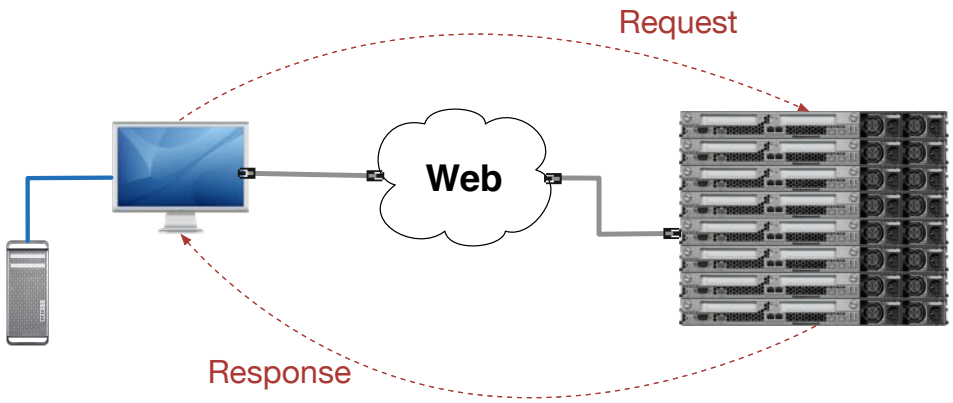


Fig. 5.5.: Client Server Interactions

Along with P2P (Peer-to-Peer) architectures, Client-Server is the most common network architecture. A client makes a request to a server, the server then decides whether it will execute the request or not. In any case the server responds to the client either with the answer to the request or by indicating that it has refused to execute the request.

Stateless

The second constraint added to the system is *statelessness client-server communication*. This means that a server will not recognize a recurring client, nor can it derive a client's state based on its former requests. A client must send all the necessary information upon each request. In that way, the session's state is no longer kept on the server side, but the client side. The statelessness constraint induces three properties: (1) Visibility, (2) Reliability and (3) Scalability. *Visibility* means that the service does not need to wait for additional data sent by the client to execute the request. All the necessary information is contained in the request. *Reliability* allows recovery from failures with ease. Since no client related data is tracked on the service side, recovery is done by re-executing the request. *Scalability* allows adding more servers and load balance between them without a complicated session handling or some means of sharing space. Additionally, no memory is occupied on the server beyond the request, which makes the server working more efficiently.

This constraint also induces some drawbacks, the biggest one being the increase in network traffic. Since the client is responsible for session tracking, this piece of information needs to be sent together with each request, thus increasing the amount of packets sent over the network. Another drawback is the loss of application control. The server has no control over the session state of any client and therefore cannot influence it. The server has no

means to correct faulty client session handling. A direct consequence is the increased engineering necessary for each client implementation.

Cache

The third constraint is the addition of a client side *cache* system. Content served by servers does not always change between requests. Therefore, for some content, it makes sense to keep a local copy for further consultation. Yet, the client still has to ask the server if the content has changed since the last visit, and if so, the server will send back the new document and the final cost, measured in bandwidth, is the same as without a cache system. However, if the cached copy is still the latest version of the document, the server does not need to send a new copy of the document to the client, thus greatly improving bandwidth consumption and also speed. Consequently, adding a cache improves the average latency of the interactions. While the notion of a caching system had been around since HTTP/1.0, it was greatly improved with the introduction of the *ETag* header in HTTP/1.1. Moreover, such a cache can also sit somewhere in between the client and the server on a proxy. The role of the proxy is then to decide whether the request needs to be forwarded to the server or if it can reply with a cached copy of the requested document.

Uniform Interface

The fourth constraint consists of the language used by clients and servers. REST architectural style imposes the use of a *Uniform Interface*. Compared to other means of delivering services over a network, the constraint of a uniform interface for REST architectural style is a remarkable choice. It applies the principle of generality [B7, p. 52] to the component interface. This greatly simplifies the architecture and increases visibility. Furthermore, it eases the integration of different services. Imposing a uniform interface also makes a separation of concerns; each component is responsible for doing its own business. Yet, the service interface of all these components is the same. Therefore, REST architectural style combines two powerful software engineering principles: the separation of concerns and the principle of generality. However, this choice induces some inefficiencies. A generic interface, although applicable to various situations will always perform worse than an interface developed for a specific purpose. This difference can be clearly identified when comparing RESTful web services with their WS-* counterpart [64].

The demand for a uniform interface implies the introduction of some further constraints on the components: (1) Each resource can be uniquely identified. (2) A resource can be manipulated through its representations. (3) Messages are self-descriptive and (4) Hypermedia as the Engine of Application State (HATEOAS). According to Fielding, these four constraints are the architectural elements necessary to define a concrete uniform interface. However, they are not part of the 5 basic constraints of REST architectural style.

Layered System

The fifth constraint imposes a *Layered System* to REST style architectures. Layered communication protocols have been well known since the beginning of computers. The most prominent example of a layered communication system is the OSI model for the TCP/IP

stack. Layered systems reduce the complexity of monolithic systems by separating and encapsulating functionality into different concerns. Each layer is closed (self-contained) and provides some services to the layer above. A layer does not need to know the inner guts of the layer below, which allows changing the implementation of a given layer without breaking the system. Although the separation of concerns introduces a small overhead, perceived as latency by the user, approaches like caching compensate for this drawback.

Layered systems, introduce proxys and gateways. For a client it does not matter whether a response comes directly from a server or whether it comes from an instance between him and the server; proxys and gateways are transparent to the user. As such a proxy can be seen as a shared server for many clients with responsibility for forwarding the incoming requests to the right service. In extension, layered system allow for 2-tier, 3-tier and N-tier architectures. Whereas proxies are transparent to the client, gateways appear as normal servers. Yet, they don't contain the requested resource but can transform incoming requests, forward them to a suitable server, and translate outgoing responses. These gateways are by far the most common application of the layered system constraint for REST style architectures.

Code-On-Demand

The sixth constraint allows a client to download pieces of code and execute them locally. Unlike the five other constraints, *Code-On-Demand* is optional. Downloading pieces of code implementing client-specific behavior allows the speeding up of the design and implementation of different clients. Yet, code-on-demand reduces considerable the visibility of a service. Whereas the uniform interface constraint together with the four derived constraints clearly describe any interaction, code-on-demand introduces a level of obscurity into the system. At a given point, the result of a request will be some further requests executed by code downloaded in the first step. Moreover, due to the limitations of firewalls and network policies, a server cannot be sure a given code has successfully been downloaded and deployed for the client. Making this constraint optional has advantages in situations where firewall problems, not available client execution environments etc. can be excluded without suffering from any drawbacks.

5.4. ROA: Resource Oriented Architectures

Fielding presents in his PhD five criteria, resumed in Chapter 5.3, that an architecture needs to fulfill to be considered a REST style architecture. Although the architectural style is the result of work on HTTP, the thesis does not discuss any particular implementation. Fielding takes a formal approach. Starting with a null hypothesis, he constructs an architecture by adding constraints and observing the consequences.

Richardson & Ruby, Bill Burke and others have spotted the potential of REST architectural style and have applied it to web services. Thus, the term *RESTful* is a classification of web services, where a web service is said to be RESTful if it satisfies the REST architectural style criteria, as discussed in Section 5.3. Yet, the concept of RESTful does not help architects to design web services respecting REST style architecture. REST criteria are abstract, like mathematical axioms, so they are not a big help when designing RESTful

web services. Richardson & Ruby compare REST and RESTful to object-oriented programming. They state that using an object-oriented programming language does not make the program itself object-oriented [B18, p. 12]. To overcome this limitation, both Richardson, Burke and many others, have defined a set of concepts and properties to create RESTful web services. Richardson in his book *RESTful web services* called the resulting architecture ROA(Resource Oriented Architecture).

As an architecture rather than an architectural style, it is concrete and contains at the same time a recipe for how to implement such an architecture. Depending on the literature (Richardson, Ruby [B18], Burke [B5], Wilde [79], Pautasso [64]) the exact concepts and properties differ but remain the same in essence. As such, Burke [B5, p. 21] defines 5 properties a web service needs to satisfy to be RESTful. It has to have: (1) Addressable Resources and (2) A uniform, constrained interface, (3) be Representation-oriented, (4) Statelessness communication and (5) use Hypermedia as the Engine of Application State. In [64], Pautasso et al. argue that representation-orientation can be left out, leaving only four principles. Erik Wilde further reduces the set of properties a RESTful web services has to satisfy and only keeps two criteria, mainly (1) Statelessness communications and (2) Addressable Resources [79].

This large palette of interpretations shows that REST architectural style is popular but, Fielding's definition leaves space for a lot of interpretation. The remainder of this Chapter discusses the definition given by Leonard Richardson and Sam Ruby, as this is the most complete one. Moreover, they not only define RESTful architectures but also give a methodology. As such, Richardson's approach is similar to what is called *a proof by construction* in mathematics.

5.4.1. Concepts

In Chapter 4 of *RESTful web services*, p. 171 Richardson and Ruby wrap up ROA (Resource Oriented Architecture). Accordingly, an architecture has to follow the 4 concepts presented in this Chapter plus the 4 properties of Subsection 5.4.2.

Resources

The term ROA refers to both *Resource* and to *Architecture*. Whereas the term *Architecture* is clearly defined, the interpretation of *Resource* varies, depending on the context. A resource in ROA and in Business Processes do not share the same definition. Even worse, they are two disjointed, unrelated concepts. According to [B18, p. 136]:

> A resource is anything that's important enough to be referenced as a thing in itself.

Although this definition seems vague at a first glance, it boils it down to its simplest form. By that, a resource might be:

- A bird,
- The wikipedia article about *Douglas Adams*[3],
- The temperature in my office,
- A cup of coffee,

[3]http://en.wikipedia.org/wiki/Douglas_Adams

- The next Java release,
- A list of todos,
- The research budget of the university for the current year.

Therefore, a *Resource* is anything a user might refer or link to. In this term, Richardson et al. do not differentiate between a resource with a physical manifestation or a purely virtual. Although, he states that representations of physical objects „. . . are bound to be disappointing“ he does not further investigate this problem at first.

Richardson et al. integrate resources like a bird for example, into the system by the convention that a user never accesses a resource directly. He interacts with it through representations (see Subsection 5.4.1 for more details about representations.) In order to interact with a resource, the user has to address it somehow. In RESTful systems, URIs are used to identify a resource (see Subsection 5.4.1).

URIs

Each resource needs a least one URI through which a user can talk to the resource. This can be seen, like the address of the resource, and where users have to look for it. The concept of resource is highly coupled to that of URIs. If something is not identifiable via a URI it is not a resource. Richardson et al. use the URIs as defined by Tim B. Lee in [rfc1630].

URIs are highly coupled to the Web since they have managed, for the first time in the history of computer networks, to provide a simple yet powerful way to combine protocols and contents in a usable manner. With URIs it is possible to tag content on the web, to share it or save it for later consultation. Therefore, URIs play an important role in the development of the web and since RESTful web services are based on the building blocks of the web, they do the same for RESTful web services.

The biggest problem with URIs is how they are structured. Opinions on this topic vary. The supporters of Roy Fielding claim that „A REST API must not define fixed resource names or hierarchies (an obvious coupling of client and server).“[4]. They argue that REST uses a uniform interface which induces HATEOAS (see Subsection 5.3.2) consequently, a client only knows one entry point and discovers related resources from this. Others, like Richardson et al. argue that URI design is important to support meaningful URIs. Both approaches are defensible; Fieldings' point of view has the advantage that clients and server are always loosely coupled. Even if the set of resources changes, the client will still work properly. On the other hand, Richardson's approach has the merit that clients can browse faster to find a given resource. For example, when a client is looking at a financial report for the year 2013, he can easily guess the URI for the same report as the previous year (given that this report exists). Following Fieldings' approach, such an induction would be impossible since URIs are not predictable.

Yet, introducing semantically meaningful and predictable URIs introduces another problem. When asking two different people on how they would name a given resource, chances are high they offer different answers. Consequently, these edge cases need some special attention. One such edge case is whether one resource can have two or more URIs pointing to it. For example the URI `http://example.com/shoes/drmartens/1460` and

[4] http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven (last retrieved December 8, 2013)

`http://example.com/shoes/product_of_month` might point to the same type of shoes. Whereas this does not cause any trouble regarding the definition of URIs, Richardson et al. propose to chose one of the URIs as the canonical one (which should respond with the status code 200 when a client requests it) and all others should be pointers to the former (respond with a 303 plus the canonical URI). The second edge case to consider is what happens when one URI points to two or more different resources. Reading through the specification for a URI, this is clearly not allowed and would break the whole concept of URIs. Therefore, this case does not need special care.

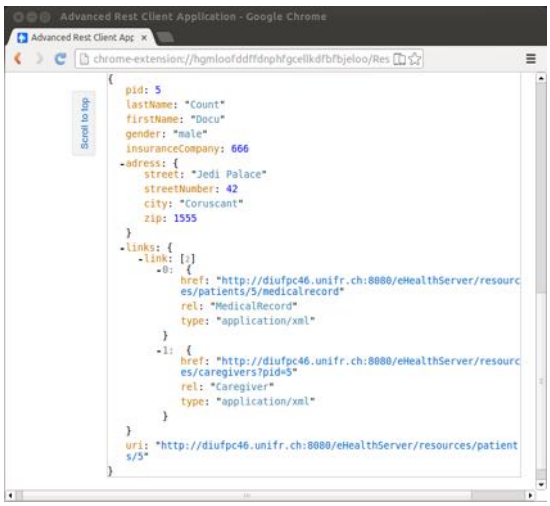
In sum, using semantically meaningful URIs for RESTful applications, while not a great advantage for non-human clients because of their loose-coupling, greatly improves the user experience for humans.

Representations

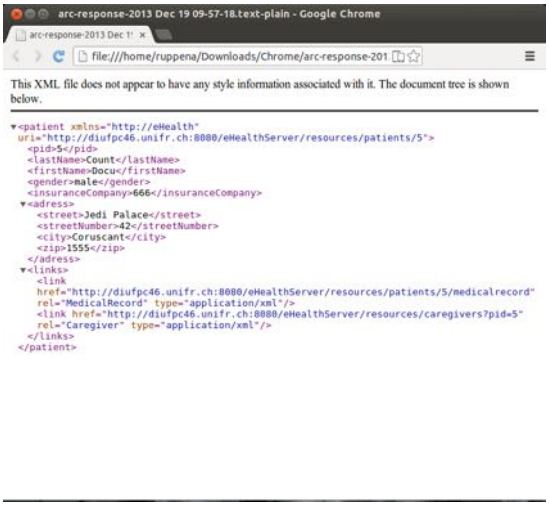
Each application is composed of several resources. They represent the view of the developer of the structure of the application data. A resource is more an idea or a concept than something real. To access the information held by a given resource, the client needs to know, or to discover its associated URI through which the client can request the resource. A web-server cannot send back a concept, but a *representation* of it. A resource is composed of data about something and has several representations associated with it, through which clients interact with the resource. For instance, if a client requests a resource, the server sends back an adequate representation of it. On the other hand, a client can also send a representation in the body of the request, which is then used by the server to either create a new or update an existing resource associated with the requested URI. Figure 5.6 depicts this situation. Figures 5.6a, 5.6b and 5.6c show three different representations of the same resource. The information contained in these representations is always the same, the only difference being how the information is presented. Figure 5.6a is a JSON representation of the resource. This content-type is meant for machine-to-machine interaction and web service consumption by programs. Figure 5.6b is an XML representation of the same resource. Whereas it is widely used for machine-to-machine interactions, it is also quite easy for people to read. Finally, Figure 5.6c is the HTML representation and is mainly intended for human users.

If a resource holds several representations, the client needs to choose the right one somehow. The traditional approach is to rely on HTTP and its content negotiation capabilities; based on the header sent by the client, the server can make an educated guess about which representation will be the most suitable for this client. Since representations of a same resource not only differ in their content-type but also in other aspects like the language, the server can base its choice on several headers. For an English speaking client preferring HTML representations, the server will try to send back an English HTML page of the requested resource.

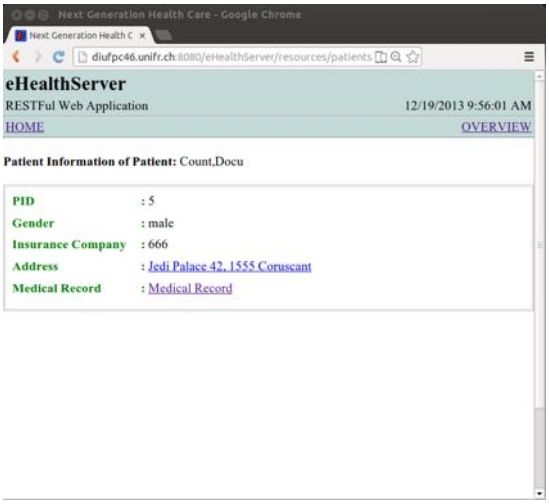
Richardson et al. advocate another approach, stating that it is useful to introduce a new URI for each representation. The canonical representation is accessible through a canonical URI and other representations are available under the URI with an according suffix. This approach also has its advantages. The URI contains all the information sent to the server by a client. By bookmarking this URI a client will always get the same result, assuming the resource has not changed. This behavior would not necessarily be



(a) JSON RepresentationLAN



(b) XML Representation



(c) HTML Representation

Fig. 5.6.: Several Representations of the same Resource

true if the information guiding the selection of an appropriated representation were based on headers sent together with each request.

Whereas Richardson et al.'s argument is a question of taste, there is another reason to prefer different URIs for different representations: if the client is non-human, it is important that for each request, the same representation is sent, otherwise, the client will have difficulty interpreting the received bytes. This is more difficult if the server makes this decision for the client. The unpredictability comes from the fact that the interpretation of these client-headers is not mandatory. The server is free to ignore these headers. Whether one approach is preferred over another often depends on the programming language used to implement the RESTful service.

So far, two types of resources have been discussed; those representing data, for example, the actual weather in Fribourg and those representing physical objects like a light bulb. Since the representation is the only bit of information a client can see of a resource, it should be related to it and be useful for the client. It is rather easy to define representation in the first category: data is best represented by data. However, what is a

representation for a physical object? Since a web-server can only send bytes, the object itself cannot be a representation. However, meta-data about the object is a good representation. Since meta-data is also data, it is the closest we can get in terms of data for a physical object. For the light bulb, its actual state would be meta-data. Fortunately, this information represents a smart light bulb quite well. Of course, this smart light bulb can be characterized through additional meta-data like its color temperature, its socket, its wattage etc. It is up to the developer to chose the right set of meta-data to represent the physical object.

Links between Resources

RESTful web services encourage linking different resources together. The representation of a resources should, if possible, contain links to related resources. This is a key difference between an RPC-style service and a truly REST-ful service.

Links are one of the big concepts of the web and are responsible for the success of the web. Humans don't browse the web by entering different URIs; instead, starting from one point — the Google search results page, for example — they explore the web by following links. REST advocates this principle also for the non-human web as well. The main argument for introducing links between resources is the loose coupling of client(s) and server(s). If the client discovers related resources by parsing the representation it can handle situations where new resources appear or old ones disappear. A hard coded client would not be able to manage such a situation and would break.

5.4.2. Properties

Besides the 4 concepts of Subsection 5.4.1, a Resource Oriented Architecture also adopts the 4 properties presented in this chapter. The properties are derived from the above concepts so that they complement them and make the definition of ROA more precise.

Addressability

A RESTful web service breaks the available information down into resources, each one representing one piece of information. Subsection 5.4.1 explained that each part of interest should have an associated resource. If the space of available information is divided into resources, a user can request and manipulate them; an application is said to be *addressable* if it exposes its pieces of information over resources. Since each resource has an associated URI, the application is addressable if it has many (and potentially infinite) URIs.

Addressability is one of the key differences between ROA and WS-* like architectures. This also explains why there is so much confusion between REST and many RESTlike or REST-RPC services. The latter combine the advantage of addressability with the programmatic approach of remote procedures. This also seems to be the most important property of a RESTful web service for a human user. For many other problems, a user will find a way to work around them or accept the imposed way. Thus, if a piece of information cannot be addressed, it will be difficult to work with.

Moreover, URIs can be chained. This, for example, makes it possible to have Google Translate translates a whole web-page by passing its address as a parameter. If the web-

page to be translated is not addressable, the only means of specifying the input for Google Translate would be to copy the text.

Statelessness

Statelessness is the second property of ROA. Richardson et al.'s definition is the same as Fielding's. A set of interactions is stateless if all requests of a set can be carried out in complete isolation. As a direct consequence, the server does not recognize a returning client; instead, he needs to send all the necessary information together with each request. This means that a client will never have to first bring the server into a given state before executing some request.

Yet, statelessness does not completely rules out states. A client can have a state and track its path through the resources. A resource can also have a state as addressability means that every interesting piece of information has to be exposed as a resource. Thus, if the state of a resource is important, it is exposed as a resource and the client can interact with it. However, the interaction between a client and a server is always stateless.

Searching Amazon for Jellyfish, for example, the client either uses the provided form or he accesses the URI `http://www.amazon.de/s/ref=sr_pg_2?keywords=jellyfish`. The resulting web-page contains links to articles related to Jellyfish but also links to further results. In the first place the user is interested in the resource *all Jellfish related articles*. The server answers with a partial list and links to further results. Yet all theses further results pages have unique URIs and can also be addressed directly. Therefore, if the client is interested in the third results page of the Jellyfish search, he can directly access it through its address `http://www.amazon.de/s/ref=sr_pg_2?keywords=jellyfish&page=3`. Thus, the interaction between the client and the server remains stateless.

Connectedness

Connectedness is highly coupled with the concept of Links between Resources (5.4.1). The human web is popular because resources are well linked together and allow a seamless browsing experience. Google also uses this property to judge a website's quality. Highly connected web-sites were more highly ranked than web-sites without any links [WEB42]. Google not only applies these principles for judging scanned web-sites but also integrates them into their own products. For example, the representation of a Google search contains a list of links to other resources. These resources are associated with the search query and the query resource is linked to related result resources. The representation also contains more links: navigation, translation, cached copies etc. Richardson et al. call them *levers of state*. They indicate to the client how to get from one page (current state) to the next page (future state). Listing 5.8 shows the XML representation of a simple door containing no links at all. Such a resource is not connected and therefore is not allowed in a RESTful web service. However, it is simple to make this resource connected. Listing 5.9 shows on line 6, that it is sufficient to add a link pointing to a related resource, indicating whether the door is open or closed.

```
1 <door>
2   <id>496</id>
3   <color>brown</color>
4   <height units="cm">195</height>
```

```
5   <width units="cm">110</width>
6 </door>
```

List. 5.8: RPC Resource

```
1 <door>
2   <id>496</id>
3   <color>brown</color>
4   <height units="cm">195</height>
5   <width units="cm">110</width>
6   <ref name="locked">http://example.com/door/496/locked/</ref>
7 </door>
```

List. 5.9: REST Resource

If a resource is linked to many other resources and contains a collection of links allowing a user to walk on a path from one resource to the next, it is said to be *connected*. Thus, *connectedness* is a measure of how much a resource is *connected*. Since these connections represent the choice a client has to make to go from one state to another, they can be seen as a tree in through which the client will choose a path. For this reason Richardson et al. argue that Hypermedia as the Engine of Application State (HATEOAS) is a synonym of *connectedness*. According to Richardson et al. HATEOAS can be defined in the following way [B18, p. 159]:

> The current state of an HTTP "session" is not stored on the server as a resource, but tracked by the client as an application state, and created by the path the client takes through the Web.

Therefore, the connectedness is a mandatory property for HATEOAS. If a resource has no links to others, a user cannot navigate from it to some other resource.

A Uniform Interface

A Resource Oriented Architecture must adopt a uniform interface, as described by Fielding and discussed in Section 5.3.2. Although HTTP is the most prominent implementation of a uniform interface for REST operations, Fielding does not impose any concrete implementation. Richardson et al. take a more practical approach, and adopt HTTP as their uniform interface.

According to Richardson et al. six of the eight HTTP methods are of interest for ROA: 4 are used to construct the constraints of the uniform interface and 2 offer some interesting capabilities on top of the 4 base methods but are not necessary to construct a uniform interface. To manipulate data, or in this case, resources, an interface needs at least four methods, one corresponding to each CRUD action. `GET` is used to retrieve resources, mainly done as shown in the various examples in Section 5.2 as in Listing 5.4, where the resource at `www.example.com` is retrieved. On the other hand, `DELETE` is used to destroy a resource. The body of the response to a `DELETE` request may either contain a status messages or nothing at all.

Modification is done with `PUT`. However, depending on whether a resource already exists at the supplied URI or not, `PUT` acts as a modifier or creator. Given a URI which already exists, a `PUT` request updates this resource with the new representation sent together with the `PUT` request. If, on the other hand, no resources exists at the supplied URI,

a `PUT` request will create a new resource according to the representation sent with the request. `PUT` used for updating and/or creating can be quite confusing. However, it is fully compatible with any CRUD approach. Since the user specifies the URI of a concrete resource, he wants to modify exactly this element. The creation of a new element under this URI is just a special case of modification where the initial state is `NULL` and the final state is a resource. One can argue that the resource already existed but without any content.

New resources are created with a `POST` request to the parent URI of the future resource. Unlike `PUT`, which sometime creates and sometimes modifies a resource, `POST` is only used for resource creation. The user supplies together with the request, a representation of the resource. Whereas the definition of this method is somewhat fuzzy concerning how `POST` behaves: *„... The actual function performed by the POST method is determined by the server ...“* and *„... The action performed by the POST method might not result in a resource that can be identified by a URI...“* [rfc2616, p. 53], in the context of REST and ROA its behavior is well defined. Upon sending a `POST` request together with a representation, the server will create a new resource with the supplied representation having an address that is a sub-URI of the requested one.

5.5. Key Concepts introduced in this Chapter

This chapter introduced the notion of REST architectures, of RESTful web services and Resource Oriented Architectures. These three concepts play a major role in building the xWoT. Like the WoT, the extended WoT uses RESTful web services to connect smart devices with the virtual world. Understanding the implications of REST architectural style, the underlining HTTP protocol and ROA (Resource Oriented Architecture) are key elements in defining a correct meta-model for the xWoT.

Fielding's principles offer the most complete description of a REST architecture but as discussed, also the most abstract one. Richardson and Ruby who translated the REST requirements to web services, formulate a slightly different set of constraints and introduce ROA. Although, the xWoT components generated from xWoT models which are derived from the xWoT meta-model are more concerned with ROA then REST, it is still necessary to take into consideration the basic REST constraints as introduced in Fielding's work to build a viable meta-model.

Obviously, *resources* are of central interest to the meta-model and its associated methodology. Starting with an observable entity, the first step in the methodology is to break down the entity into resources and define their mutual dependencies in the form of a tree hierarchy. Tied to the concept of resources are also its various *representations*, like `JSON`, `XML` or `HTML` and the contained information. Addressability and naming conventions are another hot topic for the xWoT meta-model and some conventions are already given by the meta-model and therefore applied to each concrete model.

The concept of links between different resources and the property of *connectedness* are, although slightly different coupled. But, what is true for the web should also be true for the xWoT. Instead of knowing all available resources, users should discover them by browsing. As mandated by REST architectures xWoT components have a *uniform interface*. For now, HTTP is used as the interface but, the metal-model being protocol

agnostic, it is just a matter of changing or extending the generator to incorporate another protocol like CoAP [rfc7252].

Considering the duality of an xWoT component with a visible outer interface over which it is connected to the Web plus a hidden inner structure, this chapter forms the foundation of this outer interface and influences how the meta-model represent it. The xWoT adopts Richardson and Ruby's point of view and definition of RESTful web services. Additionally, the xWoT mandates for meaningful resource names, facilitating browsing for human users.

6
Software Components

6.1. Introduction

Today, software components are a common approach when designing new software. The principles of independent and reusable software coupled with the power of object-oriented programming languages avoids the software crisis. Components can be seen as the next logical step after classes and objects.

This chapter is divided into three parts. The first lays the foundation of software components from procedural languages over object-oriented languages to finally coming up with truly reusable pieces of code, the software components. The second part mathematically introduces interfaces. With the help of Hoare triples basic requirements for any interface can be formulated. The third and last part, introduces a component's life-cycle. Starting with a real world example, the notion of lifecycle is translated to software components and will later be applied to xWoT components.

6.2. Foundations

Over the past few decades, hardware capabilities and available computing power has steadily increased. In the early days, the complexity of computer programs was highly coupled with this increase in power. This situation leads to what is known today as the *Software Crisis*. Not only did the quantity of code grow but this growth went along with

a number of failed and abandoned projects. Software became unmanageable, unverifiable and incomprehensible. On the other hand, the increase in computing power allowed the creation of re-usable software. In the past, CPU cycles were expensive and programs could not afford to waste too much of them. As a result, each piece of software was highly optimized in terms of CPU cycles, memory and so on. At this time, it was cheaper to optimize code than to waste CPU cycles. Nowadays, the situation has changed. Computers have enough power and adding more is less expensive than optimizing and personalizing each piece of code.

As software projects got bigger, they needed more time to complete and were thus more likely to have their requirements changed. If the development could not handle these changes, the final solution would not meet the client's expectations. Ledbetter and Cox discussed these problems in 1985 in their article "Software-ICs" published in BYTE magazine [46]. They compared software with hardware to make analogies. Integrated Circuits are one of the successes of modern hardware. Instead of starting over with completely new designs for each type of hardware, vendors use integrated circuits. They come packaged with a fixed interface which allows for their easy integration into many different hardware systems. Introducing a similar mechanism for software is the only way to avoid the software crisis.

According to Ledbetter et al., systems must be built capable of withstanding change. A key concept in building robust software is encapsulation. Message/object technology is the most suited to support this approach. As long as calls are optimized to limit the waste in CPU cycles, the *how* is the most important question. Yet in a message/object system, messages should specify the *what* and leave the details of *how* this is implemented hidden from the user. The specification of *what* corresponds to the interface of the software, i.e. the publicly exposed part. The *how* depends on multiple factors and is hidden from clients. Another key element for reusable software is *inheritance*. When a hardware manufacturer produces a new version of a circuit he takes the current version as the base and adapts the design according to the new requirements. The same is true for software. To implement a solution for a given task, a developer should not start with an empty program but rather look for already existing solutions for problems close to the current one. If such a program exists, the developer can inherit all the parts which do not need to change and concentrate his efforts on the parts which are different.

Today, in a service oriented world, components are a key element. The evolution of remote procedure calls, web services and, in general n-tier architectures is unthinkable without the foundations of software components and component architectures. According to C. Szyperski [B20, p. 3] „Components are for composition“. They support the principle of reusable software and Software-ICs.

6.2.1. Modules and Classes

The terms, *class*, *object*, *module* and *component* are often used in conjunction as by their nature, they all describe different aspects and/or abstractions of the same system. Pascal, as a purely procedural language has no concept of clearly defined interfaces. Listing 6.1 shows an example implementation of a *coffee* data type. The program defines that a coffee has two properties, `coffeetype` and `size`. Additionally, a coffee has a cost, which is a function of its size. Although Pascal can reflect this approach as shown in lines 1

to 11 of Listing 6.1, it is not able to clearly separate the definition of the coffee from its usage in lines 14 to 16.

```
1  program Coffee
2  var coffeetype : string;
3  var size : integer;

5  function getCost(): integer;
6  var
7    cost: integer;
8  begin
9    cost = size*;
10   getCost := cost;
11 end

13 begin
14 coffeetype = 'Dark Roast';
15 size = 3;
16 getCost();
17 end
```

List. 6.1: Coffee with Pascal

In the 1980s, Niklaus Wirth, the father of Modula, added the concept of *MODULE* which, following the software design principles, allows the clear separation of concerns. Each *module* is a unit of compilation and can therefore be converted into byte code independently. Listing 6.2 is again an implementation of a coffee, this time in Modula-2. Compared to the implementation in Pascal, the code is bigger. This comes mainly from the strict separation of concerns. To reproduce similar functionality to Listing 6.1 with Modula-2, three modules are needed. The first, a *definition module*, only contains the contract offered by the future module. This *definition module* is then used by an *implementation module* containing concrete implementations for each defined procedure. Finally, this module can be imported into the *usage module*. From this point on, the `Coffee` definition is available and its function can be called.

```
1  (* Definition of Coffee *)
2  DEFINITION MODULE Coffee;
3  PROCEDURE getCost(size : INTEGER) : INTEGER;
4  VAR coffeetype : ARRAY [0..0] OF CHAR;
5  END Coffee.


8  (* Implementation of Coffee *)
9  IMPLEMENTATION MODULE Coffee;
10 PROCEDURE getCost(size : INTEGER) : INTEGER;
11 BEGIN
12   RETURN(size*3);
13 END;
14 BEGIN
15 END Coffee.


18 (* Usage of Coffee *)
19 MODULE CoffeeShop;
20 FROM Coffee IMPORT getCost;
21 VAR mysize : INTEGER;
22 BEGIN
```

```
23   coffeetype := "Dark Roast"
24   mysize := 3;
25   getCost(mysize)
26 END Coffee.
```

List. 6.2: Coffee with Modula-2

Bertrand Meyer merely speaks of modules rather than components [B16]. When he introduced the Eiffel programming language [B15], Meyer claimed that "a class is a better module" [B17, p. 170]. This is not surprising since Eiffel is an object oriented programming language whereas Modula is not. He stated that software needs to be structured into units to allow: *Reusability*, *Extensibility* and *Modularity*. Meyer motivates these principles through the example of an ADT (Abstract Data Type). The ADT can be any abstract representation of real data, but Meyer's most prominent example is a LIFO (Last-in, First-out) *stack* as shown on Figure 6.1. A user can put elements in the stack, one at a time and later remove these elements, one at a time, starting with the topmost. Suppose that initially, the stack is already populated with two elements, a triangle and a circle. If a user then puts another element, a square in the stack, this element becomes the topmost, as shown in Figure 6.1b. Elements are removed from the stack starting with the topmost. In the case of this small example, the new situation is depicted in Figure 6.1c. The stack again contains the triangle and the circle. Thus, if the user now asks for the topmost element, without removing it, the stack will reply with a circle (Figure 6.1d). This example illustrates well the *Information Hiding* principle. Whereas the client of the ADT only sees the public available methods to put and remove elements, he has no idea how the stack is implemented. It may be done with an array list, a circular buffer, a linked list etc. The biggest part of the implementation is hidden from the user and changing it will not break compatibility with clients as long as the public part remains stable. It is important to note that no client can see or access this hidden part of the module, only the public part is visible and known to clients. This fact is underlined by the *Open-Closed* principle. According to B. Meyer, a module should be open for extensions through inheritance but closed and ready to use for clients. Additionally, B. Meyer supports the concept of (de)-composability. Decomposition is widely used for algorithms (divide and conquer). The same should be true for any component. Through decomposition a complex problem can be split up into several smaller parts. In a module, functions should be used to achieve such decomposition. Instead of having one big function solving the problem, the function can rely on sub-routines. Composability on the other hand, emphasizes the principle of reusability. New modules can be generated by combining already existing ones. According to B. Meyer ADT are realized through classes, in modern object-oriented programming languages [B17, p. 23].

Classes and the fundamental software engineering principles, as formulated by Meyer allow for clean software design. Mainly through the mechanism of inheritance, it is possible to write frameworks where a user only fills in the gaps to have a fully running system. Erich Gamma et al. examined how the mechanisms of modern programming languages help to achieve *Reusability, Extensibility* and *Modularity*. In their book „Design Patterns" [B6], the Gang of Four (Gamma, Helm, Johnson and Vlissides) offer detailed recipes for standard designs, applicable to many real-life situations. Such recipes are called *Design Patterns* and one of them, the decorator pattern, is shown in Figure 6.2. Each pattern uses a unique structure of different classes and a sophisticated collaboration of these to achieve the pattern's goal. Design patterns are the next logical step to

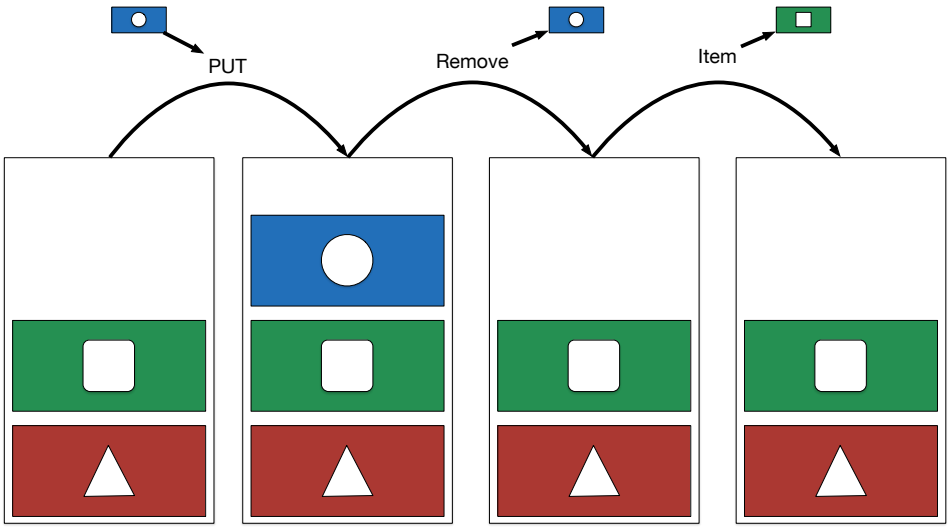


Fig. 6.1.: Abstract Data Type — A LIFO Stack

modules and classes. By combining classes into new structures they form another type of conglomerate. Although patterns neither enhance nor reduce the concept of a unit of compilation, they help to organize them.

Programming languages like Java or C# use yet another mechanism to group classes together, called packages. A package contains a number of different classes evolving in the same universe. In this sense, packages are a kind of module. The classes in Figure 6.2 for example, would all be grouped together into one package. Yet, the notion of module here is somewhat weakened. In contrast to the initial meaning of compilation unit, packages are merely used as a way to separate and control the different namespaces of an application.

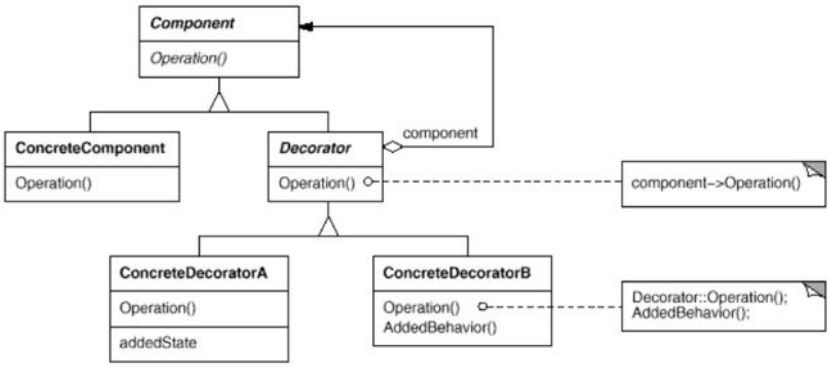


Fig. 6.2.: Decorator Pattern (from [B6, p. 177])

6.2.2. Objects and Prototypes

Besides *classes*, object-oriented programming languages also introduce *objects*. Whereas a class is an abstract description, like the ADT (Abstract Data Type) , objects are concrete realizations. Classes come to life only through instances of them. A class formulates an abstract plan of how such realizations should be created or instantiated. On the other

hand, if no such abstract plan exists but there is a base-object from which all others are cloned, it is referred to as a *prototype object*. Javascript, for example, uses both approaches to create objects. Classes are primarily used to create instances of them, prototype objects to modify objects. Listing 6.3 shows how prototype objects are used in Javascript. In lines 11 and 12, two Coffee objects are instantiated. These objects contain all defined fields and methods from the Coffee class. Line 14 adds a new field to the coffee1 object, *milk* and sets it to be true. From this moment on, coffee1 has one more field than coffee2. This mechanism for modifying objects relies on the prototype object. The reader should note that the class definition uses the prototype approach to define the `getCost` method (lines 7 to 9). Listing 6.3 shows only one way in which objects can be constructed with Javascript. A good overview of best practices regarding prototype objects in Javascript can be found as an ebook online [WEB34]. Although Javascript is the best known prototype-based programming language, it is not the only one following this paradigm. The Smalltalk inspired Self programming language [WEB56] and io [WEB24] are also prototype based.

```
1  // Define the class
2  function Coffee(_type, _size){
3    this.type = _type;
4    this.size = _size;
5  }
6  // Add a method
7  Coffee.prototype.getCost(){
8    return this.size*3;
9  }
10 // Instantiate two object
11 var coffee1 = new Coffee('Dark Roast', 3);
12 var coffee2 = new Coffee('Espresso', 1);
13 // Only this coffee object has the milk property
14 coffee1.milk=True;
```

List. 6.3: Coffee with Javascript

Listing 6.3 shows a class definition with an additional function defined externally. Whereas, this is the agreed way of defining objects in Javascript, it is also possible to embed the function definition within the class. Listing 6.4 shows the same `Coffee` class as in Listing 6.3 with the difference that the *getCost* method is this time embedded. As a consequence, each new instance of this class will hold a copy of the function `getCost`. Yet, possessing copies of a class function's is only required in rare use-cases.

```
1  function Coffee(_type, _size){
2    this.type = _type;
3    this.size = _size;
4    this.getCost = function(){
5      return this.size*3;
6    }
7  }
```

List. 6.4: Internally defined methods in Javascript

Since an object is an instance of a class, it represents an image of something concrete. While classes are immutable, with no starting or end point, objects have a life-cycle. They come to life through their instantiation by a constructor. Through this creation, some memory space is allocated to the objects and its fields are initialized with default

values. From this point on, clients can manipulate the object. During its lifespan, the content an object holds can evolve and adapt to new situations. Imagine a `Counter` class. The class holds a number, the counter and a public method increasing the counter upon each call. Listing 6.5 shows a primitive implementation of this `Counter` class. To increase readability, the class is instantiated and called from the `main` method. Line 14 instantiates a new `Counter` object with a value of `0`. This line start the life-cycle of this object. The next two lines, 15 and 16 both call the `increaseCounter()` function which acts on the `counter` field of the class thus, changing its actual state. Finally, in line 17, the program terminates and the counter object is garbage collected.

```
1  public class Counter
2  {
3    private int counter = 0;
4
5    public Counter(int _counter){
6      counter = _counter;
7    }
8
9    public void increaseCounter(){
10     counter = counter + 1;
11   }
12
13   public void main(String args[]){
14     mycounter = new Counter(0);
15     mycounter.increaseCounter();
16     mycounter.increaseCounter();
17   }
18 }
```

List. 6.5: Simple Counter Class

6.2.3. Components

Depending on the domain of application, the definition of a component varies. A simple query for components on Google returns over 150Mio results. For example, components denote the two chemicals in a *two component adhesive*. In the video world, component is a kind of video input and output connection. Java defines a component as "an object having a graphical representation that can be displayed on the screen and that can interact with the user."[1] In electronics, a component is a part of an electrical circuit. For mathematics, component also has a handful of different definitions and meanings mainly dealing with groups and topologies. Also in computer science, these terms are sometimes misused as in the *component object*[2] [B20, p. 29].

Definition 4 (*Software Components (from [B20])*)

> *A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties. (Szyperski [B20, p. 34])*

Generally speaking, a component is a part of some larger system. Although, it seems evident that the most general definitions of a component also apply to the field of computer

[1] `http://docs.oracle.com/javase/7/docs/api/java/awt/Component.html`
[2] An enumeration object having a unique instance in windows

science a more precise one is necessary. Szyperski proposed in [B20] a well accepted definition (over 500 citations, according to ACM Digital Library) of a component in software engineering.

This definition of a component encompasses four concepts:

- **Interfaces:** each component has a well-defined and stable interface, an API over which it can communicate with other components or pieces of software. This interfaces allows the decoupling of the client and the component. As a side effect, a component can be re-used for other clients. Additionally, as long as the interface does not change, the inner guts of a component may change, generally without affecting the clients.
- **Explicit Context dependencies:** a component needs to explicitly define its requirements for a suitable deployment environment. These needs are a component's context. Although the provided interfaces are generally well documented, this is not the case for the component's context. Even today this is not true for all major components. Java components living in a Java application server have no means to specify their needs. Developers have to thoroughly document their components in terms of needed databases, security containers etc.
- **Reusability:** Cox et al. mandate since the mid 80's for re-usable software, called Software-ICs [9, 46]. By comparing software developers with hardware vendors they concluded that the only way to avoid a *Software Crisis* is to build reusable software. Obviously the best component is the one with no context dependencies and provides the most suitable service. But this would defeat the component approach. On the other hand, the more a component is based on reusable software, the more context dependencies it needs. Since all these dependencies evolve separately, this lowers the components robustness. Figure 6.3 described this situation. Finding the right balance between robustness and dependencies always relates to the use-case.
- **Lifecycle:** Each component has a lifecyle. Depending on the current state, a component may not have its full capacity. The lifecycle is generally provided and supported by the container running the component.

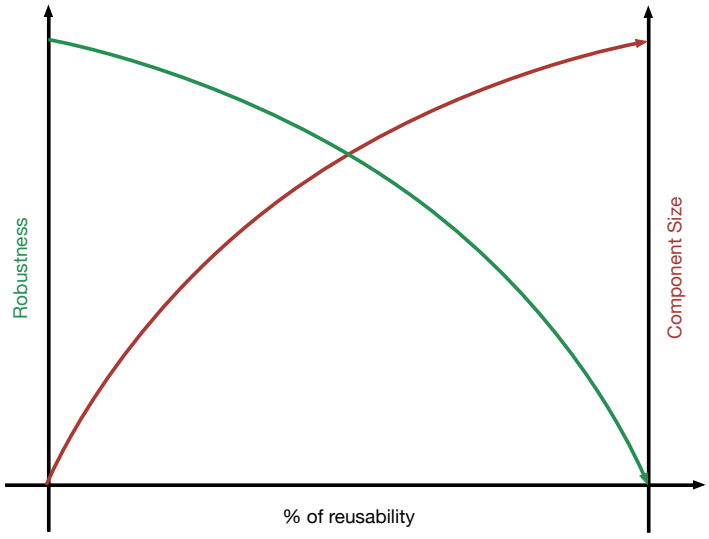


Fig. 6.3.: Robustness vs. Size of Component

The remainder of this chapter focuses on two properties: (1) Interfaces and (2) Lifecycle, which are both of special interest for Part III of this thesis.

6.3. Interfaces

Interfaces are part of the specification of a component. In Bertrand Meyers ADT, they represent the visible part of the iceberg (Figure 6.4). Therefore, the interface can be seen as a contract. For components, this contract binds two parties: the component as service provider and the client as service consumer. Such an interface contains the set of operations which can be called by the client. Additionally, each operation depends on a set of pre-conditions and implies a set of post-conditions. The client is in charge to prepare the system so that it fully complies with the set of pre-conditions. On the other hand, the component responds with a result plus some post-conditions. Usually, the pre-conditions are established before the operation starts and the post-conditions become true before the program returns, if it returns at all. Hoare's logic [32] proposes a formal way to prove the correctness of computer programs. This logic also allows the expressing of interfaces through *Hoare triples* in the form of

$$\{P\}S\{Q\} \tag{6.1}$$

Where $\{P\}$ is the set of pre-conditions which have to be true before the execution of the statements S. If the execution of S terminates, then the set of post-conditions $\{Q\}$ becomes true. These triples ensure *partial correctness*. Partial since they do not ensure that the execution of S terminates. If S eventually terminates, then this would even lead to *total correctness*. However, to prove termination of S additional proofs are required. Equation 6.2 shows a few Hoare Triples as defined in Equation 6.1

Definition 5 (*Predicate*)

Given the function P defined as $P : x \to \{true, false\}$ $\forall x$ then, P is called a predicate.

Definition 6 (*Hoare Triple*)

If P and Q are two predicates, then the equation $\{P\}S\{Q\}$ holds. Therefore, if in the beginning P is true then, after the execution of S, Q is also true.

One can now easily understand that the following triples are *Hoare Triples*. The first example states that if the variable x evaluates to 10 and if the variable y evaluates to 2 then, after the execution of the statement $z := x/y$, the variable z evaluates to 5. In this case the assignment $z := x/y$ evaluates to $z := 10/2$ and thus sets the variable z to 5. Therefore, the set of post-conditions is true and the logic holds. The same reasoning can be applied to the two other examples.

$$\begin{array}{rrcc} \{x = 10 & \&\& \quad y = 2\} & z := x/y & \{z = 5\} \\ & \{true\} & a := 1 & \{a = 1\} \\ & \{y! = 0\} & z := x/y & \{z = x/y\} \end{array} \tag{6.2}$$

Regarding interfaces, the set $\{P\}$ represents the pre-conditions established by the client before calling the interface. Although this set can be any predicate usually, for an interface, this means that all the needed input parameters are ready and respect the interface needs. For example, for an integer division the definition of the interface would look like:

```
/**
* Integer division function. The functions takes two integers,
* numerator and the divisor and return the result of the integer
* division of the two numbers.
*
* @params _numerator any integer value.
* @params _divisor any non-zero integer value.
* @throws An IllegalArgumentException is thrown if the divisor is zero.
* @returns result of the integer division of the arguments.
**/
public int divide(int _numerator, int _divisor) throws IllegalArgumentException;
```

From this listing, a client can conclude several facts, the most obvious being, there exists a function called `divide` which divides two integers, returns another integer and sometimes throws an exception. Most programming languages don't allow any further specification of pre- and post-conditions. Java circumvents this limitation with javadoc annotations to each function and class. Unlike normal comments, javadoc introduces special documentation fields which can be used to signal pre- and post-conditions of a method. The `@params` annotation describes the input parameters, `@returns` describes the return value of the function. Although this is not a formal description, a client can deduce the corresponding Hoare Triple. For the `divide` function this would be:

$$\{\text{numerator} \in \mathbb{Z} \land \text{divisor} \in \mathbb{Z} \backslash \{0\}\} \quad \texttt{divide} \quad \{\text{true}\} \tag{6.3}$$

Looking at a typical implementation of an integer division as given in Listing 6.6, proves that the Hoare triple as deduced in Equation 6.3 is correct. The function accepts all integers and produces an error if the divisor is zero. Furthermore, the function has no post-conditions.

```
public int divide(int _numerator, int _divisor) throws IllegalArgumentException
{
  if(_divisor == 0)
  {
    trows new IllegalArgumentException("Cannot divide by 0");
  }
  return _numerator/_divisor;
}
```

List. 6.6: Integer division implementation

The interface as a contract between a service and its clients implies that it should not change over time. If the interface changes, some clients may stop working. However, it is unlikely that a component will stop evolving once it has been published and used by some clients. Similarly, it is unlikely that all clients will adopt a new version of a component the day it is published. To overcome this limitation, the contract can introduce versions. Prefixing the contract with a version number allows different versions of the same component to run in parallel. As already discussed in Subsection 6.2.1, one way to introduce versioning is packages. Packages provide a sort of namespace for the classes they contain. Accordingly, it is possible to define a class `Shapes` in the package

`ch.unifr.softeng.v1` and another, newer version in the package `ch.unifr.softeng.v2`. A client can then choose which version to use and old clients can continue to use old versions.

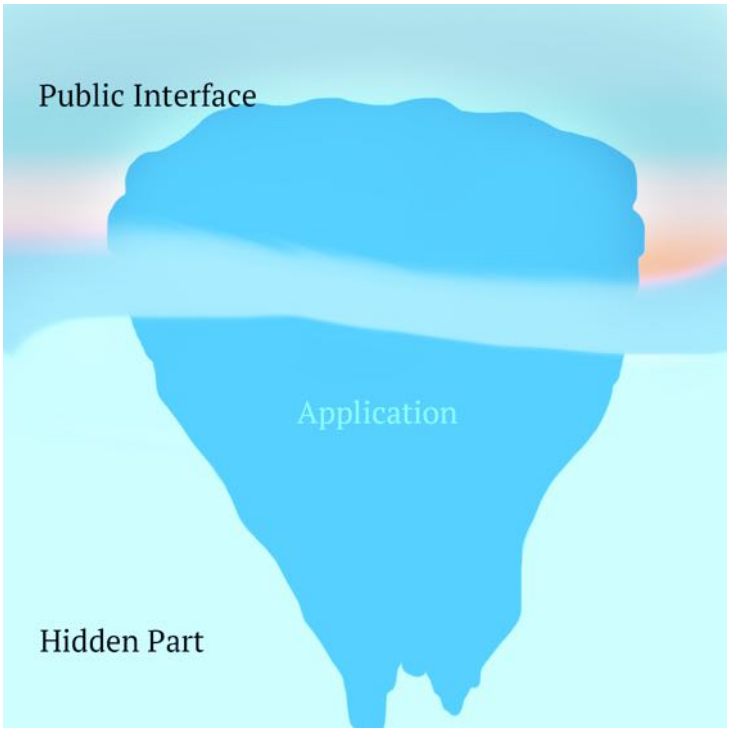


Fig. 6.4.: Visible and Hidden part of Software

Even if changes introduced with the new release of a component do not modify the interface, clients can still break. This can be the case if an algorithm needs more time to complete or if a stream has increased capacity. In the first case, clients could interpret the increased waiting time as time-out and abort the operation. In the second case, slow clients can be overwhelmed with the quantity of information delivered by the service. Therefore, it is good practice to split the evolution of a component into versions.

So far, only functional requirements have been discussed and how they are translated into the interface of a component. Yet, most software also comes with some non-functional requirements. These range from security requirements to pricing models or QoS (Quality of Service) restrictions. Although the topic is not new, today there exist still no standard for how such constraints can be defined in the interface. In [37] the authors propose to standardize the way Quality of Service (QoS) related aspects should be integrated into UDDI and SOAP(Simple Object Access Protocol) messages. This suggestion is partially based on work done by Ran Shuping [67]. With the introduction of web services, these non-functional requirements gained in momentum. Mani and Nagarajan, two software engineers at IBM had already hunted down the problematic of QoS for web services back in 2002 and identified the fundamental requirements for supporting QoS [WEB67].

6.4. Life Cycle

The definition of a component given in Subsection 6.2.3 pointed out four concepts, among them, the concept of a component's lifecycle. This is not only true for components but for

any product. Clothes have a lifecycle, fruit and vegetables follow one, software products also are supported by a lifecycle. They all have in common that a lifecycle is an ordered sequence of transitional states. Some lifecycle are true cycles, whereas others go from an initial state to a final one, traversing several intermediate states. Generally, we can identify at least one initial state. The transitions between the different states are triggered by events. Depending on the event, the next state might not always be the same. A simple example is the lifecycle of a bottle of wine. First, the grapes are picked and delivered to a winery. There the grapes are sorted, washed and destemmed. The fruit is then pressed and the juice (for white wine) is stored in large barrels where it goes through primary fermentation. After this first process, the juice is generally transferred to new barrels where a secondary fermentation process starts. Depending on the kind of wine, after the second fermentation it has to stay in barrels (sometimes wooden) to mature. Before the wine is put into bottles it is sometimes refined by mixing different wines together. This process is called blending and ensures that the wine tastes as expected and small inconsistencies in taste can be corrected at this stage. Finally, the wine is bottled and ready for consumption. Figure 6.5 shows this lifecycle and the related states from harvest (initial state) to consumption (final state).

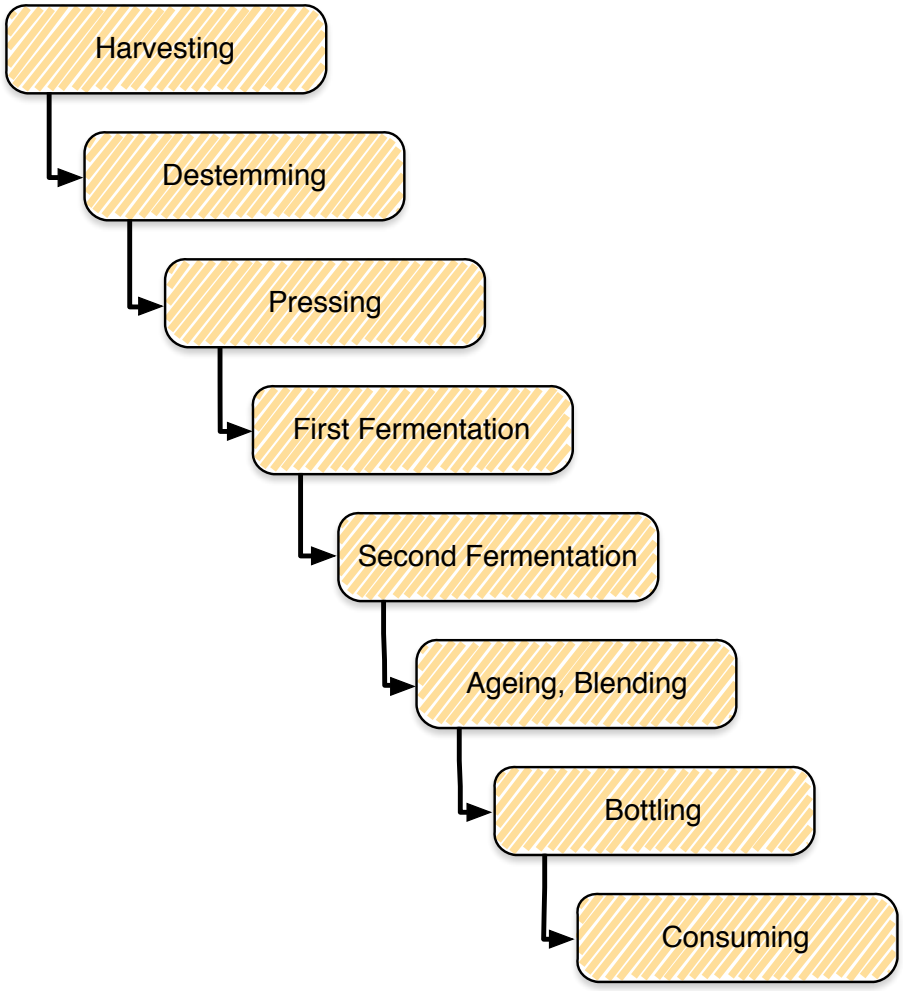


Fig. 6.5.: Vinification lifecycle

In a similar manner, software has a lifecycle. In its simplest form, it may be *development — operation — end-of-life*. Yet, when studying the software development process, it appears that it is much more complicated than just *development*. The same applies to *operation*. Services don't simply run; but they have a well defined lifecycle. Services at most times need some supporting infrastructure and this infrastructure also dictates the lifecycle of the supported content. This is already true for very simple services. A small webpage containing only HTML code already needs some infrastructure to allow clients to browse it. A server like Apache [WEB3] provides simple HTTP interactions and acts as a mediator between the managed pages and clients wanting to access to this content. The growth in complexity of the provided service is directly coupled with the increase in complexity of the supporting infrastructure. This raised complexity also introduces bigger lifecycles.

According to Definition 4 components are deployed in a context and have a lifecycle which not only depends on the components, but also on the context they are deployed in. Nonetheless, the differences between different contexts are small. In the Java world, the Jetty server is an example of such a container. But, there are a number of others, providing different levels of functionality. As such, Glassfish or JBoss are fully blown application servers whereas Jetty or Tomcat are generally called web-containers. It is however amusing to note that fully blown application servers like Glassfish rely on the smaller web-containers to provide web functionality. Thus, in the end their lifecycle will be quite similar.

Writing *Servlets* is one way to exploit the capabilities of an application server. A servlet is a Java class implementing either directly or through heritage the `javax.servlet.Servlet` class. The best known implementing classes of the `Servlet` interface is `HTTPServlet`, which already provides methods to handle the standard HTTP calls and the `GenericServlet` which serves as a skeleton for protocol-independent servlets. By implementing the `Servlet` interface, each class also inherits the servlet lifecycle.

Parts of a servlet's lifecycle are imposed by the supporting application server and parts by the servlet technology. The first step in this lifecycle is to *deploy* the compiled package to the application server. Through this process the application is configured and enabled. For this to happen, the application is bundled with a descriptor containing instructions for the application server about needed resources like a database connection or a security framework. If the application passes this stage it can be *started*. During the starting phase, the container initializes the servlet. This involves the injection of needed dependencies and the creation of a new instance. After the successful creation, the servlet is said to be *ready* to serve incoming requests dispatched by the container. If at some point, the servlet is shut down, the container calls up the servlet's destroy method. The servlet is then ready to either be started again or to be undeployed.

Such a container cannot only deploy simple servlets but full components. In the Java world, these components are called *Beans*. Generally speaking, a servlet is just a simple controller. Incoming requests are dispatched from the container to the servlet which then decides what to do with the request and what response to send back. Beans, on the other hand, contain reusable parts of software and are also deployed in some container. There are several types of beans, stateless-session beans, stateful-session beans, message beans and singleton-session beans. All have slightly different lifecycles due to their nature. The lifecycle of a stateful session bean is a good example of all types of Java beans. Figure 6.6 gives a rough overview of its lifecycle. In the beginning, the bean does not exist. The

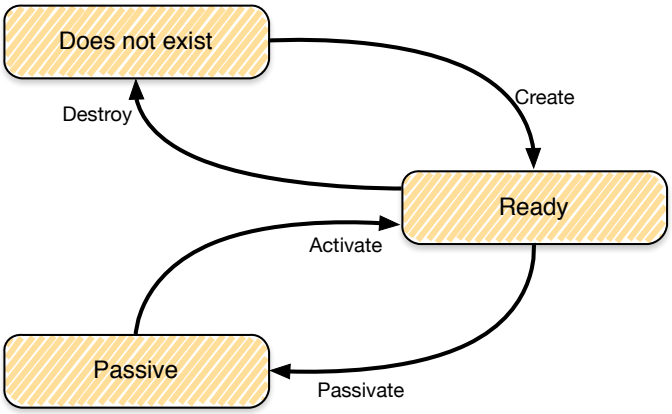


Fig. 6.6.: Lifecycle of a Stateful Session Bean

application server is then responsible for injecting the needed dependencies and creating a new instance by calling the bean's `init` method. After a successful creation, the beans is *ready* to receive calls. However, the application server may decide at any time that the bean is currently not needed and put it into a *passive* state and call the bean's `@PrePassivate` method to move it from memory to storage. As soon as a client invokes any method on the bean, the container wakes the bean up by calling its `@PostActivate` method. In the end, if the bean is not used anymore, it can be disposed of. To do this, the container calls the `@Remove` and the `@PreDestroy` methods. Through this last action, the component is again in its initial state *Does not exist*.

Different types of beans have slightly different lifecycles but, the main part of the lifecycle is enforced by the supporting container. This also means that a given container may not be adapted to every sort of component. Some, may require other lifecycles and therefore other containers. It is, however, important to note that the lifecycle is always the result of a component's requirements plus a container's capability.

6.5. Key Concepts introduced in this Chapter

This chapter introduced the notion of software components. The need for components is motivated by the *Software Crisis*. Starting with languages like *Pascal* which are unable to separate the definition from its implementation and its use, the evolution to modules as defined by *Modula* and finally real objects as we know them from modern languages like *Java* was illustrated.

Upon its introduction, Pascal was quite popular and was even forked into the commercial *Borland / Turbo Pascal*. At this time there was no need for re-usable software. Each piece of code was self-contained. Soon, industry realized that this approach was not viable. The need for reusable software or at least in part, was realized. Modula introduced a few novel concepts allowing the separation of concerns. Interface definitions were separated from their implementation and from their usage. However, this was still not enough to solve the software crisis. Although, Modula isolates definitions from their implementation and usage, this mechanism is more like a namespace, far away from real objects and classes. Finally, object-oriented languages like Eiffel opened the door to real software

components. Objects and classes, as introduced by Meyer, are open for extension and closed for modification. Therefore, they form compilable units, ready to be used by clients.

Today a similar situation exists for the IoT, where no standards are imposed, neither regarding the outer interface of a smart device nor its inner structure. Although the WoT mandates the use of RESTful interfaces, it also faces the same problems. Each smart device is unique in its inner and outer structure making it difficult to deploy hundreds or thousands of units and use them in different scenarios. Additionally, there are no guidelines on how to structure an entity, making it quite difficult to re-use an already deployed smart device in a different scenario. This leaves a situation where smart devices are closed for modification (at least partly) but, definitely not open for extension.

The aim of the xWoT meta-model is to help developers create well-structured, deployable and easily reusable components. Having a component based approach for the xWoT leads to such independently deployable units as mandated by Definition 4. Our xWoT components have a well-defined interface supported by RESTful web services plus some additional semantics, presented in Part III. This makes them ready to use in many scenarios. Coming back to Meyer's point of view, these modules are closed for modification. On the other hand, by adopting the structure imposed by the meta-model, they become open for extension. Although not specified explicitly, adopting REST as the outer interface makes the Web a context dependency of smart devices and serves as a deployment environment. The web also imposes its lifecycle, at least partly, on xWoT components. Moreover, with the aid of the meta-model introduced in Part III, these components become easily reusable.

7
Meta-Model

7.1. Introduction

Components are one key factor for successful software engineering. They allow for a great re-usability of code. Yet, to maximize the usability and also the re-usability of such components, they need to be built carefully. Models and modeling tools support the developers during the planning phase and greatly impact on the quality of the final product. Furthermore, developers, business analysts and clients need some communication interface over which they can share specifications and requirements in a standardized way.

The first part of this chapter discusses the key elements involved in modeling from a software engineering point of view. This encompasses the four layers involved, starting with endeavors and going through models and meta-models. The last layer, meta-meta-modeling, is outside the scope of this thesis and is therefore simply ignored. The second part introduces some tools supporting the creation and utilization of meta-models with eclipse. Based on the first two parts, the last part fully develops a simple example to show the application of models and meta-models in a concrete situation.

7.2. Definitions and Vocabulary

Modeling is commonly used to capture requirements before a project starts. It is also often used to document the current state of a project. Furthermore, models serve as

decision-base and common language between developers, requirements engineers and the client. To reach this level of abstraction and platform independence, models need a common, well-defined language. This is what meta-models are about. They define how the elements of models are named, how they are structured and what relationships they share.

In the field of software engineering, software development methodologies is a big field of interest with a long history. Since the early beginnings, best practices exist for different situations to guide the developers through the process of development. The gang of four, proposed a collection of models, called design patterns, tailored for a variety of common situations in software engineering. It is common sense that these models together define a methodology in this context. In fact,

- *WordNet* defines a methodology as „the system of methods followed in a particular discipline“ or as „the branch of philosophy that analyzes the principles and procedures of inquiry in a particular discipline“ [WEB47].
- *Dictionary.com* gives three meanings for methodology: The first is „a set or system of methods, principles, and rules for regulating a given discipline, as in the arts or science“ [WEB12]. The second definition applies to philosophy whereas the last is about education.
- The *Encyclopedia Britanica* gives yet another definition. Accordingly, a methodology is either „a body of methods, rules and postulates employed by a discipline“ or „the analysis of the principles or procedures of inquiry in a particular field“ [WEB33].

From these definitions two main streams can clearly be identified. On the one hand, a methodology is a collection of methods for a given field of interest. This is how most people would probably define methodology. The methods describe how something can be done. Like this, the collection is source of information. On the other hand, a methodology can describe the study of procedures applied in a given discipline. Etymologically, the second definition is more accurate as it is directly be derived from the Greek word metodologia ($\mu\epsilon\theta o\delta o\lambda o\gamma\iota\alpha$) which in turn is derived from the two ancient Greek words methodos ($\mu\epsilon\theta o\delta o\sigma$) and logos ($\lambda o\gamma o\sigma$). Therefore, the etymological meaning is *study of knowledge*. Whereas the first definition is rather passive and a methodology is characterized as a tool, the second one implies a mental activity. The dictionaries and encyclopedia cited above agree that this second definition is mostly used in philosophy and philosophical reasoning. Although, in the field of software engineering, the process of modeling implies cognitive skills, its meaning is closer to the first definition. Consequently, this thesis adopts the definition given by Gonzalez-Perez et al. in their book *Metamodeling for Software Engineering* [B9].

Definition 7 (*Methodology (from [B9])*)

A methodology is a systematic way of doing things in a particular discipline [B9, p. 3].

Additionally, method is just a synonym of methodology. Definition 7 contains 4 concepts:

1. It is *a way* of doing something; a methodology is not a goal but its application leads to an end. A methodology is a tool.
2. The application of a methodology is *systematic*. This ensures that results are predictable and most importantly, repeatable.

3. It is about how *things are done*. This means that a methodology transform things. The state before applying the methodology is not the same as the state after.
4. It applies to a *particular domain*. Methodologies are not general truths instead, they are tailored for a precise domain. In this case, the domain is software engineering and, as depicted in Section 8.2, the Web of Things.

Methodologies are not transcendental; humans define them specific to a domain of application. There are two major approaches to how new methodologies can be created: (1) *Tailoring* is a widely used approach in software engineering, mainly through the mechanisms of inheritance supported in various programming languages. An already existing generic methodology is customized to the current needs. To adapt a methodology to new and changing situations, some ancestor methodology needs to exist. Since this ancestor serves as a basis for new methodologies, it is composed of two parts: a fixed and a variable. The fixed part remains unchanged during adaptations whereas the variable part changes with each new methodology. This duality is not without some problems. The creator of the a-priori methodology has to decide which parts are fixed and which variable. Nonetheless, finding the line between the two is a difficult task. If the methodology has too many fixed parts it can only serve as a-priori methodology for a very restricted number of situations. If on the other hand, the a-priori methodology only contains variable parts, it will not be a good ancestor for any new methodology. (2) *Method engineering* on the other hand dynamically composes new methodologies out of existing ones, and is proven to be useful and stable. There is no generic template as for the tailoring approach. Instead, methodologies are built onto what already exists. This approach highly supports the re-usability of methods. Additionally, method engineering has the advantage that nobody has to decide about a static part. If a method component is useful, it is taken as it and embedded with other method components. It is important to note that this second approach asks for a new kind of engineer, a method engineer. Someone has to build these method components and make them available from a repository. Starting from a given problem, these people browse such a repository and if no adapted methodology is found, create one adapted to their current endeavor. Subsequently, software developers build on top of these methodologies. They use them to create new software. Thus, methodologies are a common playground for both, method- and software engineers. Hence, both should be able to read, write and interpret methodologies.

Whether tailoring or model engineering is used to create new models, they need to be expressed somehow and for this, they need a representation. One could imagine that a metamodel is a kind of special model and indeed, the "meta" of metamodel means beyond in Greek ($\mu\epsilon\tau\alpha$). More usually, the prefix "meta" is used in philosophy to indicate that something is about its own category. Therefore, a metamodel is a model about models. This is also the definition found in the glossary of the MDA Guide [54]. Although simplistic, this interpretation is already quite close to what metamodeling is about. Definition 7 states that a methodology is a model, therefore, the process of creating models is called modeling. By extension, the process of creating the language supporting the process of modeling is called metamodeling. Definition 8 gives a formal definition of the term metamodel as it will be use throughout the rest of this thesis.

Definition 8 (*Metamodel (from [B9])*)

> *A* metamodel *is a domain-specific language oriented towards the representation of software development methodologies and endeavors [B9, p. 18].*

7.2.1. Endeavors and Systems under Study

The definition of a metamodel (see Definition 8) introduces the term endeavor. Modeling is the task of describing with a given language parts of reality. For the model, this part of the reality is called the *System under Study (SUS)* or *endeavor*. Basically, anything can be a SUS (System under Study) , a tree, the weather or software. As long as it is observable, it can be a SUS. The same reasoning applies to models: when building models of a SUS, they become real and part of a reality. Models become observable and can therefore turn into a SUS.

Moreover, Definition 8 explicitly asks for a language describing the model. The semiotic triangle [58] (also called Ullmann's Triangle [B22]) summarized by Guizzardi [26] on Figure 7.1 resumes the relationship between SUS and models. The cognitive model abstracts a SUS. Although, details of the SUS get lost through the process of abstraction, the key properties remain. This new, simpler form of the initial SUS facilitates reasoning. Such an abstraction can be expressed with a function α mapping one representation (the SUS) into a new one (the cognitive model). Let α be a mapping from the set S of SUS into the set M of cognitive models

$$\alpha : S \to M \tag{7.1}$$

then, the following relation 7.2 holds.

$$\exists \quad \mathrm{SUS}_1, \mathrm{SUS}_2 \in S | \alpha(\mathrm{SUS}_1) = \alpha(\mathrm{SUS}_2) \implies \mathrm{SUS}_1 = \mathrm{SUS}_2 \tag{7.2}$$

Basically, the function α is a one-to-many mapping. Each model can have several concrete realizations (SUS). Yet, the set M does not contain real models which can be communicated to others; it merely consists of visualizations of SUS as the person creating models imagines a SUS. To communicate these visualizations with others, they need to be expressed in some common language. The model, by using such a language, reifies the cognitive model into a model (also called communicated model). This relationship is expressed with δ, a mapping between the model and the cognitive model. It is important to note that there is no direct connection between the SUS and the model. Although the model represents the SUS, they are only linked through the cognitive model of the creator, albeit, the relation μ between a model and a SUS is the composition of the relation α and δ ($\mu = \alpha \circ \delta$) in software engineering μ is often equated with α.

7.2.2. Models

Although, model and SUS are not directly connected in the semiotic triangle (see Figure 7.1), one can still generalize that a model is an abstraction of an endeavor. Gonzalez-Perez formulates three criteria such a model must meet:

1. *Abstraction:* the model is a simpler form of the SUS, sharing the essential parts but lacking the complexity of the SUS. Accordingly, a model can stand for several endeavors.
2. *Homomorphism*: any operation on the SUS is also possible on the model. Therefore, to solve the SUS, it is sufficient to solve the model.
3. *Purpose:* A model represent its endeavors and acts on behalf of them.

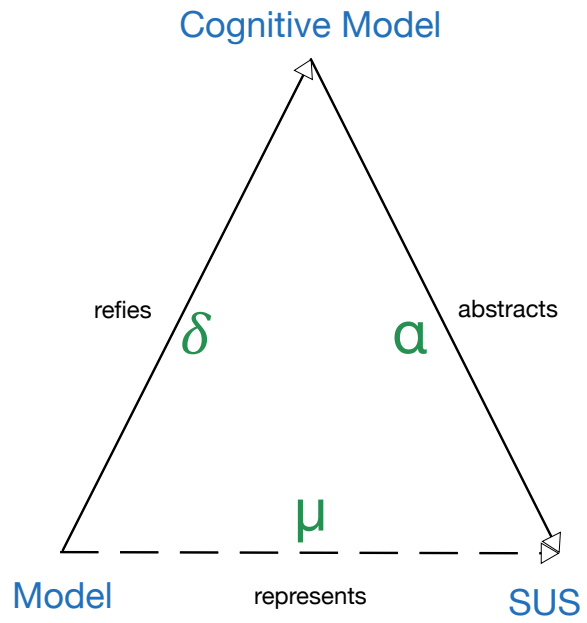


Fig. 7.1.: The Semiotic Triangle merged with the Seidewitz' terminology (after [B10, p. 4])

To fulfill these requirements, a model always needs to be connected to some SUS. It cannot stand for itself alone and is useless without any endeavor. Gonzalez calls this link from the model to the SUS *interpretive mapping*. He means that any entity of the model can be mapped to an equivalent entity of the SUS. However, these mappings are not always one-to-one and depend on the characteristics of the entity. A car for example, can be represented with a model. The latter, although a simplified version, contains all the important parts, like wheels, the engine, headlamps etc. Indeed, if the model contains a steering wheel, there is a one-to-one mapping to a steering wheel of a real car. A bag in the cargo bay of the model, on the other hand, is a one-to-many mapping. It is only in the model to illustrate that bags can be stored in the cargo bay. But, it does not map any particular real bag. Instead, different ones can be represented by this model bag. Therefore, interpretive mappings can be off two different types: one-to-one and one-to-many. Gonzalez-Perez enumerates three categories of interpretive mapping. The one-to-many mappings can be of two different types: examples and property sets. This leaves the following types of interpretive mapping:

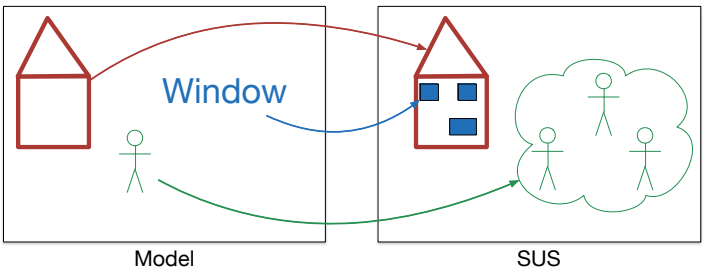


Fig. 7.2.: Different types of mappings. In red, an Isotypcial mapping, in green a Prototypcial mapping and in blue a Metatypical mapping.

1. *Isotypcial* mappings are one-to-one. Each model entity maps directly to a SUS entity.
2. *Prototypical* mappings map one model entity to many SUS entities. The model entity is expressed as an example which stands for a whole category of entities.
3. *Metatypical* mappings are also one-to-many. However, instead of specifying the category of entities through an example, it is specified declaratively. The model entity contains a set of properties which must be true for any mapped entity of SUS.

The car example has already used two of these categories: first, the steering wheel is an isotypical mapping between the model and the SUS and for each steering wheel in the model, there is exactly one steering wheel in the car; second, the bag model in the cargo bay is an example of a prototypical mapping standing for different realizations of a bag in reality and the class of real bags is specified through an example bag. Metatypical mappings, the third type, can be found in UML (Unified Modeling Language) class diagrams. Each class definition stands for a collection of objects in the system. In UML these objects are represented through one class with a set of properties in the form of field and function definitions.

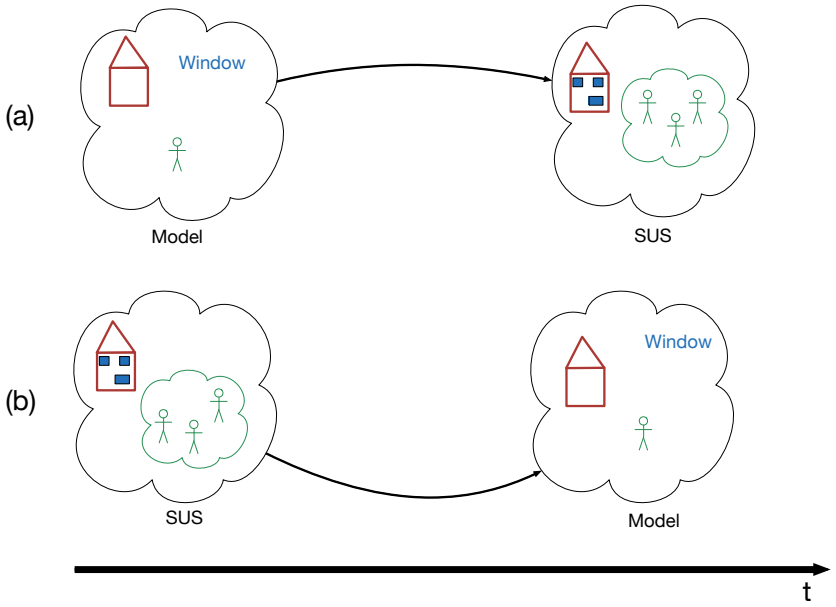


Fig. 7.3.: Temporal relationship between models and SUS. The situation (a) shows an analysis whereas (b) depicts a specification.

Along with the type of mapping between a model and its associated endeavors, there are different types of model. Depending on the situation, a model can exist either after the SUS or before the SUS. The first is useful to abstract from a concrete situation and to reason about a given problem. The latter is merely used in computer science to specify how a system should work before actually implementing it. Depending on the author, a more or less fine-grained partitioning of models is given. Yet, all agree on these two categories. Gonzalez-Perez calls them *backward-looking* if the model exists after the SUS and *forward-looking* otherwise [B9, p. 23]. Henderson-Sellers merely speaks of *analysis* and *specification* model [B10, p. 39] and others ([68, p. 13], [30, p. 389]) talk about *descriptive* and *prescriptive* ones.

7.2.3. Meta-Models

Everything discussed so far also applies to meta-models. As models are an abstract view of an endeavor, meta-model are an abstract view of models. Hence, the semiotic triangle [58] can be applied to models. Whereas, in its original form, an endeavor is the SUS, now the model becomes the SUS through a process of abstraction; somebody can create the equivalent of a cognitive model. Henderson-Sellers calls this a *meta-conceptualization*. As in the semiotic triangle, this is just a theoretical representation of the models. To share this illustration with the world with the help of a language, a meta-model is reified from this meta-conceptualization. Figure 7.4 resumes this situation. Again, three mappings are used to transform real world models into meta-conceptualizations and finally, meta-models.

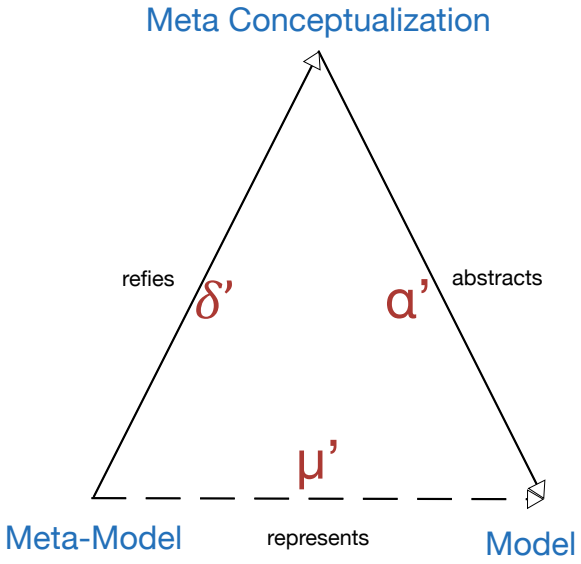


Fig. 7.4.: Semiotic triangle applied to meta-models (after [B10, p. 4])

As for the relation between SUS and models, there is no direct relation between models and meta-models. The latter only exist through a meta-conceptualization expressed in some language. Therefore, the *represents* relation can be expressed as follows:

$$\mu' = \alpha' \circ \delta' \tag{7.3}$$

In software engineering however, the relation δ' is most times ignored, defining μ' as:

$$\mu' = \alpha' \tag{7.4}$$

This defines the meta-model as an abstraction of the model. In Equation 7.1 of Chapter 7.2.1 the *abstraction* function α is defined as a mapping from a set S of SUS into a set M of models. The model abstraction function α' is defined in a similar manner:

$$\alpha' : M \to MM \tag{7.5}$$

where M is the set of models and MM is the set of meta-models. Since, meta-models are an abstraction of models the following equation also holds:

$$\exists \quad m_1, m_2 \in M | \alpha'(m_1) = \alpha'(m_2) \implies m_1 = m_2 \tag{7.6}$$

Meta-models and their models need to be homomorphic. All important structures of the model are also present in the meta-model, allowing reasoning about the model with the meta-model. Mathematically, a homomorphism is a relation between two groups

Definition 9 (*Homomorphism*)

Let A and B be two groups, and f a map from A to B. Then f is said to be a homomorphism *of A into B if the following property holds for all x*

$$\forall x, y \in A, f(xy) = f(x)f(y)$$

Based on this definition we can introduce the following theorem

Theorem 7.1

Let $f : A \to A'$ and $g : A' \to A''$ be two homomorphisms. Then the composite map $f \circ g$ is a homomorphism from A into A''.

The proof holds on a few lines. The interested reader can find them in Lang's Undergraduate Algebra [B13, p. 34]. Theorem 7.1 applied to SUS, models and meta-models leads to

$$\alpha : S \to M, \text{and } \alpha' : M \to MM \Rightarrow \alpha \circ \alpha' : S \to MM$$

Through this relation, a meta-model, although abstract, still shares — to some extent — important aspects of the endeavors of a given domain. Therefore, as discussed in Definition 8, a meta-model is always domain specific.

This approach gives a three layered hierarchy where the endeavors sit on the bottom layer. Subsection 7.2.2 showed that endeavors are created from models and methodologies. For this reason, it makes sense to define models as the next layer, sitting on top of endeavors. This produces a hierarchy where models serve as templates for endeavors. From the beginning of Subsection 7.2.3 and from Equation (7.5) models are derived from methodologies. Accordingly, they should form the next layer. Following the semantics of this hierarchy, it defines methodologies as descendants of meta-models which is compatible with Equation (7.5).

Yet, this reasoning does not explain how meta-models are generated. Following the same approach, there must be an additional layer serving as template for creating new meta-models. Figure 7.5 shows that the OMG (Object Management Group) introduced such a fourth layer called the meta-meta-model in their hierarchy [59, p. 35]. Furthermore, the OMG defines the relation between a layer and its parent layer as type *instance-of*. This means that objects living in the endeavor layer are instances-of the model layer. The OMG calls this architecture Meta Object Facility (MOF) with a meta-meta-model in its center. In addition to the four layers of Figure 7.5, MOF (Meta Object Facility) also defines some standards for how to handle models. Of them, the XMI (XML Metadata Interchange) format is used to exchange metadata. Besides the MOF architecture, the OMG also proposes a smaller one, EMOF (Essential MOF) as a subset of MOF. EMOF

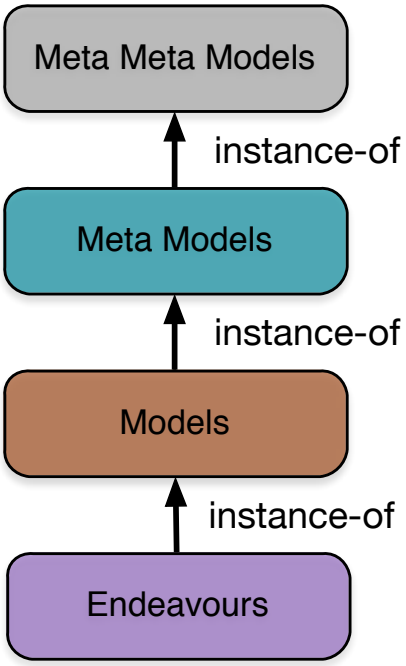


Fig. 7.5.: The OMG four layer approach to meta-modeling

is an important subset of MOF in the sense that it is fully compatible with Ecore, the core of the Eclipse Modeling Framework (EMF).

Based on these considerations and analogous to the definitions of α and α' in Equations (7.1) and (7.5), we can define another map α'' as

$$\alpha'' : MM \to MMM \tag{7.7}$$

Applying Theorem 7.1 once again, to the maps α, α' and α'' as defined previously results in the following equation:

$$\begin{aligned} \text{Given} \quad & \alpha : S \to M, \text{and } \alpha' : M \to MM, \text{and } \alpha'' : MM \to MMM \\ & \Rightarrow \alpha \circ \alpha' \circ \alpha'' : S \to MMM \end{aligned} \tag{7.8}$$

On a more abstract level, the OMG calls these layers $M0$ to $M3$, $M0$ representing the endeavors layer and $M3$ the meta-meta-models layer. Given these notations, the Equations (7.1), (7.5) and (7.7) can be rewritten as follows:

$$\begin{aligned} \alpha : & \quad M0 \to M1 \\ \alpha' : & \quad M1 \to M2 \\ \alpha'' : & \quad M2 \to M3 \end{aligned}$$

In *Metamodelling for software engineering*, Gonzalez-Perez presented yet another view

of meta-models. According to Gonzalez-Perez, the building blocks of models are *model units*, the smallest and indivisible parts of a model. Each model unit stands for one major characteristic present in the SUS. As explained, there is a homomorphism α between a SUS and its model. Model units form the basic elements this map is working on. Consider the car example introduced in Subsection 7.2.2; the steering wheel, brakes, engine and doors would be model units to model cars. Sometimes, a model is composed of several similar model units. A car, for example has three or five doors, each being a model unit. Although each door is different in the model (it is impossible to exchange the driver's door with the rear door), they all share certain properties. If several model units share a set of properties, they are of the same *model unit kind*. Both, the driver's door and the rear door are doors which can be opened and closed.

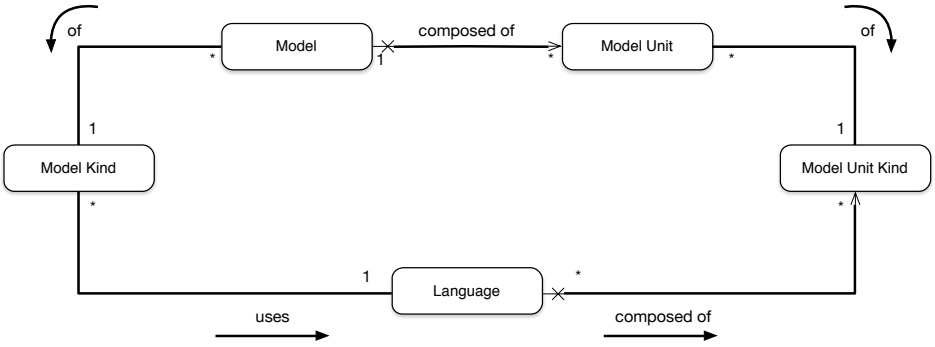


Fig. 7.6.: Another view of meta-models (modified from [B9, p. 28]).

Models, on the other hand, are specific for to a given domain. Consequently, each model is of a given *model kind*. For example, with the example model units, steering wheel, breaks, engine etc. it is possible to model cars, trucks, buses, tractors etc. All these models share, that they represent some type of automobile. Accordingly, their model kind would be the automobile. Just as models use a set of model units, a model kind uses a set of model unit kinds. Since a model unit kind is, although abstract, related to a domain, this set only contains model units kinds with some relationship. To continue with the car example, it would not make any sense for a model kind to use the *class* or *attribute* model unit kind. These ordered sets of model unit kinds, can be grouped into a *language*. It is important to note that such a language is just a structured collection of model unit kinds; it does not contain or define any notation. If one is needed, it has to be defined alongside the language. Since, the objects of such a collection have some relationship, model unit kinds are not defined independently but rather as a meta-model.

7.3. Example

Based on the definitions and conventions introduced in the previous section, let us look at a concrete example of meta-modeling. As before, cars can serve as an example use-case to show through a concrete example, what a SUS, a model and a meta-model would look like. Implementing a complete meta-model for the automobile industry is clearly outside the scope of this thesis. Instead, the example focuses on the important concepts presented in the previous chapters. According to Subsection 7.2.2, there are two approaches for creating models and meta-models: forward and backward looking approaches. The former

creates the meta-model before having any concrete models and creates models before knowing any concrete SUS. The latter takes the opposite approach.

Fig. 7.7.: Example Model Unit Kinds forming an automobile Language

As usual in the automobile industry, this example takes a forward-looking approach. This means, starting with a meta-model for the automobile industry, the designers first create a model of a new car and after some iterations and refinements, a concrete car is build. Usually, when designers start to build a new car, they don't reinvent the wheel each time. An automobile is always based on the same set of parts for instance, wheels, doors, steering wheel, lights, engine, seats and HVAC (Heating, Ventilation and Air Conditioning). Although there are differences between wheels, they all serve the same purpose, as the contact point between the automobile and the ground and can therefore be clearly identified as wheels. Compared to the definitions introduced in the previous chapter, each element of this set represents a *model unit kind*. This set would then be the *language* for the automobile industry. Figure 7.7 shows some elements of this set. It is important to note that all the elements relate to the automobile industry, so a language for this domain would include them. According to the nomenclature introduced previously, this set is the meta-model for automobiles. In addition to the individual model unit kinds, the meta-model also introduces some structure within these elements. There is for example, a relationship between the steering wheel and the wheels. Common sense also dictates that doors are used by passengers to board and get out of a car and therefore need to be accessible to the latter.

Between different models — and sometimes also between different brands — a considerable amount of parts remain the same (Skoda and VW for example share large parts of the cockpit). This limits the number of available parts for a given section of a car. In terms of the previous chapter, these parts are the *model units*, the atomic components of every car model. One such model unit would be the front shield of Tesla's Model X. All concrete Model Xs manufactured by Tesla will share the same front shield, thus they share the same *model unit*. However, a Bentley Continental has a different front shield. This leads to two different model units available for car models. Still, both models of front shield are of the same *model unit kind*. Starting with these model units, designers create a model for a new car. Although different models use the same model units, they

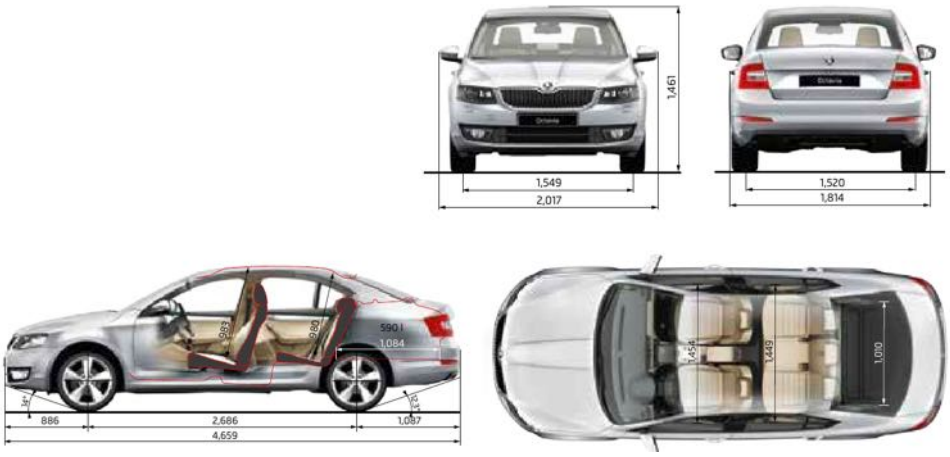


Fig. 7.8.: A model for a Skoda Octavia (from [WEB57])

do not necessarily look the same. Aspects like shape, size, placement of different model units etc. are only defined in a concrete model. Figure 7.8 shows part of such a model for a Skoda Octavia. One can identify the important parts of this car and their relationship to each other. From this point on, all concrete Skoda Octavia, like the one shown on Figure 7.9, will be instances of this model with slight variations due to different extras.

Fig. 7.9.: Concrete Skoda Octavia (from [WEB57])

7.4. Modelling with Eclipse

The first part of this chapter formally introduced the concept of meta-models and related terms leading to two visions of meta-models, one based on the four-layered model of the OMG and the other based on the concept of language and model unit kinds presented in [B9]. The second step was to show the application of these concepts to a real world example, the automobile industry. While the example gives a rough feeling about meta-models and the related concepts, it does not explain how meta-models are built and how they are finally used to create instances of concrete models. This section shows, with the help of a simple example, how to create a fully functional meta-model. The modeling tools bundled with recent Eclipse versions assist during the process of creating new meta-models but, also support instance creations of these meta-models. Indeed, it is possible

to add new meta-models to eclipse's modeling environment from which new instances can be derived. The advantage of these tools is the way they visualize the information. Actually, meta-models are represented like UML class diagrams. Since meta-models are a combination of terms forming a language plus some relations between the individual terms (see Subsection 7.2.3), visualizing the structure of a meta-model greatly helps to understand it.

Eclipse's modeling capabilities are supported by the core EMF including a meta-model called *Ecore*. Besides supporting the creation of models, it also provides runtime support for the created models. Regarding the OMG notation, Ecore is not only capable of generating models sitting on the $M2$-layer but can also create instances of the $M1$-layer. To support this task, Ecore provides a set of primitives. Among them, the EClass which represents one model unit kind. As with each UML class, the EClass can be further specified with EAttributes and EOperations. At the very end, this chapter develops a small example with EMF (Eclipse Modeling Framework) . To keep the example simple, a meta-model representing people and families is used instead of the car example, as its associated meta-model would not be suitable to show some important properties like *inclusion* and *extension*. Listing 7.1 shows the most important model unit kind when talking about families, a Person. Moreover, the listing shows that a person has a few attributes like a first and a last name or height. Defining the Person model unit kind is straight forward and the final code is very similar to Java code. The Person class is made up of a number of attributes, most of which are of a simple type. However, the eyeColor and gender attributes both have an enum type. Enum types have the same meaning as in Java, they implement a fixed list of choices for a given attribute, so gender can either be MALE or FEMALE. Each instance of Person has to choose one of these two values for the gender attribute.

```
1  class Person{
2     attr String firstname;
3     attr String lastname;
4     attr Integer size;
5     attr Integer weight;
6     attr Colors eyeColor;
7     attr Gender gender;
8  }

10 enum Gender{
11    MALE;
12    FEMALE;
13 }
14 enum Colors{
15    BROWN;
16    BLUE;
17    GREEN;
18 }
```

List. 7.1: Person model unit kind expressed in Emfatic

Although, a real person has many more attributes, those defined in Listing 7.1 are sufficient to illustrate the use-case. Yet, the concept of family encompasses some more model unit kinds at least — a family is made of parents and children. To keep the use case simple, a family with zero children is also allowed. This description directly translates to the Emfatic code of Listing 7.2. Since its first 19 lines are identical with Listing 7.1 and

to augment readability, they are stripped from Listing 7.2. Two new model unit kinds are introduced: `Parent` and `Child`. Although the Child model unit kind does not add any other attributes to Person it is still important to define it. Giving children an own class, allows them to be identified later and also permits the introduction of relations, where at one end, a Child is needed (`is-parent-of` relation for example). A Parent is also some kind of Person. Unlike the Child class, it adds another field yielding to its children. The multiplicity implies that a parent has zero or more children of type Child. In addition, Listing 7.2 shows two relations types available in Emfatic. The first, a containment reference is introduced with the children field in the Parent class. Through this definition, the Parent class contains a reference to the Child class. The second, inheritance, is used by both, the Parent and the Child class and shows how a model unit kind can be a specialization of another model unit kind.

```
20 class Child extends Person {

22 }

24 class Parent extends Person{
25    val Child[*] children;
26 }
```

List. 7.2: A simple family model expressed in Emfatic

The situation depicted on Listing 7.2 can be improved. Commonly when speaking about families, the role of a father, a mother and their children can be identified. All of them are still persons but the situation can be further refined. Additionally, mother and father are commonly seen as parents. Thus starting with Listing 7.2, the Parent class can serve as a basis for the Mother and the Father class. Furthermore, a child has exactly one mother and one father, reflected in lines 33 and 34 in Listing 7.3. The Child class defines one containment reference to its Father and another containment reference to its mother.

Since Father and Mother are the only two possible types of Parent, it is also a good idea to forbid the instantiation of other types of parents. This is achieved in the meta-model on line 20. By defining the Parent class abstract, instantiation is forbidden, yet, it still serves to define common properties for Parents, for example, the case of the `children` attribute in line 21. Accordingly, the Father and the Mother classes are both inherited from the Parent class. Here it would be nice to already fix the `gender` feature to `MALE` for the Father class and `FEMALE` for the Mother class. However, Emfatic and Ecore do not support overriding features in sub-classes. Again Listing 7.3 is not complete. Its first part is exactly the same as discussed in Listing 7.1 and has therefore been truncated here.

```
20 abstract class Parent extends Person{
21    val Child[*] children;
22 }

24 class Father extends Parent{

26 }

28 class Mother extends Parent{

30 }

32 class Child extends Person {
```

```
33    val Father[1] father;
34    val Mother[1] mother;
35 }
```

List. 7.3: A more complete family model expressed in Emfatic

This model could be further refined by defining the Person class as interface or by defining other model unit kinds common in families (like uncle or aunt). Once the meta-model is fully described in Emfatic, it can be transformed into Ecore and also an Ecore diagram. The result of this process is shown in Figure 7.10. Once the meta-model is transformed into Ecore, it can be installed into Eclipse making it available as a prototype for new models. Therefore, it would be possible to create the instance *Fam. Hofstadter* with Father *Leonard Hofstadter*, Mother *Penny Hofstadter* and two Children *Howard* and *Sheldon*.

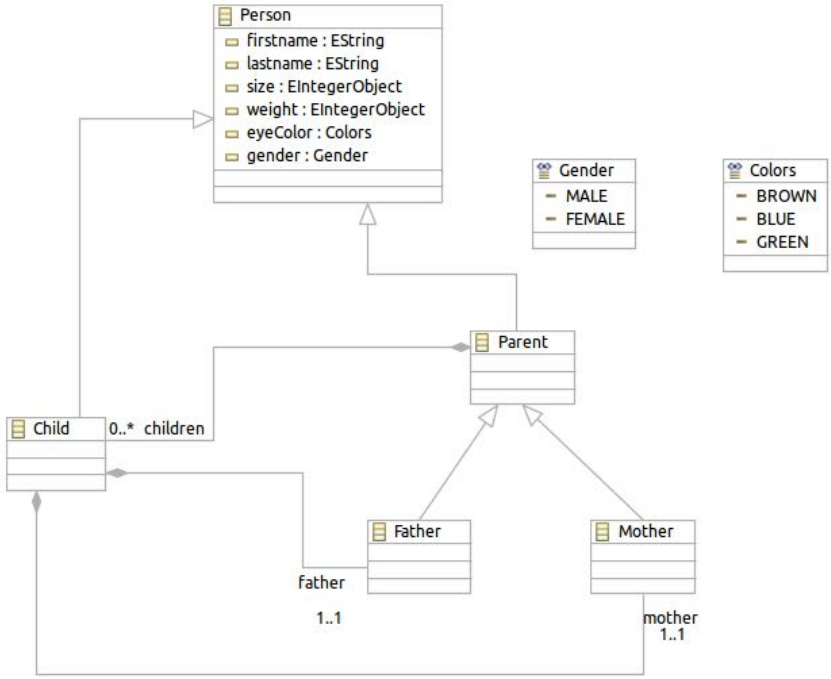


Fig. 7.10.: Meta-Model of a family in EMF

7.5. Key Concepts introduced in this Chapter

The conclusion of Chapter 6 showed that a component based architecture can only work, if all the components are well structured. Additionally, all developers should agree on some naming convention, or at least, the terminology chosen should be very clear. Furthermore, as noted, a meta-model is the suitable approach to define the inherent structure of the WoT components. To come up with a suitable meta-model this chapter introduced the notion of meta-models and the foundations of meta-modeling. In the literature, there are two major approaches to meta-modeling. The first is based on the Semiotic triangle (see 7.1) and defines the relationship between System under Study and models (respectively between models and meta-models). The second approach is based on work of Gonzalez-Perez [B9] where a *language* (a set of model unit kinds) is used to describe

models. Both approaches become at some point philosophical. The semiotic triangle underlines the difference between a cognitive model and a model, although in computer science, they are, most of the time considered as the same. The important relation for this thesis is given in Equation 7.8 describing the relationship between a SUS and the meta-model it is derived from. Since the relation between a SUS and its model is a homomorphism and the relation between a model and its meta-model is also an homomorphism, there is a relationship between a SUS and its ancestor meta-model.

The Semiotic triangle introduces the relation between a SUS and the associated model. Yet, it does not specify whether the model existed first (forward looking) or whether it is derived from the SUS (backward looking). When smart devices are created, the usual approach is to take something that already exists in the physical world, like a light bulb, and make it smart by adding a virtual representation and a communication API (REST in this case). Therefore, a developer starts with the physical light bulb and describes its properties. In the second step the physical object is divided into resources with their hierarchical dependencies. Therefore, the process of creating new models of smart devices will often be backward looking (also called analysis). This seems obvious as the main task is to re-create in the virtual world what already exists in the physical world. On the other hand, these model comply to the xWoT meta-model. Clearly, the meta-model existed before all these models and serves as the ancestor of them. Therefore, the relationship between xWoT models and the xWoT meta-model is forward looking (also called specification). This remains true even though the meta-model itself was derived from SUSs at some point. Thus, the task of creating new smart devices is at the same time an analysis (of the physical world) but also a specification (of the RESTful interface).

This discussion appears to be turning philosophical. For this thesis, it is enough to state that models are inspired by reality and they comply to the xWoT meta-model. Gonzalez-Perez prevented this situation by defining the meta-model as the language constructed out of model unit kinds. Models can then be build by picking and combining items out of this set. Additionally, Gonzalez-Perez introduced another interesting concept with this approach, both of which have importance for the xWoT meta-model. The creation of such a language means its model unit kinds have to be named, introducing concrete terminology which can then be used in models to express different SUS. Additionally, the language contains information about the relationship between the different model unit kinds. This is also important for the xWoT meta-model. The example presented in Section 7.4 shows that using the modeling tools bundled with Eclipse can express both *terminology* and *structure* in an EMF meta-model.

Part III.

The xWoT Environment

8

Towards a component-based xWoT

Section 4.3 introduced the Web of Things and traced its history from its beginnings to the current state of the art. The WoT is now at a stage where it is mature. The technologies and architectures involved have proven to be successful in many applications and consumer products. As for RESTful architectures, they are by far the preferred way for creating publicly available APIs nowadays (see Figure 3.4). Of course, this success is only partly due to the WoT. Another success factor is due to mobile phones and mobile applications where REST-like services provide an easy way to connect the front-end application running on a mobile device with the back-end data available on some servers. On the other hand, consumer products like the Koubachi plant care platform [WEB29] (1000 - 5000 downloads of the Android application accompanying the sensor) or the various fitness trackers (fitbit [WEB18], Nike+ [WEB36], Jawbone [WEB27] etc.) have successfully laid the foundation for the IoT. Through platforms like Arduino [WEB5], SunSpot [WEB58], openPicus [WEB38] and many more, developers have all the necessary tools to build numerous WoT applications.

This chapter will introduce our vision of an extended version of the Web of Things, the xWoT, embracing not only smart devices but also virtual only services. Furthermore,

the xWoT can iron out some shortcomings of the actual WoT and serves as a basis for the meta-model introduced in the next chapter. This chapter is divided into 4 sections. The first one identifies the limitations of the current WoT. The second section addresses two of these limitations by classifying the different types of client-server interactions and discussing various approaches to push information from servers back to clients. The third section introduces the extended WoT (xWoT) and defines its building blocks. Finally, the fourth section applies the introduced concepts to a concrete use-case and shows the applicability of the xWoT, as foreseen by the author.

8.1. Weaknesses of the actual WoT solutions

At a very early stage, D. Guinard et al. promoted the usage of a resource-oriented approach to build upon already established technologies [24]. The idea of creating applications out of what was already available on the Web (in this case the Things) evolved and finally became a central part of the work of D. Guinard [21, 23, 24, 25, 19]. The fast growing WoT community rapidly picked up this idea and multiple applications and even some commercial products emerged [WEB29]. The core idea is to promote smart objects or Things to first class citizens on the Web, making them directly usable and browsable like any other resource on the Web, which has made the WoT a success story. Although there were other, similar ideas for the IoT [42] the WoT also offered a way to achieve this goal giving it a huge advantage over the IoT. Today, the WoT has proven its viability but still has some important shortcomings, of which three are:

1. *Data integration.* Since the WoT is a web of Things, they are the main source of information. Past experience has shown that interesting applications however combine data from the WoT with data from purely virtual services like computational algorithms, business processes and other services. Services with no relationship to any physical object are outside the scope of the WoT. However, taking into consideration service components is an essential requirement for the long-term success of the WoT.
2. *Events.* They play an important role in any piece of software. For GUI applications these events are the mouse and key listeners used by the application to react to user input. Similarly, the WoT needs to integrate events to build interesting applications. Sensors can trigger actions only if events can be pushed to clients. The WoT lacks a common pushing infrastructure making it difficult to build applications relying on events.
3. *Building Blocks.* Mashup applications are often the link between users and the WoT and many efforts have been spent on either creating mashup editors or dedicated mashup applications. Although smart devices are the building blocks of these mashup applications, they are mostly ignored and treated like a black box. Mashup creators simply presume the existence of the necessary smart devices. Today, there are no guidelines on how to create these building blocks

8.1.1. Data Integration

Another point mostly ignored is how the WoT deals with data. By definition, the WoT is the Web where Things are first class citizens, a Web, where the participants are Things.

Although such a simple definition might look easy to understand at first glance, problems already start from the definition itself. Whereas, it seems clear that a door can be turned into a smart door by adding an actuator capable of opening and closing the door combined with a sensor monitoring its state, it is less obvious how a book can be turned into such a smart device. Intuitively, the book exists on the web as text and an Radio Frequency Identification (RFID) tag closes the gap between the physical object and its virtual representation. This shows that the notion of smart device is fuzzy. The physical book does not really become smart through an attached device. Flipping pages in the virtual book does not flip pages in the real book. Therefore, the coupling between the physical and the virtual world is not always as tight as one might think at first. The WoT does not define how much coupling is needed to take part in the WoT. Besides, mashups not only consider information provided by smart devices, but often also integrate information from traditional web sources. Twitter, Facebook and Google Maps are certainly the most popular in all sorts of mashup applications. Under the same conditions as for the book example above, it is arguable that Twitter and Facebook are virtual representations of a person and hence fully embedded into the WoT. Furthermore, Google Maps can thus be interpreted as one possible representation of a resource. However, such arguments cannot be found for every service of interest for the WoT.

The status of purely virtual services or algorithms in the WoT is an open question. In [50] Mayer et al. suggest the need for purely virtual goods because in a near future, smart devices will create such a huge amount of data that some strong filtering and aggregation mechanisms will be needed before data can be made available to the final user. In their paper, the authors introduce a market place for computational algorithms respecting the HATEOAS (Hypermedia as the Engine of Application State) principle. Upon requesting an algorithm, the response contains a certain number of forward paths which the client can choose from to continue. Furthermore, the authors provide a proof-of-concept implementation of such a marketplace. They address issues like discovery, security and billing. In [2] Alarcon et al. the authors introduce the *Resource Linking Language* based on Petri Net models using the HATEOAS principles to link resources. Such papers show that the REST community is actively doing research on common topics related to the service world: discovery, security, billing, late-binding etc. Most of this research is targeted at computational services however, since the WoT uses RESTful interfaces, it should also take into consideration service related aspects and benefit from the research already done in this domain. As promoted by Mayer in [50], computational resources can drastically reduce the amount of data produced by smart devices. Furthermore, such smart devices have, in general, limited computational capabilities and tight constraints on battery consumption. Therefore, most are only embedded with the bare minimum of business logic. Yet, linking the output of smart devices to computational resources can greatly improve their raw capabilities. Additionally, such an approach adheres to common design patterns like the re-usability of code. In fact, an aggregation algorithm can be deployed on a machine without any CPU or battery constraints. Therefore, many smart devices can use one service. Software engineers know that such re-usability has many advantages: bugs need only to be fixed once, new features become instantly available to all clients, developments costs are greatly reduced. Taking these observations and arguments as a basis, we identify four categories of purely virtual services. In [71] we discuss for each category how they can be seamlessly integrated in a RESTful architecture and thus integrated into the WoT.

Business Processes form one of the four categories we identified in [71]. In the world of SOAP services and enterprise environments, business processes have become well established. Today they are the de-facto standard when it comes to service orchestration and composition. However, in their current state they are outside the scope of RESTful architectures and the WoT. Some work has been done to integrate IoT specific tasks within the BPM specification [53, 48, 52]. S. Meyer et al. analyze in their work how business processes can take advantage of smart devices and they propose some IoT specific extensions to the BPM meta-model. Meyer et al.'s work seems to solve this problem, at least for the IoT. Since the WoT can be seen as a subset of the IoT, Meyer's work can easily be adapted to the WoT, so business processes can have access to smart devices. With the growth in complexity of WoT scenarios, orchestration and service composition also become important in the RESTful world. C. Pautasso et al. conducted some work on BPM extensions and adaptions for RESTful web services [61, 62] and propsed in [61] an extension to BPMN (Business Process Model and Notation) to deal with the specificities of REST architectures. Through this extension it is possible to model RESTful web services with BPMN. In [62] Pautasso propose an extension to BPEL (Business Process Execution Language), which allows orchestrated processes based on RESTful web services to be executed. S. Kumaran describes a business process execution engine which is completely RESTful [43]. Instead of focusing on message flows as in traditional BPM systems, they propose a more data-centric approach. Each piece of data represented is business critical. Since, by definition, the base architecture is RESTful, such execution engines are not only able to consume RESTful web services but REST client can also consume them. However, it seems that this approach implies a fully RESTful architecture for the whole infrastructure, thus excluding traditional business processes.

8.1.2. Events

When it comes to smart devices, pushing information is always an important aspect. Smart devices run on limited hardware and often have very strict requirement regarding their power consumption, therefore, they cannot afford a lot of clients constantly polling the device for new information and thus, preventing it from switching into some low power consumption mode. On the other hand, some scenarios have a strict requirement to rely on events. For example, consider the following use-case: "As soon as somebody enters the room, switch the light on if it is not already on." It is evident that this type of scenario requires at least some sort of presence sensor to detect humans and a light switch. The logic combining these two smart devices would then be coded into some mashup application. Platforms like Paraimpu, Xively and others [WEB43, WEB75, WEB64] have arisen from the necessity to push information to mashup applications. Imagine the Japan Geigermap application [WEB25], if the application had to pull the information out of each of the connected devices on a regular basis (once per hour or a few times per second), not only would the performance be outstandingly bad and the network uselessly charged with data but the Geiger counter smart devices would be unnecessary charged due to the frequent update requests. These platforms seem to overcome such limitations by storing the information provided by smart devices. Therefore, mashup applications do not rely directly on the underlying smart devices any longer. Instead, they rely on middlewares. As a consequence, the smart devices no longer offer any RESTful API. Rather, they implement a client to the REST API offered by these middlewares. The problem with

this approach is that the smart device is highly coupled to the middleware platform. If this platform should disappear, what would happen to all the smart devices relying on it? The recent history shows that such a scenario is more than some gedankenexperiment: cosm replaced Pachube, which in turn, was replaced by Xively [WEB75].

8.1.3. Building Blocks

The Web of Things as suggested by Dominique Guinard [19] and many others [7, 18, 11, 27] foresees a Web where Things are first class citizens which clients can directly interact with and and manipulate. Therefore, the Web of Things can be seen as some type of Web, similar to the one we use everyday but, instead of browsing web sites, clients browse smart devices. These smart devices are the building blocks of the WoT. By their nature, it is possible to combine the information provided by smart devices with information from any other source, as long as this source provides a RESTful interface. Many papers promote the requirement for RESTful web services as architecture for the WoT arguing for their interoperability. Having all Things speak a common language makes it easy to combine them and build new creative applications on top of them. Commonly, these applications are called *mashups*, as the one presented in [5]. Although mashup applications are not a new concept - they existed well before the WoT - it heavily makes use of this approach. Early services, as presented in Chapter 3, have already used some kind of information blending as one of their objectives. If mashups were originally limited mostly to what was available on the local machine, mixing up information rapidly gained momentum with the coming of WS-* services. In [WEB31] S. Watt not only arguments in favor of mashups in a SOA environment but he also advocates what he calls "situational applications", applications which are developed for a very small target audience. While this would be almost impossible with traditional software engineering, using web services and mashups makes it really easy to adapt a service, delivering some basic functionality to different situations at almost no cost. In this approach, the next logical step would be to let the final user construct his applications himself. Building mashup applications should become simple, so that the final user, who also knows best the business processes, can create and adapt applications to his needs. The same argument supports the WoT: interacting with the virtual side of a smart device should be as simple as touching and manipulating the Thing in the real world. Thus, much effort has been invested in inventing and building platforms allowing non-programmers to create their own small applications using their smart devices.

Smart devices with a RESTful interface are the building blocks of the WoT, and by combining several of them, mashup applications emerge. Creating sensor based mashup applications has become popular. It does not matter if a mashup mixes up information from different sources like the EPC Dashboard [57] or if it rather aggregates multiple sensors measuring the same phenomenon like the Live Flight Tracker of Figure 8.1. They all have in common that they treat the smart object like a black box, offering a RESTful interface. Since mashup designers deal with creating new applications with already available components, they don't spend much time on creating such components or designing them. A lot of work is either spent on how to connect a smart device to a network or on how to make mashup editors user friendly [76, 44, 33, WEB43, WEB75, 66]. Since the WoT imposes RESTful architectures, the how is not really the question. In order to participate, smart devices need to offer a RESTful API over which a client can interact with it.

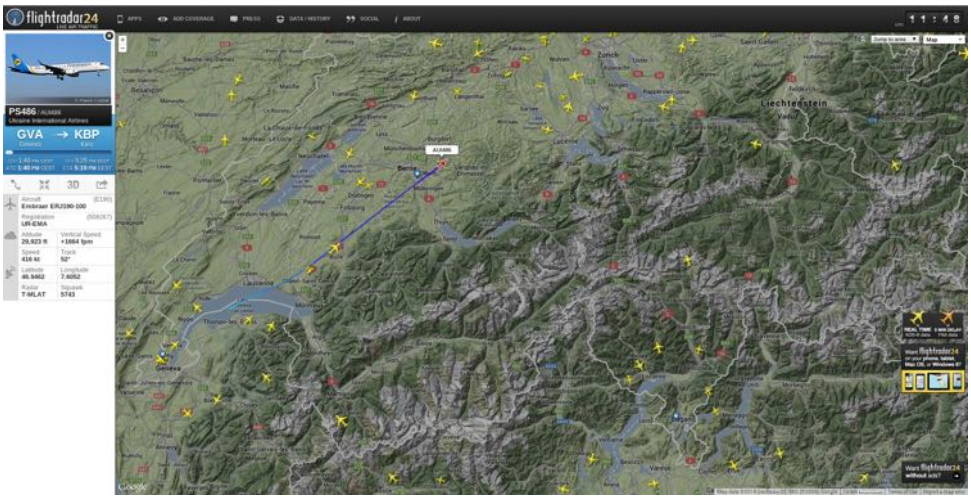


Fig. 8.1.: Live Flight Tracker from `flightradar24.com`

Although Richardson and Ruby [B18] thoroughly discuss different situations and how to model them with a RESTful web service this aspect seems greatly neglected. Usually, this is taken for granted and only a few thoughts are spent on what such a smart device looks like and what services are offered. On the other hand, judging by the amount of different consumer-ready mashup creators and platforms, it seems that they have been well studied. Xively (formerly known as cosm which was formerly known as Pachube) [WEB75], Paraimpu [WEB43, 66], ThingSpeak [WEB64] and Open.Sen.se [WEB39] are some of the most popular platforms dealing with smart devices. Offered features vary, depending on the platform but their core functionality is similar. However, WoT mashups are not restricted to specialized platforms indeed, by their nature, any mashup editor for the web can take advantage of the information offered by smart devices living in the WoT. As such, they all have in common that they treat smart devices like a black box. In order to use a smart device in a mashup, it is necessary to first read through the specification of the latter or to browse and discover the available resources and what they mean. Furthermore, given two smart devices measuring or acting on the same physical phenomenon but issued by different vendors, chances are high that the two REST APIs are completely different. This makes the integration of different smart devices more difficult.

8.2. Services and Pushing in the WoT

Having discussed some limitations of the WoT, a closer look at the problem of data provenience and information pushing is required. These two aspects play a central role for the xWoT. Besides defining the components on which other applications can build in Section 8.3, the xWoT embraces services tied to physical objects but also algorithms and other computational resources. To come up with standard components, the xWoT also needs a way to push information between the different participants. In the first part, this section classifies the different types of services of interest for the xWoT and for each class, discusses how they can be embedded in the xWoT. The second part of this sections deals with the different solutions for pushing events generated by smart devices back to clients.

8.2.1. Classification of Services

Historically, the WoT started by augmenting physical devices with a virtual side in a standardized manner. Choosing REST as an architectural style allows for the easy combination of different smart devices. Although services like Facebook, Twitter, Flickr are not related to a physical device, they become standard components of modern mashup applications. Mayer et al. [50] recommend the introduction of a market place containing computational algorithms, which can then be linked to. Integrating such services is straightforward. Different kinds of services ask for different types of interactions. Invoking Twitter to post a new Tweet is just a matter of one request. Other services might need more sophisticated interactions to complete. Sometimes, services can take a considerable amount of time to complete, so they can decompose, or can only exist in a form outside the scope of the WoT. Pautasso [61, 62], Kumaran [43] and others have investigated how RESTful web services can be integrated into business processes and how business processes can live in a RESTful environment. However, there is quite a gap between services like Twitter and full business processes. Moreover, the WoT already uses many different services today: Twitter, Facebook and Google Maps are some examples. To be of any use for the WoT, a service needs a RESTful architecture (or at least a RESTful façade). The integration of these types of services opens up new challenges which need to be addressed.

From this discussion, it already seems that at least three categories of service can be identified: (1) short services, (2) time consuming services and (3) business processes. Additionally, two more types can be defined: (4) services with a real-time constraint and (5) services which decompose. This brings the total of categories to five. In the remainder of this chapter, each of the five categories is discussed.

Short Living Services

This first category encompasses the most common services used in the WoT and mashup applications. Services living in this category are called *short living*. This means that they immediately return with the desired result. It does not matter what type of computation is carried out behind the request. The Directions API of Google is an example of such a service. Routings are available over a RESTful API behind the resource `directions`. Listing 8.1 shows the necessary query with the command line utility `curl` to fetch the direction from the address *Route des Fougères 1, 1700 Fribourg, Switzerland* to the destination *Landstrasse 9, 18374 Zingst, Deutschland*.

```
1 ruppena@tungdil:~$ curl -X GET -H "accept: application/xml" "http://maps.googleapis.
    ↪ com/maps/api/directions/xml?origin=46.794065,7.159096&destination
    ↪ =54.418504,12.751842"
```

List. 8.1: Requesting the Google Direction API

The request immediately returns with the response shown in Listing 8.2.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <DirectionsResponse>
3  <status>OK</status>
4  <route>
5   <summary>A5 and A7</summary>
6   <leg>
```

```
   ...
   <duration>
    <value>39717</value>
    <text>11 hours 2 mins</text>
   </duration>
   <distance>
    <value>1187918</value>
    <text>1,188 km</text>
   </distance>
   <start_location>
    <lat>46.7941031</lat>
    <lng>7.1591038</lng>
   </start_location>
   <end_location>
    <lat>54.4184775</lat>
    <lng>12.7518179</lng>
   </end_location>
   <start_address>Route des Fougeres 1, 1700 Fribourg, Switzerland</start_address>
   <end_address>Landstrasse 9, Western Pomerania Lagoon Area National Park, 18374
       ↪ Zingst, Germany</end_address>
  </leg>
  <copyrights>Map data 2014 GeoBasis-DE/BKG (2009), Google</copyrights>
  ...
 </route>
</DirectionsResponse>
```

List. 8.2: Truncated response of the Google Directions API

The service is requested with a `GET` request and all the information necessary to complete the request is transmitted in the URL. From a user's perspective it is impossible to tell whether this resource serves static or dynamic content and it doesn't matter. Services in this category mostly behave like smart devices: a user requests information from the resource and immediately obtains a response containing the desired data. For this reason, short living services integrate seamlessly into the WoT and consuming such services is mostly just a matter of one request. Furthermore, from a service perspective, services offered by smart devices generally fall into this category. Listing 8.3 shows how a user gets information from a smart device. Here, the smart device is a simple smart thermometer returning the current measured temperature plus some meta-information

```
ruppena@tungdil:~$ curl -X GET -H "accept: application/xml" http://10.1.1.1:9000/temp
```

List. 8.3: Request to a smart thermometer

There is almost no difference between the request to Google Direction API in Listing 8.1 and the request to the smart thermometer shown in Listing 8.3. Listing 8.4 shows that although, the XML response from the Google Direction API and that from the smart thermometer differ in their structure, they both represent the requested resource and in that are similar too.

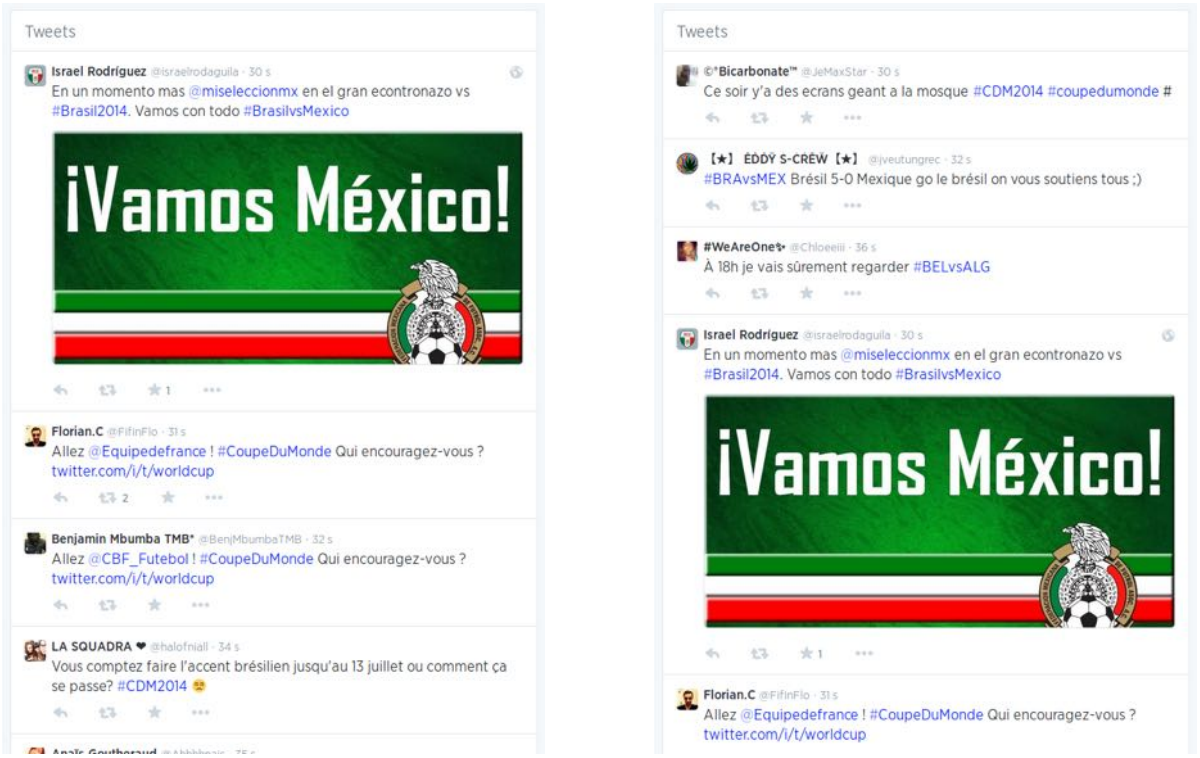
```
<?xml version="1.0"?>
<measure>
   <temperature units="celsisus" precision="2">28.36</temperature>
   <timestamp> 1402927649</timestamp>
</measure>
```

List. 8.4: Response from smart thermometer

Listings 8.1, 8.2, 8.3 and 8.4 show that a user cannot tell the difference between an invoked service returning some values from a smart devices or an answer which is the result of a computation. Furthermore, the WoT can already deal with services of this type without any further modification. Therefore, short living service and WoT services should be treated as the same.

Real-Time Services

The second category is quite similar to the first. Services which belong in this category still respond quickly with a representation of the resource to a request. Again, from a client's perspective, it is impossible to tell whether the response comes from a smart device or is the result of a computation. However, the returned representation may change over time. For example, it is unlikely that consecutive executions of the request to the Google Directions API of Listing 8.1 will result in different responses, but this is not true for all representations. Going back to the smart thermometer, it is very likely that two consecutive requests may result in two different temperature readings. In fact, chances are high that, upon each request the returned temperature changes. The difference between the two lie first, in the frequency such changes occur and second, whether the user is interested in following these changes. Indeed, a user might be interested in following the changes in temperature over a given amount of time. Therefore, he not only needs to know the temperature at certain points in time but he also needs to know when a change occurs.



(a) Result of first request (b) Result of second request

Fig. 8.2.: Two consecutive requests to Twitter Trends for #WorldCup2014 separated by a few seconds

Such services are referred to as *real-time services*. In computer science, the word "real-time" commonly implies that a client is informed about an event within a given time frame. This is also true for services within this category. As changes happen, the new information needs to be pushed to the client. While it may seem obvious that smart devices fall (at least partly) within this category, it also includes purely virtual services. Twitter is a good example of a real-time service. Although, Twitter offers a RESTful API, for the sake of simplicity the discussion is based on the HTML representation of the different resources. Figure 8.2 shows two screenshots of the `#WorldCup2014` topic separated by a few seconds. Between the two requests, the resource has already changed. This change is reflected in Figure 8.2b, which shows three new tweets compared to Figure 8.2a. Again, a user might be interested to know when a new tweet is posted.

Of course, real-time services can be consumed in a similar way to short living services. A user can `GET` the representation of a smart device or a service at any specific moment in time. But generally, a user is more interested in getting either continuous updates or notifications when values change rather than requesting the resource at specific moments. How a piece of information can be pushed from a server to a client is the topic of Subsection 8.2.2.

Delayed Services

The third category is *delayed services*. Unlike for the preceding categories, when a user requests a delayed service, the server is unable to immediately reply with a representation of the latter or within a reasonable amount of time. Reasonable is defined as the amount of time before a connection is dropped. Usually HTTP connections have a timeout, after which the client (or the server) closes the connection even though no response has been sent back. If the computation on the server is long, such dropped connections may happen. The planning of round-robin tournaments with complicated constraints like minimizing breaks as described by Briskorn [B4], is an example of time-consuming operation. A service offering this type of computation could not immediately respond to a client's request.

The combination of such computations with RESTful web services is rather delicate. Whereas a dropped connection has not much influence on the behavior of the client (given that it handles such situations gracefully), the consequences on the server side are tremendous: through the initial request, the client has initiated a computation on the server side. If the connection drops before the server can send back an answer, the computation will still continue and eventually deliver its result to nowhere. However, the client will try again and by doing that, launch the same computation a second, third and fourth time. Clearly, this is a huge waste of resources. To overcome this limitation, such scenarios could define longer HTTP timeouts. Yet, this is not a viable solution either. During the waiting period, the client gets no feedback to inform him whether the server has accepted the request, whether it is working or not or how it is progressing. Even worse, a client cannot tell if there is a connection problem between him and the server or if the process is still running on the server. Finally, augmenting the connection time-out would lead to the same problems as already mentioned: the impatient client will repeat the same request several times and flood the server with new computations which will never deliver their results to the client. The fact that systems with higher HTTP connection timeouts scale badly is another arguments against this approach. This comes

from the fact, that for each open request, the server has to reserve some memory space. Thus, it is in the interests of the server to keep the number of open connections as low as possible and so leave enough free memory for new clients.

Richardson et al. [B18] propose a simple but very efficient way to handle this situation. Instead of trying to keep the connection alive and deliver a result during this session, the server immediately responds with a message indicating that it is taking care of the computation and gives a URI where the client can check later for the desired result. In RESTful words, this means that instead of issuing a `GET` request, the client constructs a `POST` request. In response to this request, the server creates a new resource and sends back the status code `202 Accepted`, together with the URI of the newly created resource. From this point on, the client can check the new resource for the result with a regular `GET` request. Not only does the server frees up some memory, but the client can also work with the new resource like with any other. He can share, link and bookmark it. Another benefit of this solution is the long-term availability of a result. Considering that each computation, and thus each result, have their own resource, a result can be looked up several times. Since it would have been hard to compute this result, caching it for some time surely makes sense.

Decomposable-Delayed Services

Decomposable and delayed services are an extension of delayed services. From a client's perspective, such services behave like delayed services: upon requesting the service, it is unable to respond with the result. Instead, it responds with a `202 Accepted` plus some URI where the client can check the status of the computation later. Decomposable-delayed services thus suffer from the same shortcomings as delayed services. However, the same solution can be applied to make them truly RESTful. The difference between the two categories lies in the type of resources created. Whereas for simple delayed services, the server creates one resource per computation request, decomposable-delayed services create a whole hierarchy of resources. In order to solve the problem submitted, the server splits the initial problem into several smaller ones, each of the smaller problems eventually produces a result worth of being addressable as a resource. This comes from the fact that the solution to the initial problem is the concatenation of the solution of each sub-problem. This category is not to be confused with problems, which initially decompose into smaller problems and finally assemble again to compute the global solution like map-reduce jobs. Such problems produce only one solution and it does not make sense to save the intermediate results for later. Therefore, they fall into the category of delayed services. The VRP-TW (Vehicle Routing Problem with Time-Windows) shows a problem where the initial task decomposes into several sub-problems and each sub-problem produces an independent part of the overall solution. It consists of finding an optimal set of routes to be traveled by a fleet of vehicles [B8]. Solutions for this type of problem are computed in two steps:

1. Partition the point 2 to n to visit into k subgroups and
2. For each subgroup find a shortest tour.

Clearly, the solutions would consist in combining the partitioning of the space plus the optimal route for each partition. Accordingly, if the fleet has k vehicles, the problems produces $k+1$ solutions. Additionally, accessing each sub-result presents an added value. Let's imagine a modern parcel delivery station where during office-hours parcels can check-

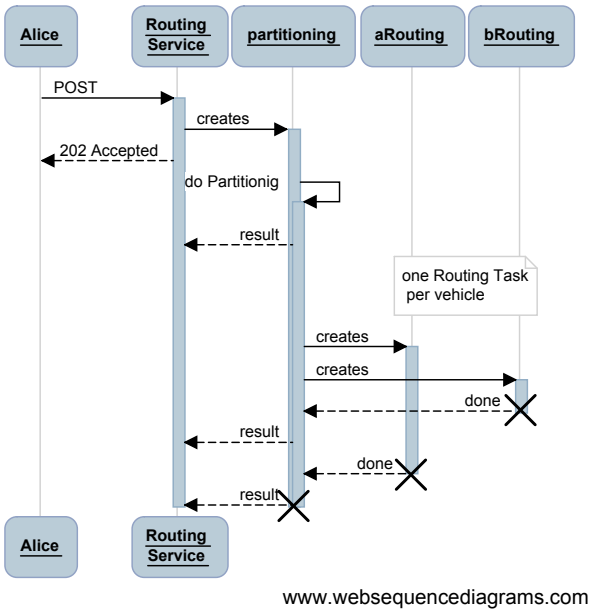


Fig. 8.3.: Sequence for solving one instance of VRP-TW with two groups

in. Upon arrival at the parcel delivery station, each one is scanned and its destination address fed into some system. Over night, the company's mashup application takes this set of destination addresses and asks the VRP-TW solver service to compute an optimal solution. Since each computed route is accessible over its own URI, each driver can check their delivery round in the morning and start delivering the parcels. Furthermore, this resource can be part of another mashup application running on each driver's smart phone. This application tracks the progress of deliveries and guides the driver to the next destination. Upon successful delivery, the driver marks the parcel as delivered in his mashup application. This updates the corresponding parcel resource at the parcel delivery station. Undelivered parcels are returned to the parcel delivery station and re-scheduled for the following day. This happens automatically, since the VRP-TW mashup first gets the destination addresses of all the parcels to be delivered before feeding them into the VRP-TW solver service.

Although VRP-TWs are complicated problems, this complexity remains hidden from the user. Figure 8.3 show a sequence diagram of a `POST` request to the VRP-TW solver service. This initial request contains all the input data for a new instance of a problem. The figure shows that the client only interacts with the *Routing Service* façade, the rest is handled by the service. Upon receiving this request, the service creates a new resource and sends back the URI with a `202 Accepted` status code. At the same time, the partitioning task is launched. As soon as it terminates, the server creates k new sub-resources, each one representing one sub-task, and launches the k routing algorithms. The client can check at any time the overall progress of the request by consulting the returned resource. As soon as the server starts the k routing tasks, this resource will contain links to the newly created subresources. From this point on, the client can also track the individual progress of one sub-task.

The concept of *task* is tightly coupled with delayed and decomposable-delayed services. When a user submits a new problem to solve, the server creates a new task. This task, represents the user's problem and is exposed through the returned resource. Before an-

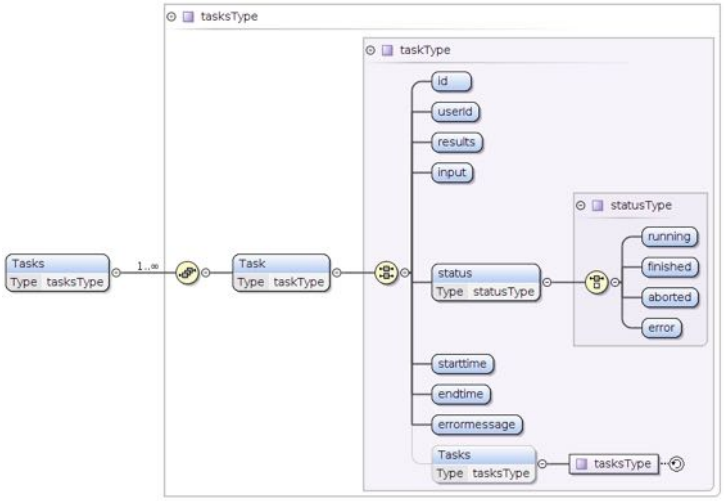


Fig. 8.4.: Common set of attributes of Tasks and Task lists

alyzing the inner guts of a task, exposed resources require examination. The top-most resource of such a service is generally the *list of available tasks* on this service. This list may show all the tasks, a sub-set of the available tasks or none, depending on the defined access policy. Each element of this list represents a *task*. Furthermore, since a task can decompose into several sub-tasks, it contains in turn a *list of sub-tasks*. Since sub-tasks play an important role in decomposable-delayed services, each element of this list can be addressed individually. This description infers a relational hierarchy, which should also be reflected in the URI associated with each resource. Associating semantics to URIs is a highly controversial subject in the WoT community. Opponents claim that semantics introduce a coupling between the RESTful service and clients [WEB6]. Furthermore, they advocate that related resources should be linked to in the body of the representation (as stated by the HATEOAS principle). We don't think that semantically meaningful URIs increase the coupling between a service and its clients, but rather, that semantically meaningful URIs benefit the WoT. In the past few years, the web has shifted from random URIs to meaningful ones (also called SEO friendly). Pretty URIs and techniques like URL rewriting transform `http://.../3408ycv0acvoj/` into `http://.../blog/20140713-google-glass` have become a standard on the web, so why shouldn't the same be true for the WoT? Semantically meaningful URIs do not violate nor replace the HATEOAS principle, but they ease the identification of what the resource is about. It is easy to guess that `http://.../temperature` is about a temperature whereas `http://.../adf43af` could be anything. Furthermore, it is not forbidden to *guess* URIs but a client needs to be aware of the associated risks. Finally, in RESTful architectures HATEOAS is always the way to go to follow a given execution path. Accordingly, the resources offered by a delayed or a decomposable-delayed service respect the following URI pattern: `http://.../tasks/<id>/<sub-id>/` where the `tasks` part points to the list of tasks on this service, `<id>` is the placeholder for one given tasks and `<sub-id>` is the placeholder identifying one sub-task of the task `<id>`.

From their structure, a task and a sub-task are very similar. Both represent a running process, which eventually produces a result. Assuming that sub-tasks don't divide recursively into sub-sub-tasks, the main difference is that a task has child resources whereas a sub-task doesn't. From a structural point of view, this can be interpreted as a task

having a list of children whereas a sub-task has an empty list with no children. Under this assumption, it appears that structurally, tasks and sub-tasks are the same. Accordingly, the starting assumption that sub-tasks don't further divide can be removed, allowing sub-tasks to have sub-sub-tasks and so on. In [70, 71] we show how a task (and consequently also a sub-task) can be structured. Although there might be minor differences from one service to another, all concur on the proposed structure as a starting point. A task is always defined by at least the following set of attributes:

- The `id`, a unique identifier for each task. This is also reflected in the URI associated with each task.
- The `userid` associates each task with a user. This information has various usages like billing or access policies.
- The `result` is the main interest of the user. Initially the task was created to later retrieve the information held by this field.
- The `input` contains a copy of all the inputs which led to the task creation. This can come in handy to distinguish tasks later or to recall the different input parameters.
- A `status` field reflects the actual status of the task. This field mainly shows whether the service has terminated the task or not. Additionally, a user can abort a task or a task can produce an error and quit.
- In case of errors, the `errormessage` field contains some hints about what went wrong. In case of successful termination as well, this field can contain some valuable information like the number of used CPU cycles (which in turn can be used for billing purposes).
- The duration of a task can be computed from the `starttime` and `endtime`. Returning both fields is more valuable than just the raw duration of the task. Again, these fields are mainly used for billing purposes and statistics.
- Finally, the `Tasks` field contains a list of `Task`, each one being a sub-task of the original problem.

Figure 8.4 summarizes this situation and shows that most attributes do not explicitly specify their type as this typically depends on each concrete situation. One service may prefer to identify its users through a string whereas another assigns a random number to each client. Yet, it is important that this attribute is available. The `status` attribute is somewhat special and has an associated `statusType`. It is quite obvious that a task always adopts at least one of the four defined statuses. Additionally, at any given moment, a task has exactly one status associated with it. Again, some services may need additional status types, like a `waiting` status to indicate that a job, although accepted is still waiting to be processed. Whether a service exposes all the attributes or just a subset is out of the scope of a general discussion. For a service limiting access to tasks owned by the requesting user, the `userid` field could be stripped from the representation as the user knows who the tasks belongs to. Furthermore, the semantic of a concrete service also decides which of these arguments are read-only and which are read-write. This reasoning recursively applies to all available resources. Only the concrete use-case decides which set of actions is possible on which resource.

Business Processes

This chapter began with simple atomic services. Gradually, we added some complexity and ended up with the decomposable-delayed services. Yet, the world of services is much bigger than what has been discussed so far. This comes from the fact that a huge amount of web services are deployed in enterprise environments and go far beyond a simple request - answer interaction. Modern information systems are often based on the composition or orchestration of a multitude of smaller services, each one responsible for carrying out one aspect of the initial problem. In that, they are similar to decomposable-delayed services, not only is the initial problem decomposed into several sub-problems but the resolution of each sub-problem can take an undefined amount of time. However, they differ from decomposable-delayed services in that for each sub-problem another web service is called on. Holiday booking portals like `ebookers.com` are a good example of this category. When a user searches for a flight and a hotel room, the website has to check what flights and rooms are available for the selected dates. If the user then selects one offer and decides to book it, the portal first needs the re-check with the airline to ensure that a place is still available on the selected flight and if so to book it. Once the flight is booked, the portal needs to do the same request with the hotel (or vice-versa). If for whatever reasons the booking for the hotel room fails, the booking for the flight is not needed anymore and the transaction has to be rolled back. Therefore, it is not sufficient to model this situation with tasks as previously discussed and set the status to *error* or *aborted* instead, another process is started to clean up behind the first one. Such problems can be categorized under business processes.

Wikipedia defines a business process in the following way:

> „A business process or business method is a collection of related, structured activities or tasks that produce a specific service or product (serve a particular goal) for a particular customer or customers. It can often be visualized with a flowchart as a sequence of activities with interleaving decision points or with a Process Matrix as a sequence of activities with relevance rules based on data in the process.“

This perfectly fits the example above where the sequence of activities included the booking of a flight and the reservation of a hotel room with the interleaving decision point where both bookings need to succeed before they can be committed. The combination of business processes and smart devices evokes three major questions: (1) How can business processes take into account and benefit from the capabilities offered by smart devices? This question is not targeted at the WoT, but on a larger scale addresses the IoT. (2) How can RESTful web services seamlessly be integrated into BPM? Although, talking about RESTful services in general, solutions to this question also apply to the WoT. (3) How can existing business processes leverage their capabilities to the WoT?

There are many ways to embed business processes into information systems. Some of them are closed source but most respect common standards like BPM and SOAP web services. Generally, business processes are outside the scope of the IoT (and, by extension, the WoT). This is only partly due to the fact that business processes mostly rely on SOAP web services instead of other types, and RESTful ones in particular. The other reason is that most business processes don't take smart objects into account. The BPM specification simply does not know how to represent a smart device. According to the reference model for the IoT (IoT-A) [78, 29, 78], a smart device is roughly the composition

of some hardware (e.g. sensors, actuators and tags) plus the way this hardware is exposed over the network. In her work [53, 52, 47], S. Meyer proposed an extension to BPM which handles smart objects and makes them addressable from within business processes. In order to achieve this goal, she proposes some extensions to the current BPMN standard to deal with the special properties of smart devices. One extension is a new type of BPMN swimline representing a smart device. This extension does not ask for a specific service architecture. Therefore, any smart device, regardless of its interface, fits into such a swimline. Although this broadens the applicability of the extension, it hinders its usability for the WoT.

Others [61, 62, 43] consider the question the other way around. Instead of thinking how RESTful services and the WoT can take advantage of business processes, they propose BPM extensions for RESTful services. Whereas the presented work is valuable in terms of how to achieve business processes for REST and how RESTful web services can be considered when modeling business processes, it does not answer the question of how already existing business processes can leverage their computational power to the WoT.

In our paper [69], we describe how some types of business process can be integrated into the WoT. The proposed architecture only applies to business processes with a clearly defined start and end and which produce a result worth feeding back into the WoT. Instead of creating RESTful interfaces for all the components involved, only one additional generic component is created. Similar to the solution proposed for delayed and decomposable-delayed services, the additional RESTful service offers only one type of resources: a *Task*. This task resource is a gateway to the underlying business process execution layer. Therefore, it takes as input a model (in BPM) which is then executed by the execution layer. The task resource represents and monitors this process in a RESTful way. Regarding its implementation, the same principles as for delayed and decomposable-delayed services apply. This means that the structure of a task follows that presented in Figure 8.4. Listing 8.5 shows how a representation of a task can be retrieved from the server and the response is shown in Listing 8.6. Since BPM files are XML files, the input and output field are Base64[1] encoded.

```
ruppena@tungdil:~$ curl -X GET -H "accept: application/xml" http://diufpc46.unifr.ch/
    ↪ jetty/TaskExecutorServer/resources/tasks/502
```

List. 8.5: Request to the task executor service for Task 502

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<task xmlns="http://domain.ehealth.digsim.homelinux.org"
    ↪ uri="http://diufpc46.unifr.ch/jetty/TaskExecutorServer/resources/tasks/502"
    ↪ status="FINISHED">
  <id>502</id>
  <userid>1</userid>
  <outputs>
    <output>
      <value>
      aHR0cDovL2RpdWZwYzQ2LnVuaWZyLmNoOjgwODAvQW5hbHlza
          ↪ XNTZXJ2ZXIvcmVzb3VyY2VzL2FuYWx5c2VzLzExMjgyOTk1MS8=
      </value>
    </output>
  </outputs>
  <input>
```

[1] http://en.wikipedia.org/wiki/Base64

```
13   PCFbQ0RBVEFbDQo8P3htbCB2ZXJzaW9uPSIxLjAiIGVuY29kaW5
         ↪ nPSJVVEYtOCI/Pg0KPGRlZmluaXRpb25zIHhtbG5zPSJodHRwOi
14       ↪.....
15   </input>
16   <startdate>2012-05-30T09:31:18+02:00</startdate>
17   <enddate>2012-05-30T09:31:48+02:00</enddate>
18 </task>
```

List. 8.6: Sample output when requesting the TaskExecutorService for Task 502

Listing 8.6 shows how the task specification from Figure 8.4 can be adapted to a particular use-case. Here, the *output* element is slightly different from that of the specification. Instead of producing only one output, a business process can generate several outputs (for example each step of the instantiated business process can produce a result). Since all these results belong to the same task, it can be split up into subtasks, each one representing one result and so using the structure as defined in Figure 8.4. Yet, it is up to the business process execution layer to notify the task interface about the state of a process and possible results. Tracking sub-processes in a consistent manner would probably be too hard. Therefore, only one resource per launched process is preferable but with several outputs. The XML of Listing 8.6 still conforms to the specification; it contains one element representing the outcome of the process. The only difference is that this output elements further splits into sub-outputs. The listing shows that the outcome of the business process is a new resource on a distant service. In fact, decoding the Base64 string of line 7 results in the URL `http://diufpc46.unifr.ch:8080/AnalysisServer/resources/analyses/112829951/`

To produce the output of Listing 8.6 we imagined the following scenario and implemented the involved components as a proof-of-concept. In a modern hospital, nurses and doctors are supported in their tasks by many smart devices that measure and report a patient's condition. Such a scenario is not too far from current reality. Already today the success of health monitoring products like Nike+ FuelBand [WEB36] or the wristbands produced by FitBit [WEB18] prove its feasibility. During a patient's stay in hospital, a nurse checks his condition on a regular basis. This involves taking measurement like blood pressure, blood oxygen and heart rate. These values are then reported in the patient's medical record. Furthermore, if some of the measurements are outside a given zone, which partly depends on the patient, some action needs to be taken. For example, if a patient's heart-rate suddenly drops to zero, he should be resuscitated. Although it would be possible to model such use-cases with mashups, business processes would be a more consistent choice. Mashup application only deal with RESTful services, however, the above process also involves humans at some stages, which are outside the scope of mashup applications.

First, each patient can be modeled as a resource. Commonly, the xWoT accepts sensors, actuators and tagged objects. According to this, a patient would be a tagged object. The tag can take the form of a wristband just like the one already used today. By reading the tag, nurses and doctors can access the virtual representation of the patient. This includes his medical record, past analyses, drugs, medical pre-conditions and so on. Since this information is the virtual representation of a patient, it is available over a RESTful interface. Making analyses is another important activity. In [72] we present an alert escalation system for medical analyses. Depending on the type of analysis, its execution can be fully automatic or require some human intervention. Still, it is possible to represent an analysis in the virtual word, the important concepts are not the sensors and actuators

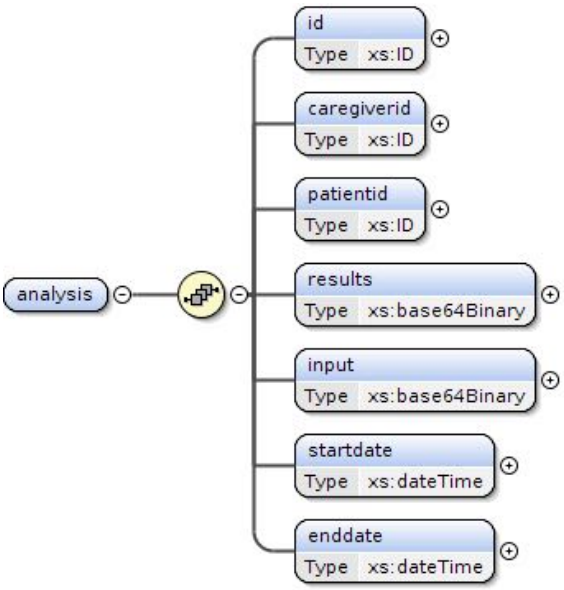


Fig. 8.5.: XML schema for analysis

involved during the execution of the latter, but the gathered and aggregated information. Such an analysis resource is the second pillar. Again, the system implemented makes only few assumptions about an analysis and the proposed input and output attributes, as shown in Figure 8.5, are reduced to a strict minimum. Figure 8.6 shows the HTML representation for the analysis, which is the result of the task of Listing 8.6. The third pillar is formed by an alert escalation system as we describe it in [72] and the fourth is the task-executor as described above.

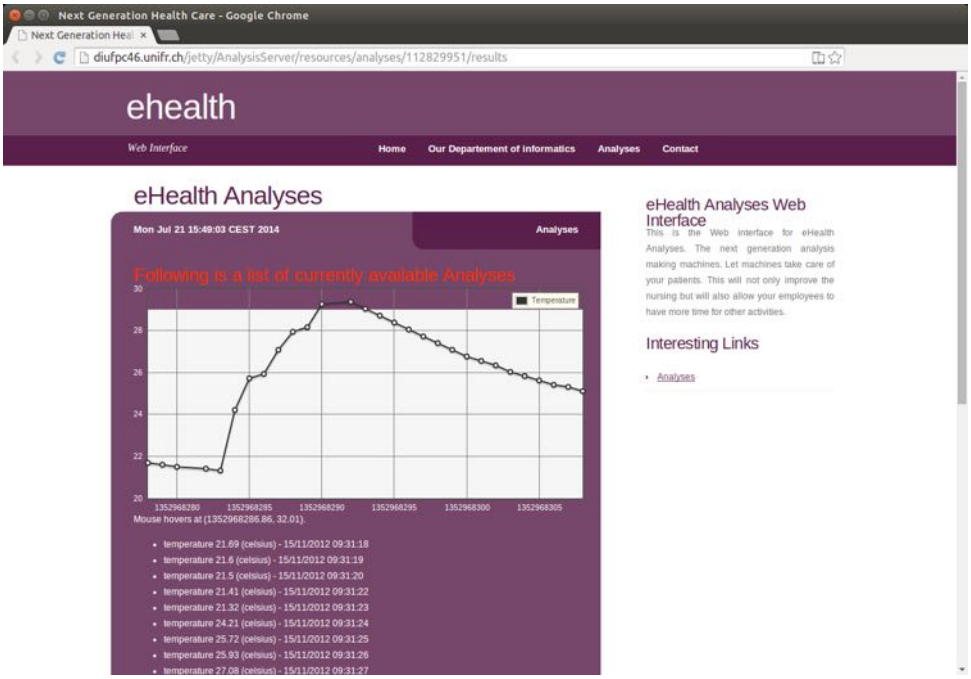


Fig. 8.6.: Result of a business process monitoring a patient's temperature

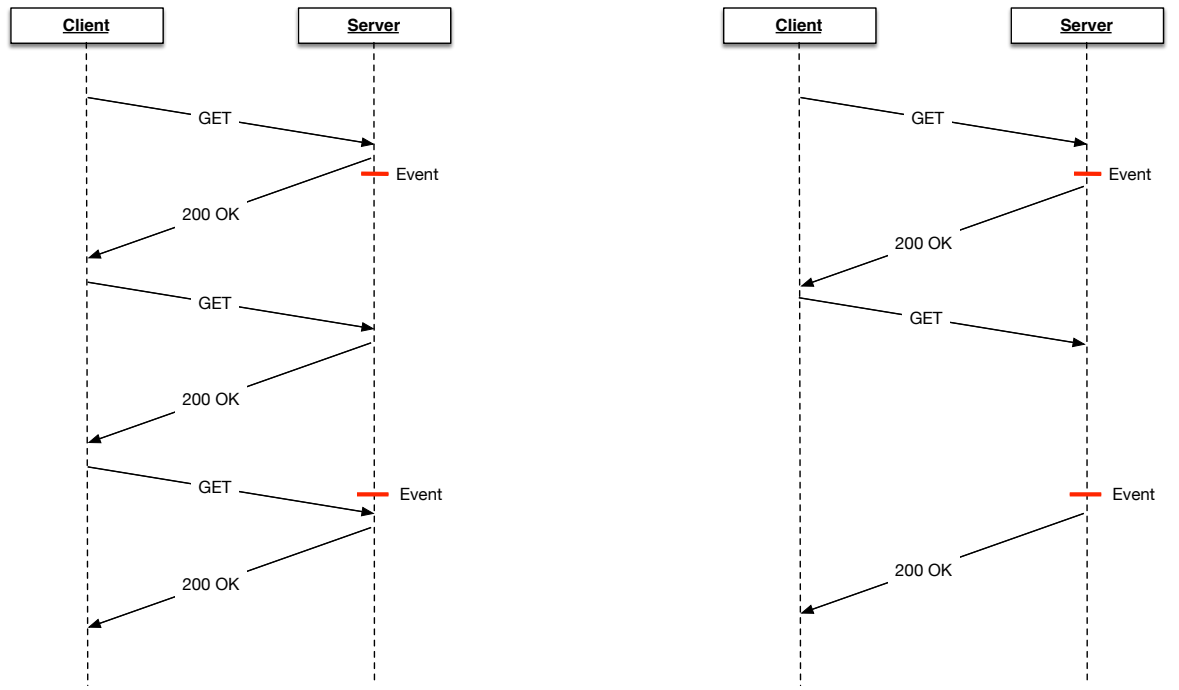
Based on these four resources, let us consider the following scenario. The first night after successful surgery, a patient's vital signs need to be monitored. Often this is not

continuous monitoring but samplings at given intervals of time. Upon each execution, each sensor has to report a few readings; these readings are stored in an analysis sheet, which is attached to the patient's medical record. Furthermore, the reported values are checked against a model to determine whether the patient's condition is good. If abnormal values are found, an alert is created which is then handled by the alert escalation system to ensure somebody takes care of it. Clearly, this scenario involves the four resource presented plus a few smart devices measuring different health related aspects. Although all the gathered information can easily be combined in a mashup application, this would not be the right choice for this scenario unless the mashup application comes with some type of scheduler. Hence, it is a better choice to let a business process take care of monitoring. Pautasso [61, 62], Kumaran [43] and Meyer [52] all propose an approach which makes it possible to model such scenarios in BPM. Finally, the last missing part is an execution engine capable of taking advantage of one of these input formats. Currently, such an engine is under development to support the extensions proposed by Meyer and supported by the European IoT-A project. For the sake of this proof-of-concept, the execution engine is only a mock-up, taking as input a BPM file, and always executes the same process. Therefore, in this scenario, the nurse creates a new instance of the surveillance BPM tailored to the patients. This involves changing some bindings so that the execution engine knows which patient to monitor and which smart devices are connected to any particular patient. The nurse then creates a new task by uploading this BPM to the task service. Upon receiving this message the task-executor will create a new task representing this business process. Furthermore, a new process is started by the execution engine. Listing 8.5 shows how such a task is created and Listing 8.6 shows the corresponding response from the task-executor. Since the business process involves monitoring a given patient, a new analysis resource is created which will hold all observations made by the process. Finally, the resource on the task-executor representing the business process points in its `result` field to the newly created resource. Additionally, the business process takes care to deduce the patient's medical condition and, if necessary, launches an alert. The system works as one would expect from any business process engine, but the generated output is fed back into the RESTful system making it available for other applications.

8.2.2. Server-Side Information Pushing

The previous section introduced a classification for RESTful services. Only short living services deliver their results immediately. All other types of service require the user to check on progress regularly. If these services could inform the user about their progress, the ergonomics would drastically improve. Repetitive checking is even more questionable for real-time services as they need a mechanism to push new information back to their subscribers. Regarding sensors, users are sometimes more interested in knowing that something has just happened than in knowing what happened. The previous section solves the *WHAT* problem well, but completely ignores the *WHEN*. This is not an architectural fault but rather a limitation of the underlying technology stack. The web is based on the client-server paradigm. This means that for each exchange it is possible to clearly identify which party is the client and which the server. Furthermore, only the client can contact the server. This limitation is only partly due to technical reasons. To overcome the limited number of IP addresses, Internet providers introduced NAT, which allow multiple

devices to share one IP address. The drawback of this technology is that devices sitting behind the NAT cannot be contacted from the outside. Although this approach seems very limiting, it also has several benefits. By shielding the internal network from the outside, NAT provide an additional layer of security and privacy. For some time now, the web has been switching from IPv4 to IPv6, solving the address space problem. Still, NAT is and probably will remain part of the topology of the web, and needs to be dealt with. This section discusses some common approaches to how this client-server limitation can be circumvented to send events to subscribers and inform them about the *WHEN*.



(a) Classic Polling. The client actively polls the server for updates

(b) Long Polling. The server keeps the connection open and answers the request only as soon as new data is available

Fig. 8.7.: Pushing information through polling

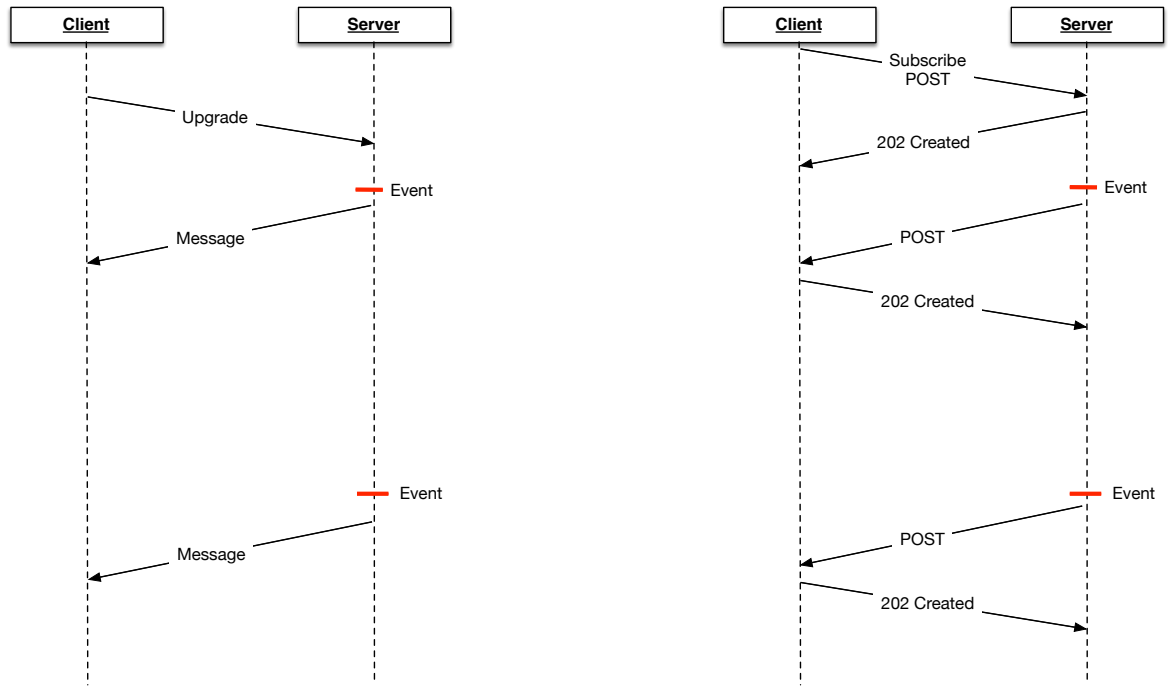
In client-server architectures, the client always initiates the communication. The server has no means of contacting the client later again. Since the WoT uses RESTful architectures, it is limited to client-server interactions. Based on this paradigm, several solutions to overcome this limitation have been proposed over the years. Roughly, they can be classified into two categories: (1) those based on polling (2) those not using polling. Approaches using some sort of polling are easy to develop and are very robust. In their simplest form, a client issues a `GET` request at regular intervals of time. Each time, the server responds with a representation of the requested resource. As changes occur occasionally, consecutive requests may result in the very same representations. Since there will be - at most - a fixed amount of time between two requests, this approach meets the real-time criteria. A user will, in the worst case scenario, be informed about the change after this fixed amount of time. As simple as this approach is, it comes with a set of drawbacks. Clearly, this approach has high bandwidth consumption. Whether there is a change or not, the client issues a request which has to be carried out by the server. Although, the situation can be improved by using advanced caching approaches,

there will always be some overhead when nothing changes between consecutive requests. Furthermore, this approach also has a cost on the server side expressed as either wasted CPU cycles or costs in battery lifetime etc. Polling is certainly not the right approach if the client needs to be informed about every change. However, if the client can live with a relatively high interval between requests, it might be an acceptable solution, since no additional component needs to be developed and/or deployed in order to make it work.

However, this is only true for classical polling, as shown on Figure 8.7a. In order to improve the response time of classical polling, *long polling* was proposed. The differences between classical polling (Fig. 8.7a) and long polling (Fig. 8.7b) are only minor. In Figure 8.7 the difference is only visible in the amount of time between the incoming request and the response. This is also the main difference between the two. As depicted in Figure 8.7b, long polling involves a slightly modified timing when responding to incoming requests. As for classical polling, the client issues a `GET` request to retrieve a representation of the resource. However, instead of doing this, the server keeps this connection open and uses it as soon as the resource changes and new information is available. Long polling is thus more reactive than classical polling. Not only does the server already have an open connection, but the timing is also better. Nonetheless, long polling suffers from most of the drawbacks already discussed for classic polling. It might seem, like there is no overhead at all when using long polling. This might be true if updates are more or less frequent. However, as soon as updates get less frequent, either the connection drops or the server closes it before it drops. The client then needs to open a new one. The dropped connections introduce an overhead in bandwith and also CPU consumption. The number of connections that drop not only depends on the frequency of updates, but also on the client and server configuration. Additionally, long polling can block a considerable amount of memory on the server side. For each connection, a small amount of memory is reserved during the exchange and liberated afterwards. Thus, one long polling client constantly occupies som ememory, which considerably limits the quantity of clients that a server can accept. Although polling is robust and comes with no additional development costs and always works as it is intended to, it should only be used rarely, or as a backup solution.

ATOM [rfc5023, rfc4287] and RSS [WEB51] are other popular approaches for users to subscribe to events. Such feeds of events are mostly used by blogging sites. Instead of regularly checking the blog for updates, a user can subscribe to its feed and receive new articles directly in his feed reader. Judging from the number of feed readers available for different operating systems and mobile operating systems, such software is quite common and seems popular. This is surely a reason why the WoT also considers feeds a valuable way of informing clients of updates. Yet, feeds are neither a technology nor an architecture; RSS is an XML dialect and ATOM is a protocol/format. When a client subscribes to a feed, its newsreader regularly polls the feed endpoint for new entries. Therefore, neither ATOM nor RSS can solve the information-pushing problem better than polling.

Still based on pure HTTP, most modern web-servers implement some streaming mechanism which allows it to push information back to clients. Several approaches exist to achieve this goal. Most imply a special *content-type* header indicating that the document is not yet fully delivered. This fools the client, which keeps the connection open to receive the remaining part of the document. Typically the server uses the header `Transfer-Encoding: chunked` and strips the `Content-Length` header. Therefore, a client does not know how long the received document is and keeps the connection open



(a) Pushing information through a WebSocket connection

(b) Pushing information through a WebHook provided by the client

Fig. 8.8.: Real Push

until the last part with size 0 is received. Another technique is based on the MIME type `multipart/x-mixed-replace` introduced by Netscape in the 90's and still supported by many browsers. Based on these approaches, HTML5 introduces a streaming API which uses Server Side Event (SSE) as a back-end driver. This approach is commonly used in modern web-application frameworks like Ruby on Rails to create web sites which feel more like native applications.

A completely different approach is proposed by the WebSocket specification [rfc6455]. Although WebSocket interactions do not necessarily follow RESTful principles, it is a common protocol to push information in the WoT [WEB8]. Rather than trying to miss-use HTTP to push information between clients and servers, WebSockets open a full-duplex TCP connection. Looking at the timings of WebSocket communications in Figure 8.8a, major differences appear compared to polling mechanisms. The most important one is that the client only sends one request. Commonly, this is a special HTTP request with the header `Upgrade: websocket`. The server acknowledges the request to upgrade and both switch from HTTP to WebSockets. Once the channel is open, both parties can send and receive messages. Additionally, messages can be broadcasted on channels to which users can subscribe. Therefore, one message can be sent to multiple recipients at once. Since a WebSocket is nothing more than a bare TCP connection, it is up to the developer to specify which protocol to use. It would be possible to use HTTP over a WebSocket (which does not make much sense). It is also possible to create completely new protocols running over WebSocket. However, for the sake of pushing information, there is no need for any protocol. Commonly, the WoT proposes different representation for each resource, like JSON, XML and HTML. Since notifications for events mostly make

sense in computer-to-computer interactions like mashup applications, it makes sense to use one of the B2B (Business-to-Business) oriented representations like JSON. Figure 8.9 shows the typical output from a WebSocket connection. Here, the WebSocket endpoint represents a temperature sensor.

WebSockets not only solve many problems which other approaches have, they are also very efficient. In [B14] Lubbers et al. talk about the many advantages of WebSockets over other pushing architectures. In their book [B14, p. 143], the authors evaluate the performance of WebSockets compared to classical long-polling and conclude that already for an application scenario involving 1000 users, each issuing one request per second, the reduction in bandwidth is drastic. According to their setup, the drop in bandwidth is a factor of 400. Of course, values are scenario-specific and vary. Yet, it remains true that overall, WebSockets perform better than any sort of polling.

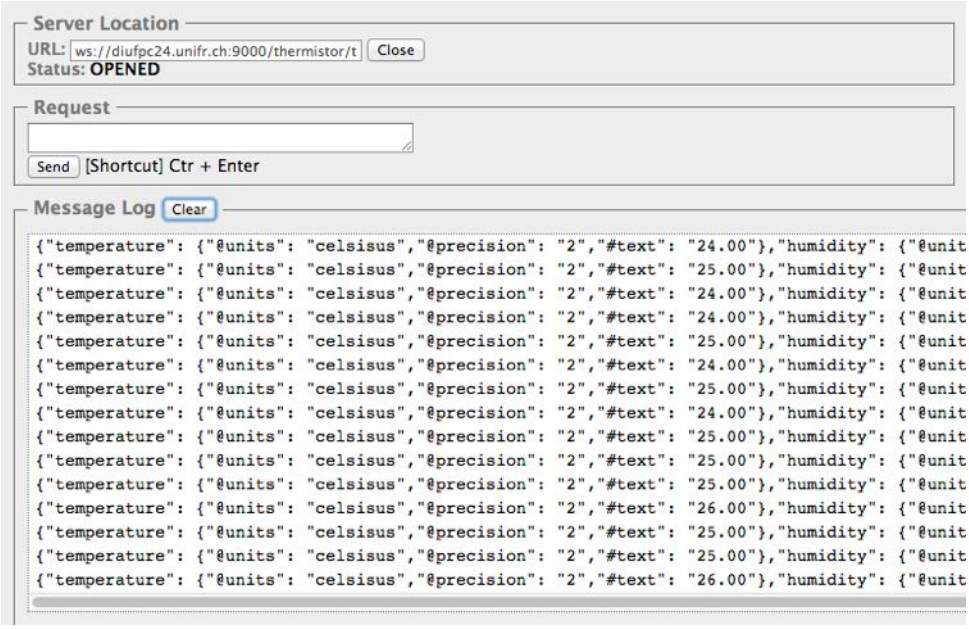


Fig. 8.9.: Pushing updates over a WebSocket connection

Through a small example of a wind sensor, Pimentel et al. [65] also show that the WebSocket protocol has a lower latency compared to other polling mechanisms. Since WebSockets are a raw full duplex TCP connection between two pairs (no protocol is mandatory), it is quite normal that its performance outranges exchanges based on a strict protocol. Furthermore, they show that long polling performs better in terms of latency than classic polling, which feels quite natural too, since the connection is ready to use when an event occurs.

Compared to polling mechanisms and other streaming approaches, WebSockets have a smaller overhead. Furthermore, new events are pushed in a timely manner to a client. Additionally, real publish-subscribe scenarios are possible and the server can inform many clients listening for one topic. Although WebSockets seem to be the perfect match when it comes to pushing information they also suffer from some drawbacks. If events only happen very rarely, the open connection will drop after a defined amount of time. One solution to prevent connection timeouts is to use some sort of heartbeat. Yet, when connection timeouts are an issue, heartbeats often introduce a considerable amount of overhead.

A completely different approach is client-server interactions for events. By switching the roles, a service-endpoint becomes a client of a resource provided by its user. Therefore, this interaction reverses the control between its pairs. For this to work, the service has to propose a special resource where clients can subscribe for events. Upon subscribing, each client provides at least one URI where he can later be contacted. Generally, this URI is linked to some simple service accepting `PUT` and `POST` requests. Each time the server generates an event, it cycles through the list of subscribers and for each, issues a `PUT` or `POST` request on the provided URI. In the literature, this approach is called webhook or URL callback. [27, 60, 22, WEB72]. Figure 8.8b show the sequencing involved in webhook publish-subscribe scenarios. Figure 8.8 shows that both approaches, WebSocket and reverse control are efficient. Both push the information about a new event in a timely manner to clients. Whereas WebSockets are more efficient when events occur frequently, they lose these advantages when events only happen rarely. In this case, an inverted client-server approach performs better. This is also true when comparing CPU consumption. For a WebSocket, the connection is kept alive all long whereas inversed client-server interaction only open a connection when there is data to send.

It seems clear that the best pushing infrastructure is one where the server can push information as soon as it becomes available, where there are no communications overhead, especially during silent times and where a communication channel is only open as long as it is worth it. Sensors in particular, can generate a lot of events. Therefore, the pushing infrastructure should also provide a simple means of filtering which events are to be pushed to a given client. These remarks are valid for any pushing infrastructure, and thus also for the xWoT. Based on this analysis, each xWoT component incorporates WebSockets and webhooks as pushing infrastructure. WebSockets can be used to push bursts of events to subscribed clients. This would be the case when following a rapidly changing phenomenon over relatively short amounts of time. For WebSocket connections, all events are pushed to all clients. There is no means of filtering the events per client. On the other hand, webhooks are great to push information about isolated events to clients in a timely manner. This would, for example, be the case when subscribing for events about the open/closed state of a window. Upon subscribing, the user advises the URI of his webhook, whether he wants `PUT` or `POST` and a short specification of which types of events he is interested in.

8.2.3. Light Bulb Example

Through a small example scenario, this section demonstrates why the service classification of Subsection 8.2.1 and the different information pushing mechanisms of Subsection 8.2.2 matter for an extended WoT (xWoT). Throughout the section, the example will grow to encompass different aspects of the xWoT. To start, consider a simple light bulb. Its main characteristic is that it can be switched on or off. This is usually done with a light switch deployed somewhere on a wall. For now, this light bulb only exists in the physical world. Interactions require physical cooperation and the presence of an operator (usually a person). To embed this light bulb in the WoT, it needs a virtual side, to represent it in the virtual world. Such a simple entity as a smart bulb can be represented through one resource. Using this resource, a client can switch the light on or off by sending `PUT` requests. The payload sent with each request is the new representation. For a smart light bulb this could look like what is shown on Listing 8.7. Basically, there is just one piece

of information, whether the light is on or not. This service is wired up with an actuator sitting in the physical world and replacing the human pushing the light switch.

```
1 {'light': 'on'}
```

List. 8.7: JSON representation of a smart light bulb

So far, the scenario is like most Things found in the WoT: an everyday object made smart by adding some electronics and a RESTful interface over which users can interact with the Thing. While switching the light bulb on and off over a RESTful interface is easily achieved, interesting scenarios and mashups need a little bit more to work with. To stay with the smart light bulb example, it would be nice to find out its current state. In the physical world, getting a light bulb's state is quite obvious and mostly does not require the active interaction of a person. We call this passive observation. However, this is not true for the virtual counterpart. A client cannot tell a light bulb's state by gazing at its RESTful interface. In order to make this information available to the virtual counterpart of the light bulb, two things are needed: (1) a physical sensor, measuring the light bulb's state and (2) a resource exposing this sensor to the virtual world. The only action making sense on the sensor resource is a `GET` returning the current state of the light bulb (as in Listing 8.7). Table 8.1 resumes the situation and also reveals the first problem: the WoT is made of sensors and actuators. Actuators act on a physical property and so accept commands. Sensors, on the other hand, measure a physical property and serve the results over their associated resource.

Resource	Available Methods
Light Switch	`PUT`
Light Sensor	`GET`

Tab. 8.1.: Resources and methods for the smart light bulb

Smart Things such as this light switch form the most basic bricks of the WoT - just one actuator or sensor tied to a RESTful interface. Yet, depending on the scenario, it is possible to model more complex situations. Imagine that a motion sensor triggers the light switch. Each time somebody enters, the light is switched on (if it is not already on). Soldering such a device is a rather easy task. Starting with the hardware from the smart light switch, it is sufficient to add another sensor to the board measuring human presence (like a passive infrared sensor). On the virtual side, nothing changes for the light switch. However, the presence sensor is represented through a new resource. Being a sensor, it accepts `GET` requests. Listing 8.8 shows a typical response from the presence resource. In this case, the sensor detects that somebody is present, which is reflected through the `true` statement returned in the representation.

```
1 {'presence':'true'}
```

List. 8.8: JSON representation of the presence sensor

In the current scenario, the light is switched on if somebody is in the room. On the hardware side, this implies that the light switch actuator is notified about some event. So far, the event of detecting a presence is only available to the hardware and used as a signal between the different hardware parts. Yet, it would be useful to provide this information on the virtual counterpart too. To push detection events to clients, any

method discussed in Subsection 8.2.2 can be used. It is the individual use-case and the expected frequency of the event which will determine the most suitable approach. The new situation is resumed in Table 8.2. For the publisher, the methods are not specified further as they typically depend on the chosen approach. Long polling would imply a `GET` request, whereas WebSocket connections use the `WS://` protocol. Therefore, it is left up to the implementation to define the available methods for this resource.

Resource	Available Methods
Light Switch	`PUT`
Light Sensor	`GET`
Presence Sensor	`GET`
Presence Publisher	various

Tab. 8.2.: Resources and methods for the smart light bulb with a presence sensor

Returning to the initial use case with a simple smart light bulb, imagine that instead of having just one smart light bulb there are dozens of them. Each one offers the same RESTful interface summarized in Table 8.1. This allows a client to control each light over its own RESTful interface, each deployed on its own server. While a mashup application could put all these services in one neat interface giving the user easy control over all the lights in the house, it is also possible to rethink the RESTful interface and come up with one adapted to the current use-case. This is a rather simple example task: the scenario has individual light bulbs as described at the very beginning of this chapter. Furthermore, the scenario implies some list of light bulbs as the main entry point. This leaves the hierarchy depicted in Figure 8.10. The topmost resource is the list of available smart light bulbs. It supports `GET` requests and returns a list of links pointing to individual smart light bulbs. A unique identifier identifies each light bulb. These `{id}`s form the second level of the RESTful interface. Again only the `GET` method is supported at this level. As discussed at the beginning of this section, a smart light bulb offers two resources: the light switch actuator and the light switch sensor.

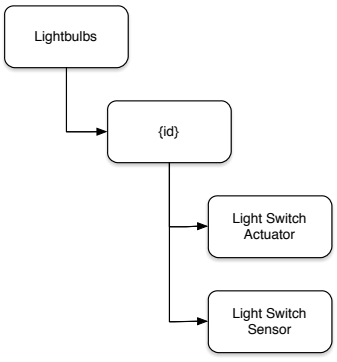


Fig. 8.10.: RESTful hierarchy of light bulbs

Modeling the scenario as depicted in Figure 8.10 allows the user to browse all available light bulbs without any need for an additional mashups. Nonetheless, the proposed model does not bar the creation of such mashup applications. Everything below the layer two of Figure 8.10 is strictly the same as summarized in Table 8.1 and can therefore be integrated into any mashup application. As already explained in the first part of this section, a client

can check whether a light bulb is currently switched on or not. Therefore to know whether there is some light in the house or not, he needs to ask all available light bulbs if they are switched on or off. If at least one of them is on, then he can conclude that there is some light in the house. This global state can also be exposed over the RESTful interface. It is sufficient to add a new resource as a child to the top-level *Lightbulbs* resource. Unlike the other resources, which are either lists or directly map to some hardware, this resource maps to a virtual only good, namely an algorithm. Figure 8.11 shows this new resource called *Status*, and accepting at least a `GET` request. Such a service perfectly fits into the *Short Living Services* category.

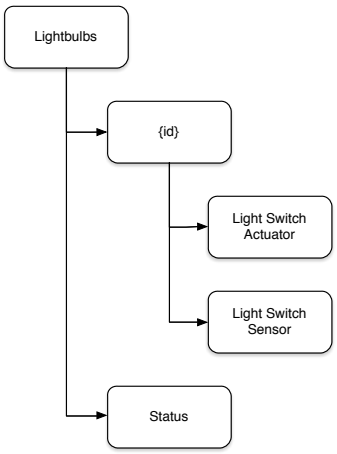


Fig. 8.11.: RESTful hierarchy of light bulbs revisited

The light bulb example and its use-cases show that the boundary between what is offered by a RESTful interface and what is accomplished in a mashup application is fuzzy. It seems that at least the core functionality of each use-case should be offered by the service itself. Moreover, when discussing the meta-model for the xWoT, we will see that different point of views lead to different RESTful APIs, even though a few components are present in all APIs. Therefore, when creating new RESTful interfaces, one crucial question is the point of view to adopt, as this will be directly translated into the RESTful API. This is especially true for use-cases involving some sort of algorithm as in the third scenario of this section.

8.3. The extended WoT

Today, the WoT encapsulates smart devices with a RESTful interface so that they can be embedded into the web and allow clients to interact with them through their browser. On the other hand, there is the rest of the classic Web as used by millions of people every day. This part of the Web includes social media websites like Twitter, Facebook, Foursquare and so on, dictionaries and encyclopedias like Wikipedia, blogs and many homepages. Finally, the web has also come up with a number of pure services. They neither belong to the WoT nor to social media, dictionaries or blog. Instead they are pure services. Starting with some input, a more or less complicated computation is done, eventually leading to a result. The website `http://checkip.dyndns.com` is such an example. Figure 8.12 and Listing 8.9 show the result of a `GET` request to this URL. While the dyndns web site offers

a service in its simplest form, there are many other websites providing some sort of service: postal zip code searching, weather etc. Additionally, some social media and dictionary websites provide in parallel an APIs, turning their websites into fully fledged services. Although these three types of web seem to be completely different and unrelated, they are not. They all use the same protocol, HTTP, as a fully blown application protocol. They are all, at least partly, embedded in the web used by humans and they all have in common that they expose some type of information. Already today, the WoT heavily relies on some of these parts of the web. Mashup applications like the *Japan Geiger Map [WEB25]*, the *Live Flight Tracker (Figure 8.1)*, the *EPCIS Mashup [57]* underline the importance of them.

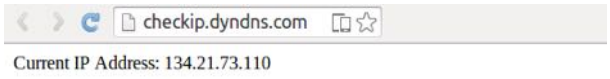


Fig. 8.12.: DynDns Current IP Check

```
<html>
  <head>
    <title>Current IP Check</title>
  </head>
  <body>Current IP Address: 134.21.73.110</
      ↪ body>
</html>
```

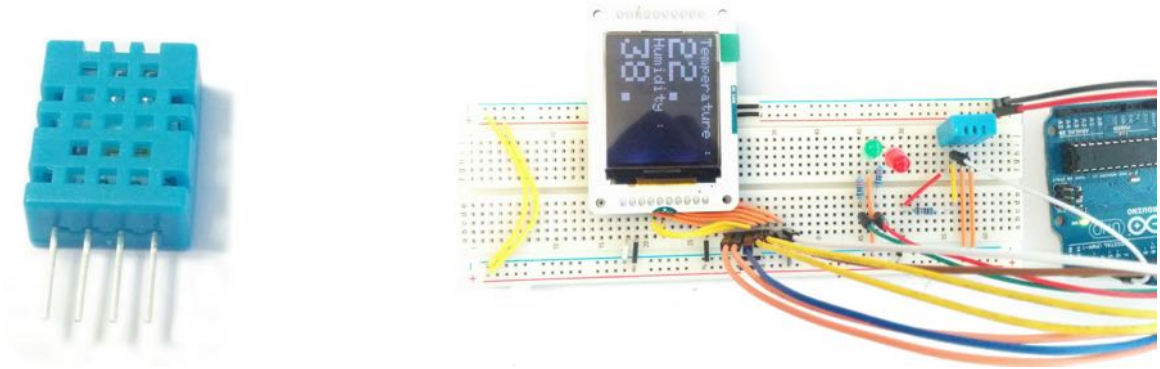
List. 8.9: DynDns Current IP Check HTML code

The xWoT embraces all these different types of service and combines them together. It does not matter if a service has a physical counterpart, if it is a social media web service or something else. As long as it offers a RESTful API it is potentially of interest for the xWoT. Therefore, the xWoT is a web where Things, but also purely virtual services are treated as first class citizens. Whereas the web encompasses all available services, the xWoT is a subset of the web only taking into consideration services related or of interest to smart devices. Before formally introducing the xWoT, we need to define the available bricks composing smart devices. These are mainly actuators, sensors and tags but also virtual hubs. The remainder of this section addresses each component individually and finally formally introduces the xWoT.

8.3.1. Sensors

A sensor measures some physical property. Generally, sensors do not act on their physical environment (although this is sometimes controversial [WEB53, WEB54]). As such, a sensor is a passive resource observing some phenomenon. It generates data, which is delivered over some interface. A client can query this interface to get the latest measure.

Translating this to its corresponding RESTful interface implies that a sensor needs at least to serve `GET` requests. This is also the general way clients interact with it. If the client wants to know the temperature, he can `GET` this information from an available temperature sensor (for example one of the form of Figure 8.13). Often this is the only available action. Since a sensor cannot change or influence the phenomenon it observes it also does not make sense for its RESTful interface to accept commands via `POST`, `PUT` or `DELETE`. There is only one exception to this rule; sometimes it is possible to configure and fine-tune a sensor. To stick with temperature sensors, temperature can be expressed in at least three ways: Kelvin [WEB28], degree Celsius [WEB7] and Fahrenheit [WEB15].



(a) A simple DHT11 sensor

(b) Fully working temperature and humidity sensors

Fig. 8.13.: Sample Sensors

Although, according to RESTful principles, each client can request any desired format, it is often convenient to define a default one. Such configuration tasks are generally done with a `PUT` request.

From a behavioral point of view, both sensors and short living services 8.2.1 behave the same way. Moreover, a client is unable to tell the difference between a service having a sensor as backend and another without. For this reason, they should be treated the same by architectural tools.

8.3.2. Actuators

An actuator can be seen as a black box which takes some input and depending on it, executes some action. In the case of a servomotor like the one in Figure 8.14, the input would be the incoming voltage. Depending on the applied voltage, the motor can speed up and by turning will have an effect on its environment. Additionally, actuators have a state. This can be as simple as a discrete binary value (on/off) or it can be a continuous one like the voltage of a servomotor.

Translating this into RESTful terms involves each actuator being represented in the virtual world through a resource. Commonly, a client sends commands to the actuator, which will respond in an adequate manner. Furthermore, this response is twofold: the resource sends back an HTTP response plus the actuator physically executes some action. Therefore, the resource representing such an actuator must at least respond to `PUT` requests. The payload of the request contains the commands to execute. For an actuator with only discrete states, this would be the new state to adopt; for actuators accepting continuous values, this would be the new value. If the state of the actuator is important, clients can query it through `GET` requests. The response to a `GET` request is the same as the payload of a `PUT` request. Therefore, the `GET` request can also be used to learn about the device and learn about the accepted payload format.

Actuators do not use `POST` and `DELETE` requests. Since each actuator is anchored in the physical world, deleting the resource would also imply deleting its physical manifestation, which obviously doesn't make sense. The same argument is valid for `POST` requests;

Fig. 8.14.: A Brushless motor used for quad copters

creating new resources would also imply creating new physical instances, which for the same reason is impossible.

8.3.3. Hubs vs. Mashups

Although raw sensors and actuators form the most basic elements, some use-cases depend on more sophisticated smart devices. We have already discussed the temperature example several times. Although it is clear that the temperature is measured by a thermistor and virtually represented through a resource, a use-case can involve fuzzy temperature definitions. Imagine a smart home full of sensors and actuators, including several temperature sensors in all the rooms in the house. Even if it is possible to `GET` the temperature reading of each individual temperature smart device, the user still would not know the overall temperature in his house. This comes from the fact that there is no physical device returning the overall temperature; instead this information is obtained by combining the information of different smart devices. In our scenario, this would be the mean temperature reading of all available smart thermometers.

Definition 10 (*Hub*)

> A hub *is a virtual device which acts just like a real one. For a user, it is impossible to distinguish between a hub aggregating several sensors and/or actuators and a raw sensor or actuator.*

Mashups or mashup applications are similar to hubs. They combine information gathered from different sources and present them to a user in an adequate manner. Most of the time, mashup applications come with a fancy GUI intended for users without any programming skills. Therefore, the problem of the overall temperature could be solved either with a mashup application or with a hub. The key difference between mashups and hubs lies in the fact that a hub exposes some information that can be re-used by others whereas a mashup doesn't.

This discussion shows that the separation between algorithms (at least in the form of a hub) and Things outside of the WoT is fuzzy. Although it is possible to separate the web into several distinct parts for classification purposes, there is no reason to keep this

separation for architectural concerns. All these parts rely on HTTP and REST as an architecture. Therefore, the xWoT proposes an approach where not only smart devices are treated as first class citizens but so are services of all kinds as long as they respect the REST principles. From the construction of mashup applications, we have learned that the most useful bricks come with a clean and easy to use interface. From this point of view, it does not matter whether, upon invocation of its RESTful interface, a smart device is performing some task or not. However, how the RESTful façade is invoked does matter. The aim of the xWoT is to define re-usable and easily deployable components acting as building blocks for mashup applications. This involves two aspects: (1) their *skin* and (2) their *inner* guts.

REST architectural style already well defines how the outer interface needs to behave. However, regarding its inner guts, there is no guidance or convention. Each developer is free to choose (and therefore is forced to choose) its own conventions when it comes to structuring the inner guts of a smart device or another RESTful service. What differentiates smart devices from other RESTful services is their inherent structure. A smart device lives both, in the physical and the virtual world. How the device is embedded in the physical world also guides its virtual representation. For resources without any physical counterpart, there is no limitation regarding their inner structure. Therefore, to define the inner guts of components, it is first necessary to analyze the factors influencing the latter. A smart device can either be a sensor, an actuator or a combination of both as in Subsection 8.2.3.

8.3.4. Formal Definition of the extended WoT

The discussion of sensors, actuators and hubs, showed that although different in the physical world, they often share a lot with their virtual only colleagues. This is also the starting point for the extended WoT:

Definition 11 (*xWoT*)

> *The extended WoT is a web made of sensors, actuators and tags forming the classical WoT plus services respecting RESTful principles.*

It seems that Definition 11 contradicts the idea behind the WoT, Things connected to the Web (see Section 4.3). In this perspective, there is no place for services, even RESTful ones, not dealing with smart devices. On the other hand in [70] we have shown that combining WoT Things with purely virtual RESTful services can be a big benefit for the WoT. Meyer at al. discusses algorithms for the WoT too [51, 50]. Additionally, from a consumer perspective, it is almost impossible to distinguish between services representing a physical device and those that don't. This is another reason to consider physical and virtual RESTful services together. Furthermore, sharing the same meta-model between WoT services and other RESTful services guarantees a seamless combination of them. Clients facing such a RESTful interface will instinctively know how to exploit it, whether it represents a smart device or not. These considerations plus Definition 11 allows the aim of the xWoT to be defined.

Definition 12 (*Aim of xWoT*)

> *The aim of the xWoT is to introduce a standard approach on how to design the building blocks for novel applications and mashups exploiting the capabilities offered by smart*

> *things and other virtual goods. To achieve this goal, the xWoT introduces a component-based methodology which is underlined by a meta-model guiding the developers during crucial architectural decisions. Finally, since the architecture respects the xWoT's meta-model, component skeletons are generated out of the specifications.*

Whereas building mashup applications has been, and still is an active research topic [57, 76, 44, 33, 31, 38], the xWoT defines how the building blocks for them need to be created and structured. It furthermore shows that mashup applications are not always the best solution. Sometimes it is better to consider a hub which stays embedded in the WoT. The outcomes of the xWoT are reusable and deployable software components. To support the creation of such components, the next chapter introduces a meta-model plus a methodology allowing to the rapid creation of xWoT compliant components.

8.4. Light Bulb Example Revisited

Subsection 8.2.3 introduced the smart light bulb as an example of a WoT application. Starting with the last situation in Figure 8.11 several components involved are identifiable. Figure 8.15 resumes the involved use-cases. Clearly, some are related to the light bulb, some to motion detection and others have no direct link to the physical world. Additionally, the right side of Figure 8.15 shows that notifications are an important aspect of the smart light bulb example.

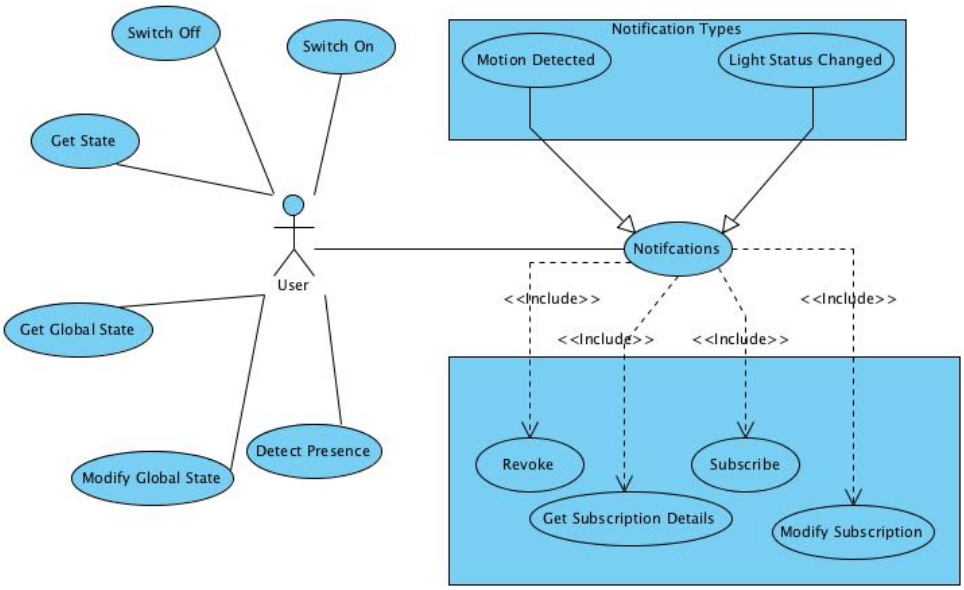


Fig. 8.15.: Use-Case diagram for the smart light bulb example

Since the system involves two physical devices, a sensor and an actuator, there are also two types of notification, one for each. Although notifications are sent from the server back to the client, the latter also plays an active role in these use-cases - to subscribe or unsubscribe for a given type of notification. Since the system cannot possibly know who is interested in what type of notification, the client has to take this first step and actively tell the system which events he would like to receive notifications of.

Now that the use-cases have been defined, it is time to turn them into deployable and reusable xWoT components. If the system is one isolated light bulb, the question does not really matter. Either way will be good enough. However, things change when multiple light bulbs need to be made smart. Translating the requirements of Figure 8.15 into one big RESTful interface will not work as this would lead to a situation where each smart light bulb offers its own version of *Get Global State* and *Modify Global State*.

One can argue that it is sufficient to create small smart devices with any RESTful interface and combine them with a mashup application to fit the requirements (e.g. `Get Global State` or `Modify Global State` use-cases). Although, this works, from a software engineering point of view, this approach is inefficient since components are not easily reusable. A mashup application is tailored to exploit a given set of actuators and sensors. However, its functionality cannot be re-used in any other use-case.

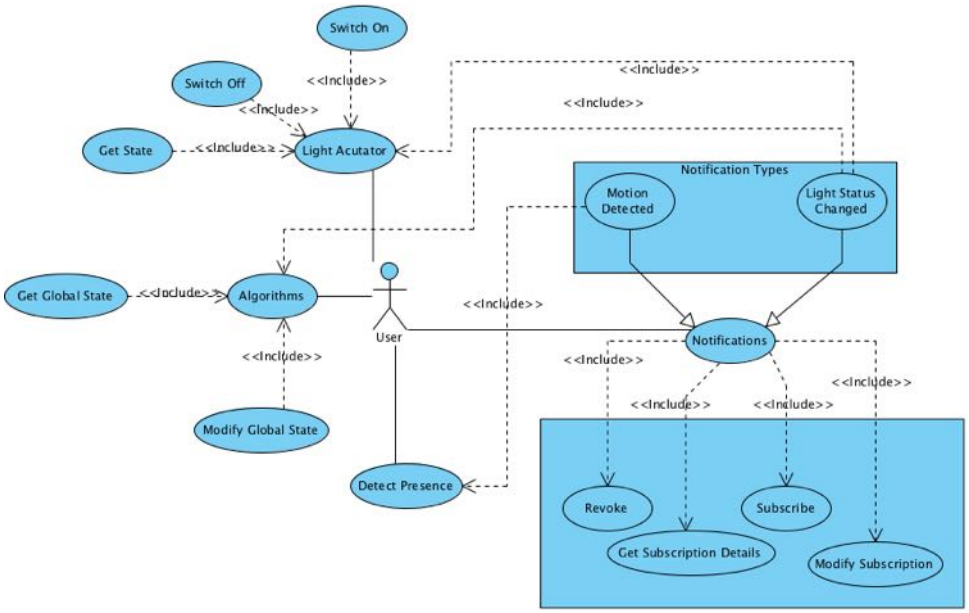


Fig. 8.16.: Extended use-case diagram for the smart light bulb example

Therefore, with a view to obtaining deployable reusable components, they must somehow be grouped together. Figure 8.16 proposes a different view of these use-cases. Here, the use-cases *Switch On*, *Switch Off* and *Get State* are extensions of the *Light Actuator* use-case. Similarly, the *Get Global State* and *Modify Global State* are modeled as extensions of the *Algorithms* use-case. The *Detect Presence* use-case remains the same as in Figure 8.15. Additionally, all three main use-cases include one of the *Notification use-cases*. This consideration naturally leaves three main parts: (1) one responsible for switching the light on and off and to report its state and (2) another for taking care of sensing movements in a given area. The last part (3) includes several algorithms foreseen by the use-case. Besides, this partitioning also feels natural in the physical world where the light switch is physically separated from the motions sensor. The important point is not how far these devices are physically separated but more the consideration that both can work individually and don't necessarily depend upon each other.

In the initial scenario, a user could switch on and off a light either by using a switch in the physical world or by invoking its associated RESTful façade. In this scenario, the light will also switch on if a person is detected. This is a purely electronic action;

the sensor detects a person and switches the light on. Although a user can `GET` the state of the presence sensor, it is more efficient to subscribe to the generated sensing events in order to be notified when somebody enters or leaves the sensing area. Here, the client is more interested in knowing when a sensor event happens then the type of event (leaving/entering). This makes it a perfect fit for notifications. Finally, the scenario foresees that in a house multiple smart light bulbs together with motion sensors can be installed. Each light can be switched on and off as explained previously. However, it is also possible to switch on or off all the lights together or to check if any light is switched on.

In this vision of creating independent components, clearly the light switch actuator will be one compilation unit. The hardware responsible for switching the light on and off is coupled with a RESTful interface exposing one resource to get and manipulate its state plus a potential second resource responsible for notification handling. These two resources are bundled together and form a service which is deployed on the light switch actuator. This service together with the actuator forms the first xWoT component. As previously discussed, the motion sensor exposes one resource returning information about the current state (whether somebody is in the monitored area or not) plus a second resource implementing the sensor notification. Again, the resulting RESTful service is directly deployed on the motion sensor. Thus, the motion sensor and its associated RESTful interface form the second xWoT component. Finally, the scenario foresees that it is possible to find out about the overall state and switch all the light bulbs on and off together. This task is carried out by a virtual service, implementing the necessary business logic, hence it is composed of a single resource representing this global system state. Since this service has no physical counterpart, it does not matter where it is deployed.

Although this design shows a clear separation of concerns, the original use-case is not implemented. Clients need to know about each deployed smart light bulb to take advantage of it. However, there is no RESTful interface grouping all these smart light bulbs and offering a convenient way to exploit them. Usually, an additional service, called *Application Scenario Service*, plays this aggregation role. Instead of creating a fourth xWoT component, the algorithm and application scenario service can be put together. The third row of Table 8.3 shows that the resulting service offers at its top level an *Algorithms* resource where clients can check and modify the overall system's status (i.e. switching off all light bulbs). Since this service is the main entry point for clients, it also has to offer a way to access individual smart light bulbs and their associated motion sensors. This is handled by the `{id}` layer where `{id}` is a placeholder for each individual smart light bulb. Beyond this level, the *Application Scenario Service* does not re-implement what has already been offered by the deployed smart light switches and motion sensors; instead the service redirects the user to the desired resource. This allows one component to represent the global scenario plus other components which some of the work is delegated to. As a side effect, the *Light Switch Service* and the *Motion Sensor Service* might be used in another similar scenario. Since they are all based on a common meta-model, compatibility between different services is ensured.

This example shows how a complex scenario can be split down into simple, individual components which are deployed separately and can live on their own. Additionally, the *Application Scenario Sensor* is a good example of how such components can be combined into new components. It is important to note that this approach is different from a classic

Light Switch Service offering:	
http://service1.com/light/	`GET`, `PUT`
http://service1.com/light/publisher/	various
Motion Sensor Service offering:	
http://service2.com/motion/	`GET`
http://service2.com/motion/publisher/	various
Application Scenario Service offering:	
http://service3.com/slb/	`GET`, `PUT`
http://service3.com/slb/{id}/light/	`GET`, `PUT`
http://service3.com/slb/{id}/light/publisher/	various
http://service3.com/slb/{id}/motion/	`GET`
http://service3.com/slb/{id}/motion/publisher/	various

Tab. 8.3.: Components and offered resources

mashup application (which would lead to a similar result). Although, a mashup also uses the *Light Switch Service* and the *Motion Sensor Service* to offer a global view of the system, the mashup application is not a component anymore. Since such a mashup does not follow the xWoT meta-model nor offer any RESTful interface, other use-cases cannot rely on it to build a novel application scenario.

9

A Model Driven Component Generation

9.1. Introduction

The building blocks forming the WoT are smart devices. A smart device can roughly be defined as a physical object to which a service interface is added. Therefore, a smart device has two attributes: (1) it has some inner (physical) structure made of sensors, tags and actuators and (2) it presents to the outer (virtual) world a clearly defined (RESTful) interface. The first encompasses the device aspects (the hardware) whereas the second takes care of the "smart" aspect.

Definition 13 (*Smart Device*)

A smart device is composed of a physical device *and a* virtual part *with a service interface. Whether the device is manipulated physically or virtually, the result remains the same. The virtual side is connected to the physical world through sensors, actuators and tags.*

The previous chapter introduced the notion of xWoT. Definition 11 specifies that it aims to create the building blocks for mashups and other applications relying on smart devices. These building blocks are nothing other than components in the sense defined in Chapter 6. This implies that they have a public part, exposed to clients, and a hidden one. Clearly the RESTful interface represents the public part of a smart device. It is the interface over which it can communicate with either other smart devices and applications (B2B) or with human clients. Instead of going from these components one step up to mashup applications, it is also possible to go one step back and analyze their underlying structure. Figure 9.1 shows a three-layered architecture where mashup applications are represented in the topmost layer. They are based on the RESTful interfaces offered by smart devices and in our case on RESTful services offered by components of the xWoT. A further layer below sits the meta-model. It names the different elements making up a component and describes their relationship. By that, the meta-model simultaneously takes care of a component's inner guts and its outer RESTful interface. Furthermore, Figure 9.1 shows - on the bottom layer - that the meta-model takes into account both types of information providers mentioned in Definition 11,

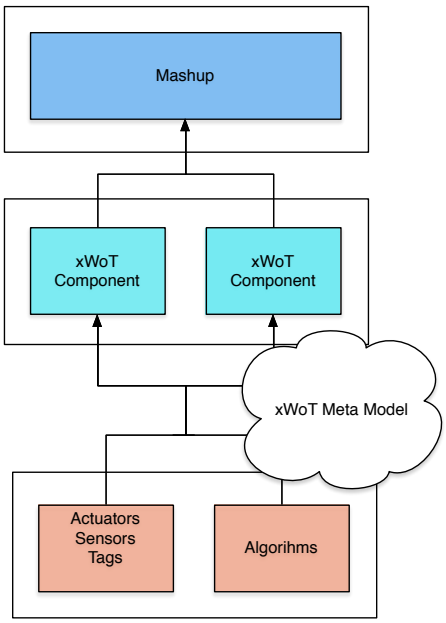


Fig. 9.1.: A three layered approach for xWoT Applications

A meta-model introduces formalism into a system. All models and by extension all endeavors which are instances of this meta-model share some common properties. Therefore, the meta-model introduces conventions inherited by all the derived models and endeavors. This chapter first discusses why building systems that respect conventions is better than fully customized ones. Based on the insights gained in the previous chapter this first section introduces the terminology plus a meta-model tailored to the needs of the xWoT. A second section introduces the methodology, based on the meta-model, which supports developers during the creation of new xWoT components. Based on this methodology, a third section shows how almost ready to be deployed xWoT components can automatically be generated. Finally, the last section discusses the influence of the Web as a supporting infrastructure of these components.

9.2. A Meta-Model for the xWoT

9.2.1. Convention over Configuration

Convention over configuration is a well-known software engineering paradigm and widely used. Traditional frameworks tend to need heavy configuration to adapt them to the current situation. On the one hand this gives the developer complete freedom regarding aspects like project structure, naming conventions, etc. On the other hand, most developers tend to adopt some conventions (at least their own). Even though they may vary from one developer to another at some point, each learns from experience of his own best practice and sticks to it for all future projects. Nevertheless, each new project still needs the same configurations over and over again. If for a given platform, all developers agree on a set of common best practices, the platform can then integrate them as conventions eliminating the need for repeated configurations. Besides a considerably speeding up in the development process, conventions also improve the readability of foreign code. This chapter first introduces the convention over configuration (also called coding by convention) paradigm and show how it helps developing applications. The second step explains the need for such an approach for the extended WoT and roughly introduce some conventions. The next chapter introduces a set of convention for the xWoT. Starting with these, a meta-model can be derived which serves as template for all future xWoT components.

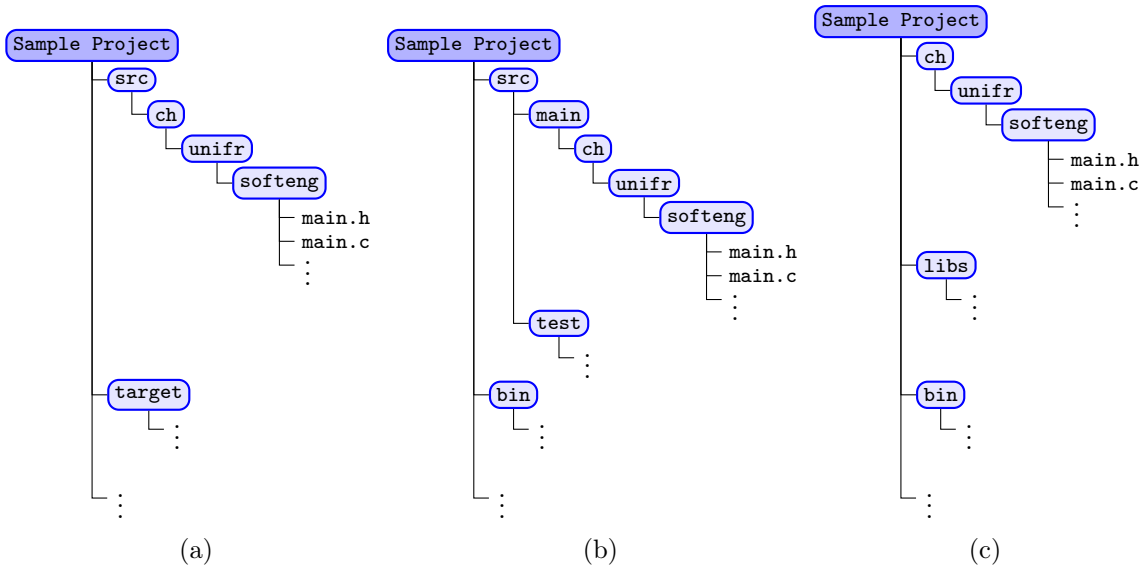


Fig. 9.2.: Different Project Setups

The Apache ANT Project [WEB61] illustrates a situation where configuration is needed. Originally, Ant was developed as a build tool for the Tomcat server. Its inventors quickly discovered the power of this tool and forked Apache Ant as a standalone project in 2000. Since then, Apache Ant has been a widely used build tool for Java code just as Makefiles in the C world are. Although these build tools know what a compilation is, they need a configuration file to provide them with information about the project's structure. Figure 9.2 shows three different setups for projects. At first glance, neither seems better than the others. Listing 9.1 shows the consequences of this situation. Although compiling a Java application is just a matter of launching the `javac` binary against all `*.java` files,

it needs a considerable amount of configuration in order to tell Ant where to find the source files, where to put the binary files, where to look for compile dependencies etc.

```
<project name="HelloWorld" default="compile" basedir=".">
    <!-- build-specific properties -->
    <property file="build.properties" />
    <!-- set global properties for this build -->
    <property name="src.dir" value="src" />
    <property name="build.dir" value="bin" />
    <property name="dist.dir" location="dist" />
    <property name="reports.dir" value="reports" />
    <!-- for code reviewing reports -->
    <property name="doc.dir" value="doc" />
    <property name="javadoc.dir" value="${doc.dir}/javadoc" />
    <property name="java2html_task.dir" location="${resources.dir}/java2html" />
    <property name="sourcedoc.dir" value="${doc.dir}/source" />
    <property name="browsable-source.dir" value="${sourcedoc.dir}/html" />
    <property name="checkstyle_task.dir" location="${resources.dir}/checkstyle" />
    <property name="checkstyle_report.dir" value="${reports.dir}/checkstyle" />
    <property name="javadoc_check.dir" value="${reports.dir}/doccheck" />
    <property name="doclets.dir" location="${resources.dir}/doclets" />
    <!-- ================================================================== -->
    <!-- Class paths -->
    <!-- ================================================================== -->
    <path id="compile.class.path">
    </path>
    <!-- ================================================================== -->
    <!-- Compile: Default -->
    <!-- ================================================================== -->
    <target name="compile" depends="init" description="Compiles all Java sources">
        <javac destdir="${build.dir}" classpathref="compile.class.path"
            ↪ deprecation="on">
            <src path="${src.dir}" />
            <include name="**/*.java" />
        </javac>
    </target>
```

List. 9.1: ANT build File

This situation could easily be avoided if all developers agreed on the same project structure. Although this seems difficult, many frameworks rely on this mechanism. By using a framework, a developer commits to accepting all the related conventions. *Struts*, a popular MVC (Model View Controller) framework is based on this pragma. One of its conventions is that controller classes should have a name ending with *Action*. As long as a developer follows this convention, Struts will automatically discover controller classes without any need to configure Struts. Clearly this convention restricts the developer when choosing class names, however compared to the benefits, this restriction is bearable. Ruby on Rails is another successful framework (a web application framework). Using it means adopting its underlying project structure. Although, the structure seems overwhelming at first it is very powerful, and once a developer is used to this structure and its associated rules, he can focus on his main task. There are a handful of success stories where developers prefer to adopt conventions in exchange for getting better, faster and more robust results.

9.2.2. Related Work

Bringing structure to the IoT is not a new idea. Over the past few years, many attempts have been made, some quite successful. Defining REST architectural style as the only communication interface has made the WoT successful [22]. By adhering to RESTful web services, the WoT structures the IoT in such a way that Things offered by different vendors remain compatible. The adoption of REST as the architectural style is the most remarkable advance in the structuring of the IoT. Although the WoT is a big step forward, REST does not structure the different smart devices. Furthermore, REST is an architectural style, how this style is applied to a given situation is left up to the developer. This leads to the situation where the same physical device can translate into an infinite set of possible virtual counterparts. This makes it hard for other developers to find suitable smart devices and it is also hard for others to understand how the RESTful interface works and which resources are available. The big advantage of the RESTful interface is to help developers creating mashup applications. A quick search on Google reveals the amount of work accomplished on physical mashups. Many papers are published in this domain but also much work has been done. Between 2010 and 2013 the number of mashup platforms like cosm and open.sen.se has grown steadily. Yet, RESTful web services alone do not help developers when building the foundations upon which such physical mashup applications are based.

Another successful approach is the European research project IoT-A. Back in 2010, S. Haller presented a technical report *The Things in the Internet of Things* [29] where the author presented a first sketch of a reference architecture for the IoT which later became the reference architecture for the IoT-A project [78, 6]. Furthermore, the authors introduced the terminology which shall be adopted by all major players in the IoT domain and will facilitate the communication between the different players involved in any software development project. This reference architecture is supported through a European project with many well-known companies involved (Siemens, SAP, IBM etc.). Figure 9.3 shows the final reference model, which only slightly differs from its initial versions in [78, 29]. Involving all the major players of a given domain in one project has the advantage that the result is supported by all of them. However, when that many players agree on a common basis, often the result reflects all opinions at the same time. A rough look at Figure 9.3 reveals that there are a lot of *many-to-many* connections. Additionally, many elements are recursive structures. Although this allows the modeling of just about any imaginable situation in any way, it has the enormous drawback that the situation is not much improved. Furthermore, as the architecture is targeted to support the IoT, a considerable effort has been put into modeling the interactions between the raw hardware and the business logic behind the API. The reference architecture has at least the benefit of introducing a clear terminology for all the involved components. However, as a model, it is just too broad to be of a direct use.

Over time, researchers also introduced some conventions for the WoT. Particularly for all sorts of semantics related to the IoT. This is not surprising, since semantics is often the biggest obstacle users face when creating new mashup applications. When building such mashups, it is usually hard to know what values a given sensor is returning, or even to know whether a given resource is a sensor or an actuator. To overcome this problem and also to enable M2M (machine-to-machine) communication, Kopecky et al. [41] introduced *hREST*, a microformat to describe RESTful web services. By injecting special `id`

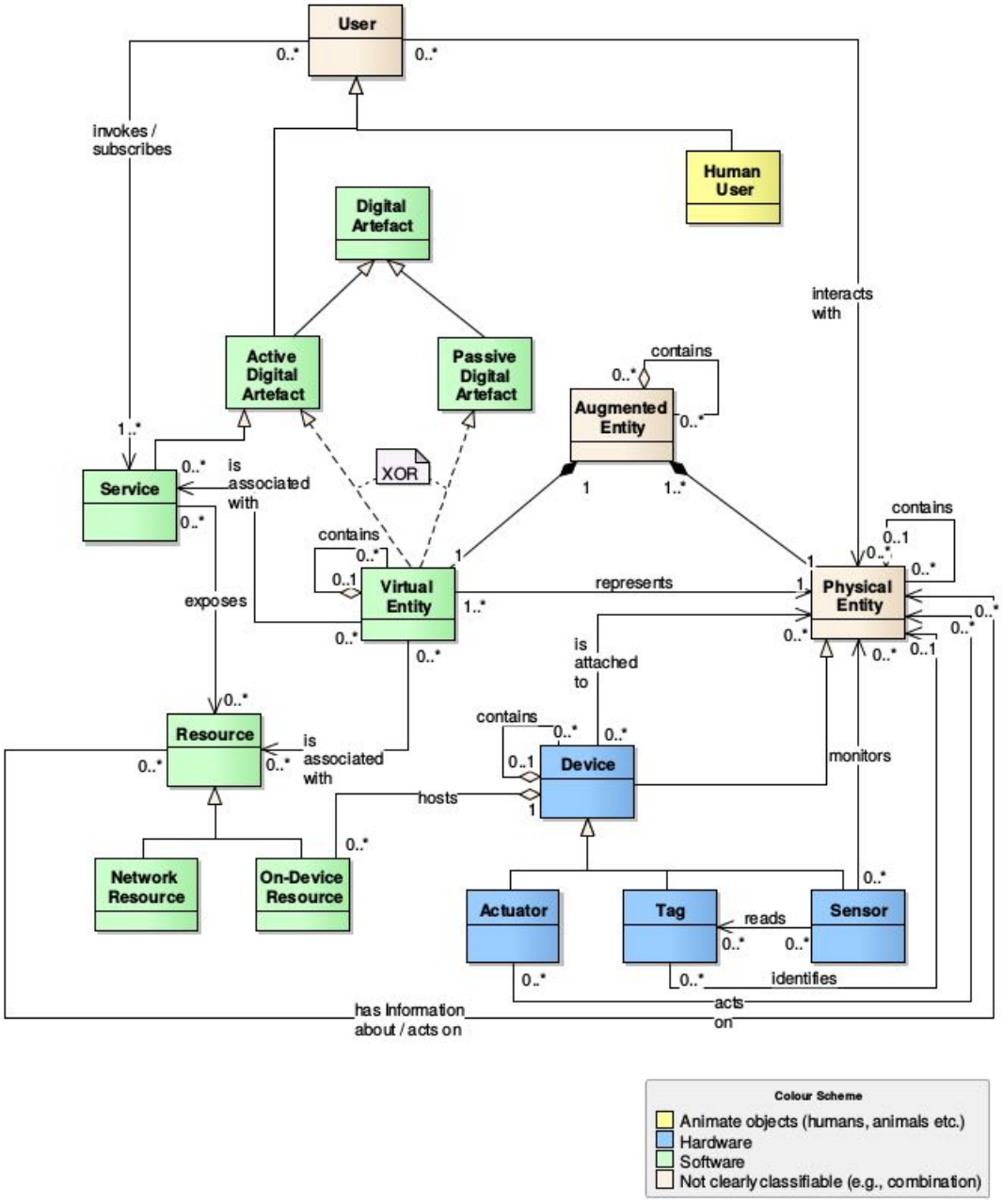


Fig. 9.3.: IoT-A reference architecture (from [6, p. 58])

attributes into the HTML code, the description of any REST interface gets a semantic meaning and is thus machine-readable. Adopting the proposed microformat finally allows RESTful services to be integrated into semantic web services through technologies like WSMO-Lite [77] and SAWSDL [12].

Another attempt to structure RESTful web services and also the WoT are Triple Spaces [13], a concept that goes back to space-based computing and parallel computing. Triple Space extends classical space-based computing by adding semantics. One of its requirements is the need for a web-like communication style. This makes REST and RESTful web services a first-class choice for supporting triple space computing. Gomez in his work [16, 15] gives some hints about how Triple Space computing can be applied to the WoT and its benefits for the WoT.

In their work, De et al. [10] formally introduced some fundamental IoT concepts. The authors describe an *Entity* as a central piece of interest to which services, resources and devices are related. Based on this observation, they introduced a very detailed ontology describing the different aspects of such an entity, containing temporal, space- and domain related features. Based on this preliminary work, the authors then defined a second ontology taking care of the represented information, called resource. This ontology is agnostic to the underlying architecture used to implement such a resource. For the authors, the resource implements an interface, which can be REST, RPC or SOA (Service Oriented Architecture). According to the authors, these semantics can be used to achieve several goals, among them semantic reasoning and dynamic association.

When in [1] Alarcón et al. discuss how RESTful services can be crawled, they also shortly describe a REST Service Description meta-model. The authors proposed ReLL (Resource Linking Language) to describe a RESTful service. The presented model takes special care of the different types of links added to each resource deployed on a RESTful service. The core of the meta-model states that a *service* (REST) exposes several *resources*, each having different *representations*. The necessary links for the ReLL to work are then added to the different *representations*. Although, the meta-model presented is very precise about the *links* and some related concepts, it completely misses the (hierarchical) relation between the available resources.

In her work, Schreier concentrated on modeling RESTful applications [73]. To achieve this goal, she proposed to capture static aspects with a structural meta-model and behavioral ones with a finite state machine. Having some (meta-)model representing the inner structure feels quite natural. Application architects relying on this work need some common structure to be captured in the structural meta-model. Yet, according to Fielding [14] representations are subject to change over time. These changes are captured in a behavioral meta-model. Both, the meta-models and the finite state machine are very complete. Going beyond resources and representation, they capture concepts like the links between resources, attributes and parameters, media-types, methods and much more.

Many conventions related to the WoT have in common that they try to bring semantics to the WoT, a current research topic trend. Projects like *WolframAlpha* [WEB74] show the power and benefits of semantics. Although, this is a popular topic, discussing semantics for the WoT is outside the scope of this thesis. Instead, the remainder of this chapter focuses on introducing a number of conventions to structure the xWoT. On the one hand, most of these conventions are reflected in an intrinsic meta-model for the xWoT. On the other hand, the associated methodology guides developers and system architects

through the process of creating xWoT compatible components. The previously mentioned conventions regarding semantics also benefit from such a structured approach since this limits the degrees of freedom of the RESTful interface of an xWoT component and imposes some minimal structure. Interestingly, we discovered several common points between all these research topics and the presented meta-model. Given that they all work in connected domains, this was not surprising, rather encouraging and validates the findings, to some extent.

9.2.3. Terminology

Part of meta-modeling consists of naming the important elements of a system, so creating a language for the latter. Before discussing the xWoT meta-model, this subsection introduces the terminology serving as a basis for the meta-model in the next subsection. With respect to the terminology defined in Subsection 7.2.3 this subsection introduces the Model Unit Kinds for the xWoT. Later, each xWoT model contains Model Units originating from these Model Unit Kinds.

Already at a very early state of the IoT-A reference architecture, S. Haller discovered [29] that the smart object is the centre-piece of the IoT and also the WoT. However, the terms *smart object*, *smart device*, *Thing* are too vague to describe this element. It is mainly this argument that pushed S. Haller to introduce the concept of *Entity of Interest*. The term is composed of two parts: (1) Entity and (2) Interest. The word Entity means *something that exits by itself* [WEB11, WEB32, WEB40] and describes something that can exist independently without conditions. This aspect is also underlined by the word's etymology derived from the Latin word *esse* meaning "be". Therefore, the *Entity of Interest* is an abstract concept. As such, it is neither dependent on any physical object nor any RESTful interface. According to the Oxford dictionary, the word *Interest* expresses curiosity about or paying attention to something. Therefore, something is *of interest* if it attracts our attention. Putting both terms together leads to a self-existing concept of particular interest here. This is exactly what the *Entity of Interest* expresses, an abstract concept about smart objects, which are the centre of interest. It is important to understand that the *Entity (of Interest)* is not necessarily a smart device. Suppose that we are interested in knowing the temperature of a given room. Then, the *Entity* is the temperature. Accordingly, some hardware is needed to measure the associated physical phenomenon. Here, the hardware is a simple thermistor sensor measuring its resistivity (which depends on the temperature). In this case, the smart device and *Entity* are the same, the temperature. However, in a more complex scenario, this can be different. If instead of knowing the temperature, we are interested in the room itself, which happens to have the ability to report its temperature (among other things), the *Entity* is no longer the temperature but the room itself. Still, at some point, a physical device with a RESTful interface is needed to complete the scenario, hence, the *Entity* corresponds to the point of view adopted by the modeler. *Entity* and *Entity of Interest* both have the same meaning and, although the former is adopted in the meta-model, both are used to designate the same concept throughout this thesis.

Definition 14 (*Entity* ■)

The Entity (of Interest) reflects the point of view adopted by the architect. It represents the core element of the current use-case. In that way, it covers both the physical and

the virtual aspects.

Considering the concept of Entity, note that it is not necessarily a synonym for a smart device. Therefore, a way to talk about smart devices is needed. Smart devices have a physical side composed of sensors, actuators and tags plus a virtual side offering a RESTful web service. Given this duality, introducing smart devices into the terminology does not make much sense. Rather, we introduce a *Physical Entity* to designate the physical manifestation of a smart device. The Physical Entity captures all the physical aspects of an Entity. Again, since an Entity does not necessarily map to one particular smart device, the physical aspects of an Entity are not limited to sensors, tags and actuators. Rather, an Entity encompasses all the physical aspects of interest. In the context of the previous examples, this means that if the Entity is the temperature, then the Physical Entity is the sensor measuring the temperature. If on the other hand, the Entity, is a room (containing among other things, a temperature sensor) then the Physical Entity not only encompasses the temperature sensor but, also the room. As will be seen later, this nuance is important in order to create accurate models. Therefore, the Physical Entity etymologically is an entity. As the two small examples show, it deals with raw sensors, actuators and tags as we already know them from the IoT, but it also handles other physical objects.

Definition 15 (*Physical Entity*)

The Physical Entity captures all relevant physical aspects of a use-case. These aspects not only include devices like sensors and actuators but any physical object of relevance for the current use-case.

So far we have introduced two main terms of our meta-model - *Entity (of) Interest* and *Physical Entity*. Furthermore, upon introducing these terms, we stumbled several times over the words sensors, actuators, tags and devices. If the Physical Entity stands for the physical aspects of an Entity, then it is composed of what makes a common device smart, tags, sensors and actuators. This is also underlined by other research like Haller [29] when he speaks about the *Things in the Internet of Things*. In our meta-model, we define them in an everyday way: a *Sensor* is a physical component measuring or observing a physical property. For example a thermistor is a physical object measuring the surrounding temperature. It is a purely passive device and does not impact on its environment (although, the observer effect shows that this point of view is questionable [WEB37].)

Definition 16 (*Sensor*)

A sensor is a physical object, capable of measuring or observing a physical phenomenon.

If a Sensor is only capable of observing a physical phenomenon, in order to fully embed the physical world into the virtual, another piece of hardware is needed, capable of acting on the physical world. This is exactly what *Actuators* do. Upon receiving some input, the Actuator starts acting and so changing the physical property it is attached to. One very common kind of Actuators are stepper motors. Physically, they are nothing more than a brushless motor, which divides the full rotation into a number of equal steps. The motor is not only capable of turning a given amount of time but also a given amount of steps. Such motors are used to open and close doors or windows.

Definition 17 (*Actuator*)

An actuator is a physical object, capable of acting on another physical object.

Whereas Sensors and Actuators are mostly not necessary for the physical object to function properly, they are the extension of the virtual world into the physical. If a client interacts with an Entity over its virtual interface, the Sensors and Actuators translate these interactions into physical ones and vice versa. *Tags* on the other hand only play a minor role in these interactions. The tag is a passive entity holding a small piece of information. This information can be anything but it commonly contains some sort of information like a URI where clients can find the virtual counterpart of the object. A Tag can for example be attached to a book. A user can then either read the paper version of the book or by scanning the Tag, access an electronic version. Whereas Sensors and Actuators tightly couple the virtual world to the physical, Tags only loosely couple the two.

Definition 18 (*Tag*)

A Tag is a passive physical attribute. It holds some information about the object it is attached to, like a URI and serves to identify the former.

Sensors and Actuators are the connecting piece between the physical and the virtual world. They translate physical phenomenon into virtual representations and translate virtual actions to the physical world. Previously, we have discussed a thermistor as example of a Sensor, and a stepper motor as an example of an Actuator. However, sometimes scenarios require more complex setups to capture and manipulate all the physical aspects an object offers. To build a smart door, at least one (stepper) motor is required to open and close the door. Yet, since it is still a door, it can be opened and closed directly, thus confusing the virtual counterpart. Therefore, to fully model such a smart door, a Sensor measures whether the door is open or closed and to what extend. Such combinations of Sensors and Actuators are common in the xWoT. A *Device* is a combination of Sensors, Actuators and, less commonly, Tags. Such a combination only happens if the Actuators, Sensors and Tags are related to each other. Combining the door's stepper motor with the room's thermistor Sensor would usually not form a Device but rather two different Devices each one containing a single Actuator plus a single Sensor.

Definition 19 (*Device*)

A device is a physical combination of several related Sensors, Actuators and/or Tags.

Since smart devices also have a virtual side, the *Virtual Entity* is the part of an Entity dealing with this facet. More precisely, the *Virtual Entity* handles all the virtual aspects of an Entity. It deals with its RESTful interface and the inherent structure of the latter. As already pointed out for the Entity of Interest as well as for the Physical Entity, the Virtual Entity is also an Entity etymologically speaking. Thus, it exists on its own without the need for any anchor point. Again, this point is quite important for the xWoT. According to Definition 11, the xWoT also deals with service only components. Such service components don't have any anchor points like a physical object; instead they can exist on their own.

Definition 20 (*Virtual Entity*)

The Virtual Entity captures all relevant virtual aspects of a given Entity. This virtual information is exposed via a RESTful interface over which clients can interact with it.

One of the main concepts of any RESTful web services are *Resources*. If a user requests or sends some information to the server, he does this by interacting with a *Resource*, where each one stands for another piece of information. Tied to the term *Resource* are often *Representations* and *Methods*. If a client requests a *Resource* the server will send back a *Representation* of the latter but never the *Resource* itself. Additionally, to uniquely identify each *Resource*, each is accessible over a different URI. Just as URIs are hierarchical and can drill down, so can *Resources*. Although, some RESTifanian mandate that URIs should not have any meaning (see Subsection 5.4.1) there is at least a hierarchical dependency between a *Resource* representing a collection containing for each item a sub *Resource*.

Definition 21 (*Resource*)

A Resource is a piece of information available over a RESTful interface. Resources can be decomposed into sub resources and offer different representations.

9.2.4. The Meta-Model

Based on the terminology introduced above we can now define the meta-model for the xWoT. Restricting its application to the xWoT results in a simple yet powerful meta-model tied with a precise methodology. Consequently, the resulting meta-model is simpler than similar approaches. Furthermore, the introduction of the meta-model allows to automatically generate code skeletons as mandated by MDA (Model Driven Architecture) [B19, 54].

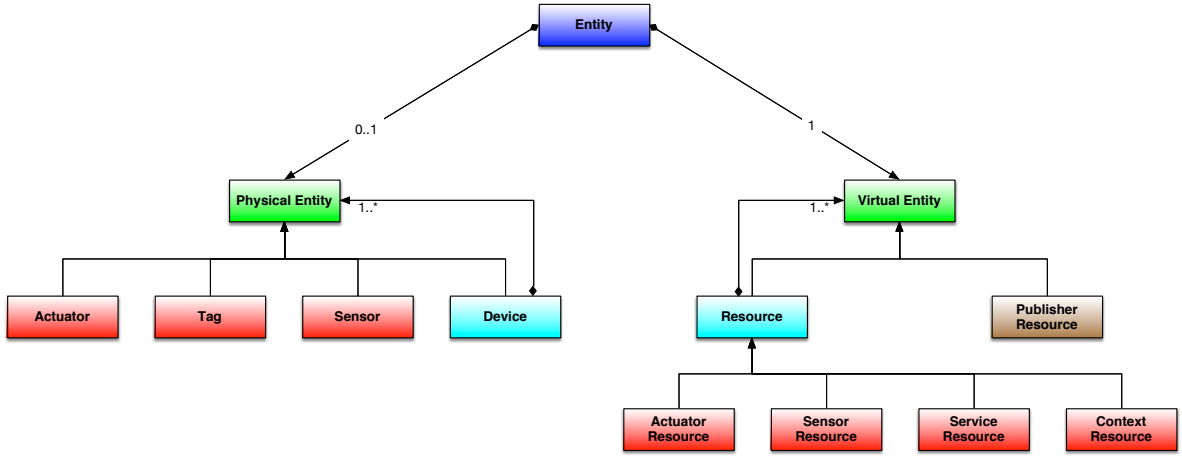


Fig. 9.4.: Overview of the xWoT meta-model

Figure 9.4 presents the initial meta-model containing the terms discussed plus some new ones. As suggested previously, xWoT components have at the same time a physical and a virtual side. This duality directly translates to the meta-model. The *Entity* is the central concept. It represents the point of view adopted by the architect during the modeling phase. Each Entity is composed of exactly one *Virtual Entity* and may contain a *Physical Entity*. Since the xWoT also deals with service only components, it is possible that some use-cases don't have any physical manifestation at all and thus don't need a Physical Entity. This nuance also explains the difference in cardinality between a Physical and a Virtual Entity.

To apply the meta-model to a broad range of scenarios, the Physical Entity is composed recursively. This allows modeling situations where an Entity is composed of more than one Sensor, Actuator or Tag. The smart door example of the previous subsection is such a case where a composed Physical Entity is necessary. Moreover, the different Sensors, Tags and Actuators can be grouped into a tree-like structure. From a software designing point of view, recursive tree structures can be modeled with a composite pattern. Applying this pattern to the Physical Entity leaves us with a situation where the Physical Entity becomes abstract. In terms of design patterns it becomes the component and defines the common interface of all composites and leaves. The Device plays the role of composite. It can either be used to attach several Sensors, Actuators and Tags to an Entity or to group different composites. Whereas the first case is the obvious application, the second is at least as important. If we try to model a smart room containing a smart light bulb, a smart heater plus a smart window, then the smart light bulb, the smart heater and the smart window each translates to a Device instance, with each grouping the necessary Sensors and Actuators to achieve its individual requirements. The smart room itself however also has a physical manifestation, which is of interest as the smart room as a whole translates to a Device instance containing the other Device instances. Figure 9.5 shows an instance of the meta-model containing the physical side only of this smart room. Whereas the smart heater and the smart light bulb are each grouped into a Device, the smart window, since it is only composed of an Actuator, stands for itself. Only Sensors, Actuator, Tags and Devices can be instantiated in any model. Defining the Physical Entity as abstract ancestor of the latter makes it possible to ensure they all look the same from the Entity point of view and, on the other hand, guarantees that only real physical objects can be modeled. If, by contrast the Physical Entity were concrete, it would be possible to create models where the physical side had no Sensors, Tags or Actuators, thus removing any coupling between the physical side and its virtual counterpart.

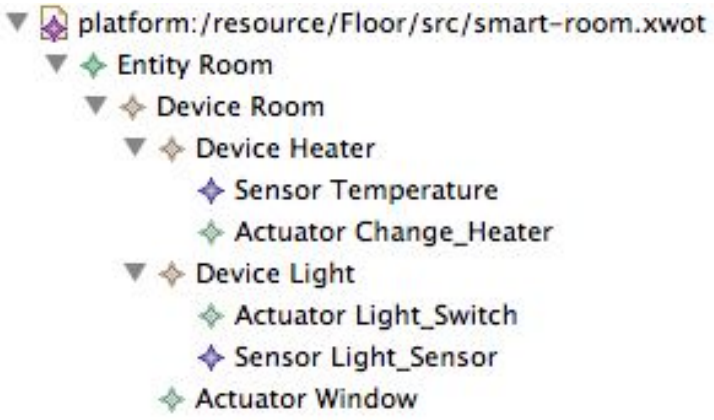


Fig. 9.5.: Model of the physical side of the room example

In order to be smart, the modeled entity also needs a virtual side. The Virtual Entity captures these aspects. If the Entity is composed of a Physical Entity, then there is a one-to-one mapping between the Physical and the Virtual Entity. Nonetheless, since the virtual side can offer resources that do not have any physical equivalent, this mapping is not onto and therefore not a bijection. This can be the case if the smart device, besides offering the raw sensor readings also implements some sort of small algorithm to work with the acquired data. The meta-model clearly reflects this in the different cardinalities

for the Physical - and the Virtual Entity. Just like the Physical Entity, the Virtual Entity is structured following the composite pattern. There are several reasons to adopt this design pattern to break down the complexity of the Virtual Entity. One of these reasons is the one-to-one mapping between the Physical and the Virtual Entity as everything that can be modeled in the physical world can also be modeled in the virtual world. Additionally, URIs form, by their nature, a hierarchical, tree-like structure and, since each resource is attached to a URI, resources also form a hierarchical structure. Richardson and Ruby [B18] also support this vision. In terms of design pattern, this means that the Virtual Entity is the component and thus abstract and not instantiable. However, it ensures that all concrete objects share this same interface and thus, appear to the Entity to be like a Virtual Entity. According to Definition 21, a Resource can decompose into several sub Resources. Unlike in the physical world, there is no anchor point where a Resource cannot decompose any further. Therefore, Resources are not the leaves of the pattern, but rather the composite. In its most basic form, the meta-model would not need any leaves at all, leaving the decision about when to stop decomposing a Resource to the architect. Nonetheless, there are other properties which allow the concept of Virtual Entity to be further designated. The one-to-one mapping between the Physical and the Virtual Entity induces for each component present on the Physical side an equivalent component on the Virtual side. Putting together these considerations leads to a model of the Virtual Entity as presented in Figure 9.6. Sensors, Actuators and Tags find their exact equivalent on the virtual side. Each Actuator maps to an Actuator Resource, each Sensor to a Sensor Resource and compositions are achieved through the Resource. One would expect Tags to map to a Tag Resource. However, as discussed earlier, a tag in the physical world is simply an identification mechanism pointing somewhere. However, the Virtual Entity can offer resources without any physical counterpart. For these reasons, the third leaf of the composite pattern of Figure 9.6 is called the Service Resource and serves at the same time to model the virtual counterpart of tags and additional virtual only goods.

Definition 22 (*Actuator Resource*)

An Actuator Resource stands for a physical Actuator. Similarly, it acts upon something and thus needs some input. In terms of RESTful web services, an Actuator Resource responds to `PUT` *requests where the body contains the updated representation.*

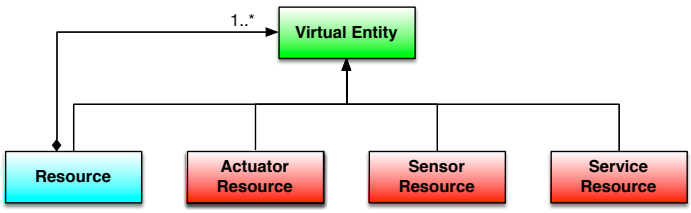


Fig. 9.6.: Simplistic model of the Virtual Entity

Definition 23 (*Sensor Resource*)

A Sensor Resource represents a physical Sensor; it is a rather passive resource capable of returning some information. Therefore, a Sensor Resource responds to `GET` *requests and each time returns an "up to date" representation of the resource.*

Although this model covers all aspects of the physical world and allows a seamless translation of the latter into the virtual world, there remain some issues. Sensors are data generators and thus suitable to require a pushing mechanism in the virtual world. Although Actuators merely work the other way around, they can also need a pushing mechanism even though this situation is less frequent. Lastly, the generic Service Resource of course should also be able to push information. For structural reasons, it makes sense to place the publisher resource near the device resource it is attached to. However, in the model in Figure 9.6 this is impossible. All non-composite resources being leaves, they are not allowed to posses children. On the other hand, the publisher is always the bottom most element a resource can offer. Accordingly, the model needs to be adapted to reflect this possibility. Figure 9.7 reflects these changes by redefining the leaves in Figure 9.6 as extensions of the Resource element. Thus, Actuator Resources, Sensor Resources and Service Resources can be composed.

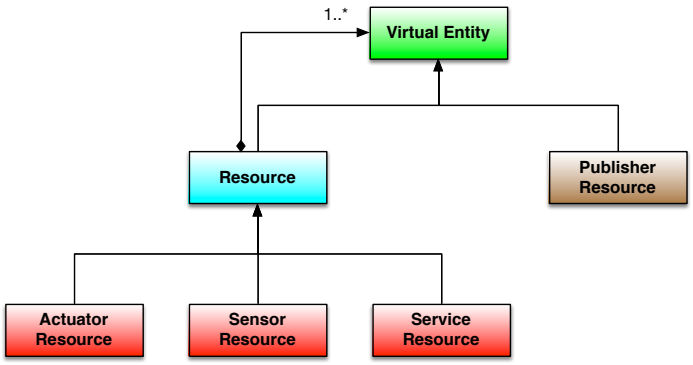


Fig. 9.7.: Less simplistic model of the Virtual Entity

Definition 24 (*Service Resource*)

A Service Resource is a general-purpose resource. It has no restrictions on what requests it responds to, nor whether it contains a publisher or not. Also the Service Resource often decomposes further into smaller Service Resources.

There are only two types of allowed compositions: (1) A Sensor Resource, an Actuator Resource or a Service Resource containing a Publisher Resource. (2) A Resource containing any combination of Sensor-, Actuator- and Service Resources. Furthermore, a Resource can also contain other Resources as children. This allows the forming of logical groups of sensors, tags and actuators and reflects what in the physical world would be seen as a device. The leaves of the previous model are replaced by only one leaf, the Publisher Resource. Whereas, for logical reasons, it does not make sense to add a Sensor Resource as child to an Actuator Resource, Publisher Resources are true leaves. A sensor in the physical world is reflected by a Sensor Resource in the virtual world and delivers the same information as in the physical world. However, the Sensor Resource can offer more information than its physical counterpart. This effect is achieved by adding a Service Resource as a child to a Sensor Resource (of course, the same holds true for an Actuator Resource).

Definition 25 (*Publisher Resource*)

A Publisher Resource contains a WebSocket endpoint plus a second resource allowing subscribers to provide a webhook. Therefore, the Publisher Resource is more a

publisher hierarchy than a single resource.

This last model already fits the defined needs very well. It respects the one-to-one mapping from the physical to the virtual side. It allows aggregating sensors and actuators into groups, it is possible to define additional virtual only goods and finally, the model integrates a mechanism to push information. Thus, the different components of the current model allow it to accurately translate the physical side into a virtual one. Nevertheless, sometimes a less accurate translation is needed. Smart devices in the physical world often not only posses an actuator to modify a physical property but also a sensor knowing the current state of the actuator. For the sake of simplicity, consider again the smart heater of the previous smart room example. The smart heater is composed of an actuator modifying the intensity of the heater. Since such an actuator is more than just an on/off switch it is potentially also interesting to know to what degree the heating device is on. On the other hand, if the state of the actuator is modified in the physical world, this should also be reflected on the virtual side. Hence, the value returned by the sensor is directly controlled by the property modified by the actuator. Other smart devices composed of sensors and actuators do not necessarily have such a strong connection between them. The smart light bulb is also composed of an actuator switching the light on and off plus a sensor returning the measured intensity of the light. However, these two devices are not as highly coupled as those from the smart heater. Besides the light bulb other factors like date and time also influence the sensor. Additionally, the actuator only has two states; it is either switched on or off. This makes it also less interesting to report back the actuator's state.

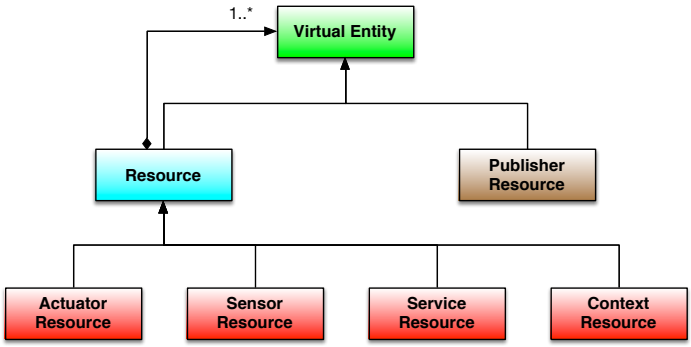


Fig. 9.8.: Final model of the Virtual Entity

Of course, the current meta-model already allows for modeling such situations. Since such a device is represented on the physical side as a Device containing one Sensor and one Actuator, it would translate on the virtual side to one Resource containing a Sensor Resource plus an Actuator Resource. Not only is this translation accurate regarding the imposed one-to-one mapping, but it also feels natural at first glance. However, in the context of a RESTful service, this approach quickly becomes cumbersome. In fact, such a translation requires one REST resource, for example, a *heater_actuator* accepting `PUT` requests, to change the current state of the smart heater plus a second resource, *heater_state* accepting `GET` requests, to return the actual state of the smart heater. The *heater_actuator* does not accept `GET` requests, neither does the *heater_state* accept `PUT` requests. Therefore, in these cases, it makes sense to combine the physical sensor and the physical actuator into one single REST resource, *heater*, this time accepting `GET` and

PUT requests. Although, this simplification violates the request for a one-to-one mapping, it makes the generated RESTful interface much more usable. Moreover, the one-to-one mapping is still present but it has become a semantical one-to-one mapping between the physical and the virtual side. As will seen later, the different tools associated with the meta-model also integrate other simplification approaches. Their common goal is to create the most suitable RESTful interface maintaining a strong link to the physical world it represents. This consideration lead to the final model presented in Figure 9.8.

Definition 26 (*Context Resource* ■)

A Context Resource is syntactic sugar to combine a physical sensor and a physical actuator into one virtual device. For this reason, a Context Resource behaves like a Sensor Resource but, at the same time, also behaves like an Actuator Resource.

The *Virtual Entity* is the abstract component serving as a common interface for all other components. The *Resource* class plays the role of the composite serving as patron for all non-leaf classes. Since *Resources* are used to group other resources, it is concrete and can be instantiated. The four classes *Actuator Resource*, *Sensor Resource Service Resource* and *Context Resource* are also composite classes, although their composition follows some restrictions. Finally, the *Publisher Resource* is the only leaf of the current design. This class contains a whole publisher hierarchy, which is discussed later, and therefore it cannot be broken down any further. This last approach is also the one that made it into the final xWoT meta-model shown in Figure 9.4.

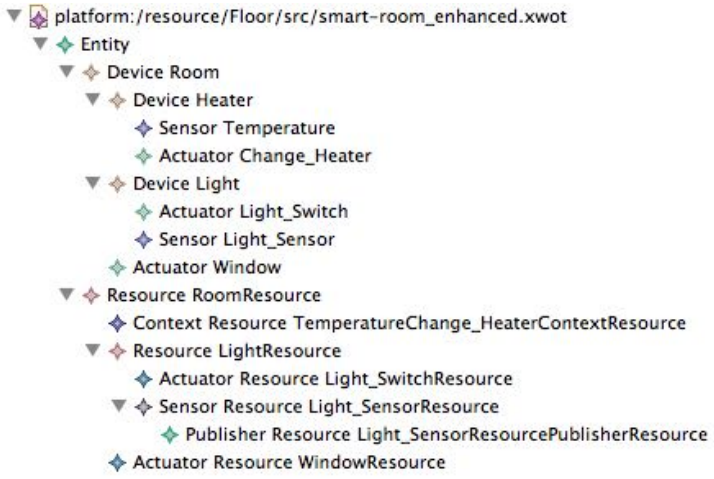


Fig. 9.9.: Model of a smart room

Now that we know how the meta-model handles the virtual side, we can further explore example of a smart room containing a smart heater, a smart light bulb and a smart window. The physical side has already been modeled in Figure 9.5. The corresponding virtual side can be modeled taking into account the details discussed. This results in the final model in Figure 9.9. Accordingly, each smart device has an equivalent on the virtual side. The smart window is the most basic smart device; it is only made of one actuator capable of opening and closing it consequently, it translates to an Actuator Resource on the virtual side called *WindowResource*. The smart light bulb is a little bit more complex as it is composed of a sensor and an actuator. Although, combined into one physical device, their connection is not strong enough to translate into a Context Resource. Therefore, the smart light bulb is translated into a Resource

called *LightResource* containing a Sensor Resource (*Light_SensorResource*) and an Actuator Resource (*Light_SwitchResource*). This structure also delivers the associated Uniform Resource Locator (URL) schema: `http://example.com/room/light/switch` for the Actuator Resource and `http://example.com/room/light/ambientlight` for the Sensor Resource. Additionally, the *Light_SensorResource* offers a Publisher Resource (*Light_SensorResourcePublisherResource*) to notify clients about any drastic changes in the ambient light. Such changes generally involve somebody switching the light on or off. Finally, the smart heater is also a composed device. On the physical side it is composed of an actuator to increase or decrease the output of the heater and a sensor measuring the current output of the heater. Since the two devices are highly coupled (they act on and observe the same physical property) they are translated into a Context Resource (*TemperatureChange_HeaterContextResource*). Since a Context Resource represents both a sensor and an actuator, it could also contain a Publisher Resource. However, to keep the example simple, no publisher is associated with the smart heater.

It is now clear why the meta-model presented is simpler than similar approaches like the reference architecture issued from the IoT-A project (see Figure 9.3). Using RESTful web services allows the modeling of the service components to be simplified. It is enough to partition the space into different resources. REST takes care of the concrete interfaces. Furthermore, REST allows to easy reference to concepts like *Resource* and *Methods*. This eliminates the need for an arduous discussion about what forms the service interface can take. On the other hand, the meta-model is also simpler than others targeting RESTful services like [73]. The difference here comes from the fact that the focus of the meta-model is the xWoT and not RESTful web services in general. However, it would be possible to extend the meta-model with the concepts and relations presented in [73]. However, the associated methodology and tools take automatic care of some important REST aspects. After the introduction of the methodology in Subsection 9.3, we will return to the xWoT meta-model and further refine it. Other published models put too much focus on one aspect of RESTful web services, for instance in [1] Alarcón et al. present a *Resource Linking Language*. This focus is also found in the associated resource model, which emphasizes the HATEOAS principle and thus, gives a lot of space to the definition of links between resources but lacks other details. Web Application Description Language (WADL) [28] is another controversial approach to describe RESTful web services. RESTifanians advocate that there is no need for an equivalent of the WSDL format. Discussing whether WADL is used in RESTful web services or not is outside the scope of this thesis. Moreover, WADL is not a suitable approach to model xWoT applications as it only covers RESTful aspects, but is unable to handle xWoT specific facets.

9.3. Methodology

According to Chapter 7 and more precisely to Definition 8, a meta-model introduces a language for a given domain. Additionally, the meta-model structures this language by defining the different relationships between the terms composing the language. Subsections 9.2.3 and 9.2.4 introduce the terminology and the meta-model respectively for the xWoT. Together with already established standards like REST, the meta-model also allows the introduction of a methodology to design and create new xWoT applications.

The aim of this methodology is to support the system architects during the initial design phase of a project, and also the developers with various tools based on the standards introduced with the meta-model. This section introduces the methodology supported by the xWoT meta-model. Highly coupled with this methodology is also a set of tools, automating parts of the development process. Finally, this section concludes by showing the application of all these tools through a small example.

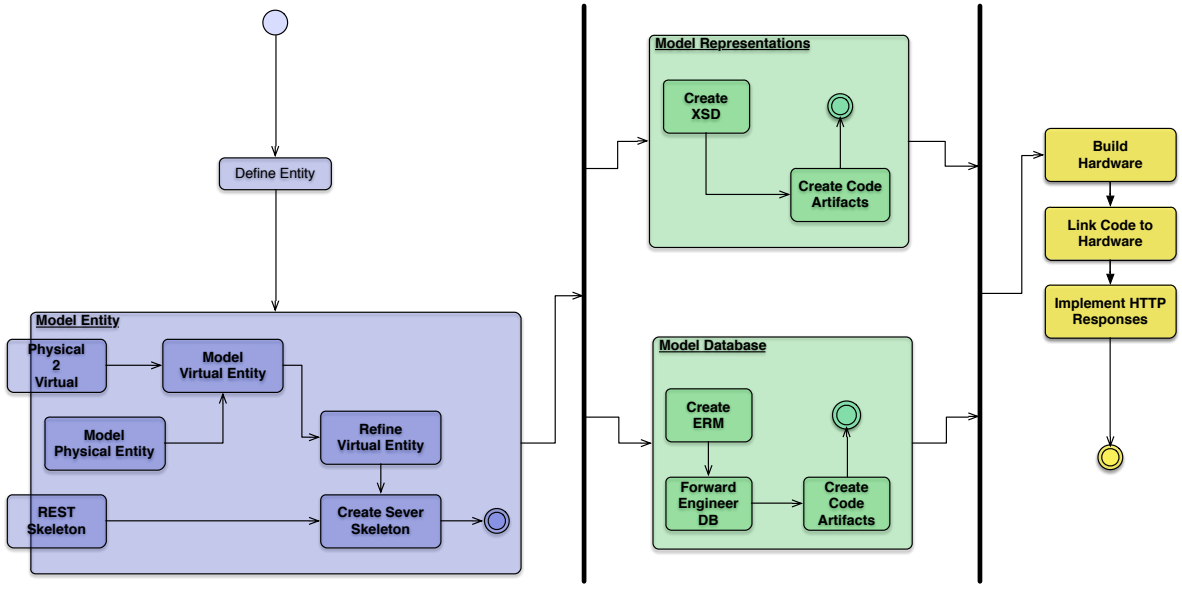


Fig. 9.10.: Overview of the xWoT methodology

The methodology involves several steps, starting with an idea and finishing with a working smart device. During the modeling, the xWoT meta-model has a strong influence on all these steps. Figure 9.10 shows the whole modeling process for creating new xWoT applications. Roughly it can be divided into three phases: (1) Modeling the Entity and its associated physical and virtual side. (2) Modeling the representations and the underlying data model, if needed and (3) Implementing the still empty methods responsible for treating the incoming REST requests. Figure 9.10 also shows these three phases. The first lays the foundation for everything else and is highly coupled with the xWoT meta-model. In addition, all the important decisions and choices are made during this phase. The outcome of this first phase is a project skeleton for the planned REST service. As a skeleton, it already contains all the defined resources and templates for methods to treat the incoming REST requests as defined in the meta-model. The second phase is concerned with data and its representations. Commonly, in RESTful web services, there are two types of data: (1) data living on the server, stored in some DBMS (Database Management System) and (2) data transferred between clients and servers. Both types have their own associated modeling phase. Yet, as they are independent of each other, these activities can be executed in parallel, so it does not matter whether one starts with modeling the server side or the exchanged data. Finally, the third phase puts the different pieces together. It uses the outcome of the preceding phases to implement a fully functional RESTful web service.

9.3.1. Entity Modeling

Modeling the Entity and its associated concepts is the first step when creating new xWoT services. With the aid of the xWoT meta-model, software architects can directly start modeling new scenarios. First, the adopted point of view is chosen. If the scenario talks about a room containing several sensors and actuators, then the Entity would be the *room*. This concept further serves as the anchor point for any other sensors, actuators, tags and devices. After defining the Entity, the Physical Entity can be modeled. Again, the xWoT meta-model support the system architect in this task.

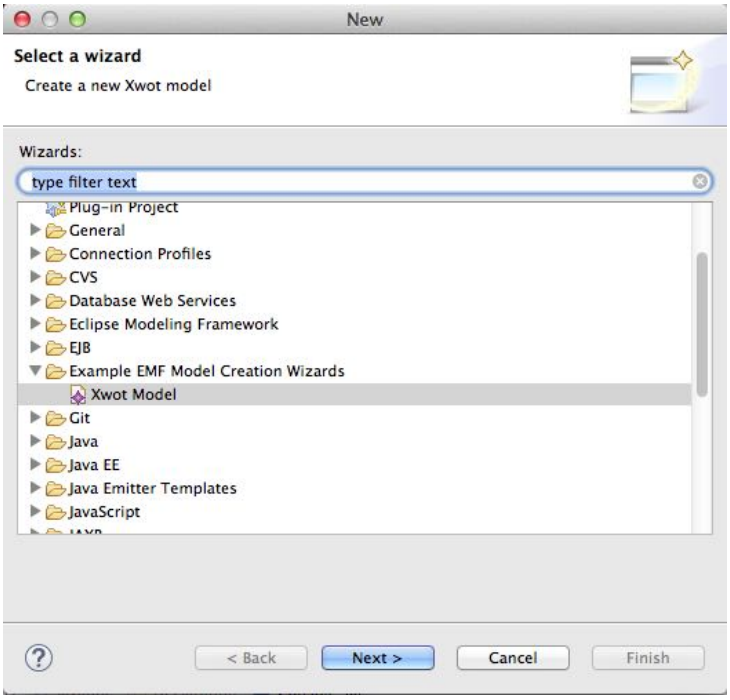


Fig. 9.11.: Creating a new xWoT model in Eclipse

The xWoT meta-model is defined in EMF and translated within Eclipse into an Ecore meta-model (see Chapter 7). Eclipse is then able to transform the meta-model into a deployable plugin. Installing this plugin adds a new type of project to the Eclipse menu: *xWoT Model* (see Figure 9.11). Upon choosing this project, the integrated editor opens a new file already containing an Entity and assists the user through the remaining process of creating a new xWoT compatible model. Upon adding a new element to the model, a panel opens on the righthand side, showing key-values pairs. Figure 9.12 shows an empty model with just the Entity as a node. The panel on the right contains the available properties for each node. For instance, the Entity node only contains one property called `Name`. In this case, it holds the value *Room* to indicate that the Entity is about a smart room.

After defining the Entity, the next step shapes the Physical Entity. If the Entity is a very simple smart device, the physical side can be composed of just one sensor, actuator or tag. If it is slightly more complicated, the physical side will start with a Device to group the different components. Figure 9.13a shows the model after adding a device to the physical side. Again, the lefthand side shows the hierarchy of the component and the righthand side contains the property view. Since the smart room contains more than juste an actuator, sensor or tag, the topmost element of the Physical Entity is a Device. On

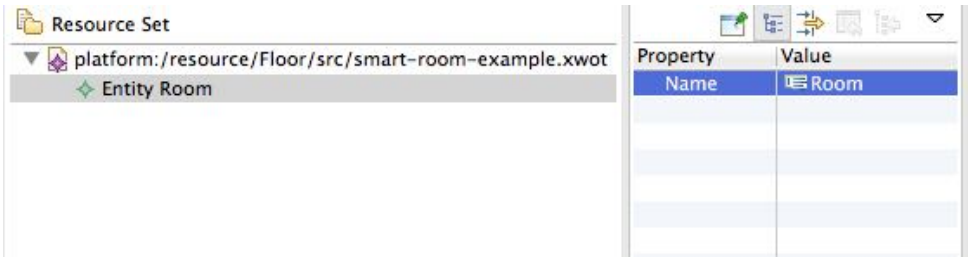
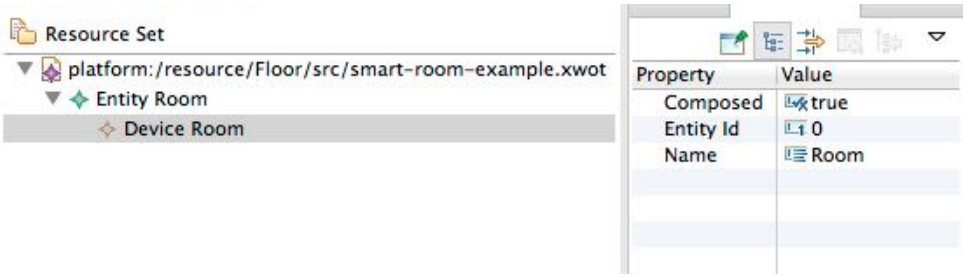


Fig. 9.12.: Defining the Entity, in this case a smart room

the property view, its attributes are defined. The Device has a name, in this case *room*. Furthermore, the Boolean property `Composed` indicates whether a Device is composed or not. This switch influences the generated RESTful web services (see Section 9.4) later in the process. Finally, the `Entity Id` attribute serves to uniquely identify each item of the model. Currently this property is not used, but future versions of the associated tools will not only allow forward engineering of the model to code but also backward engineering. For this to work, code and model need to be strongly coupled.



(a) Defining the Physical Entity

(b) Adding a first device to the physical side of the Entity

Fig. 9.13.: Working with the physical side

In the next step, the remainder of the Physical Entity is modeled. Figure 9.13b shows how the model editor supports the architect in this task by limiting the available nodes to those that are possible according to the meta-model. Regarding the meta-model as presented in Figure 9.4, the only possible children for a Device node are either another Device node or a Tag, Sensor or Actuator node. The editor is smart enough to detect this situation and eliminates all other elements. The outcome of this step should look similar to the physical model of Figure 9.5 which illustrated what a smart room could look like. From a more technical point of view, the outcome of this modeling is a partitioning of the physical space. Sensors, Actuators and Tags are grouped together into physical Devices and/or into logical Devices (for a detailed explanation of the difference between a logical and a physical Device see Section 9.4 and the `Composed` property of Figure 9.13a).

Subsection 9.2.4 defined the relationship between the physical and the virtual side as a one-to-one mapping. Since the physical side has already been modeled, it is now time to do the same for the virtual side, starting by translating the physical elements into virtual ones. This translation can easily be achieved by applying the following set of rules:

1. Each Device maps to a Resource. The `Composed` property must be the same as well as the `Name` and `Entity Id` properties. Since the concept of resources is highly coupled with URLs The Virtual Entity has yet another property: `URI`. Although there are no strict guidelines when it comes to URL naming, for the reasons discussed earlier (see Subsection 5.4.1) it should be meaningful and have a relationship with the Device it stands for.
2. A Sensor maps to a SensorResource. Furthermore, for its properties the same transformation rules applies as for the Device to Resource transformation.
3. An Actuator maps to an ActuatorResource. Again, for its properties the same transformation rules applies as for the Device to Resource transformation.
4. Finally, Tags have no direct equivalent. Since tags merely serve to identify something in the physical world, they map to a general purpose ServiceResource.
5. Since sensors are generating data, it makes most sense to attach a publish-subscribe mechanism to any resource representing a sensor. Also actuators can produce events. Therefore, for each SensorResource or ActuatorResource, evaluate whether a PublisherResource is needed.

After these initial translations, a few simplifications can be applied, making the RESTful interface more intuitive to use. (1) For each sibling made of an actuator and a sensor, check whether they can be combined into a ContextResource. (2) Since a ContextResource is the combination of a SensorResource and an ActuatorResource it inherits the PublisherResource previously attached to the SensorResource. (3) If a Resource contains only one child, then simplify the hierarchy by removing the Resource and attaching the child to its grandfather. This simplification breaks the one-to-one mapping but is beneficial for the RESTful API as it saves one useless layer in the REST hierarchy. The order of these simplifications is important. If before the first simplification, a Resource possesses two children it might be that afterwards only one ContextResource remains. Thus, this node becomes a candidate for simplification step two. If the second simplification were carried out first, it would possibly miss some ContextResource children. For simplicity, consider again the previously defined smart room example. A smart room is made of a smart HVAC device, a smart light bulb plus a smart window. After applying the first three modeling steps, the model should look like the one in Figure 9.14. To keep things simple, the `Name` properties of the virtual elements are composed of the `Name` properties of their physical counterpart plus a suffix (*Resource* and *ContextResource*). Where PublisherResources do not have an equivalent in the physical world, simply use the `Name` attribute of their parent plus a *PublisherResource* suffix.

The translation of the physical to the virtual side completed, the next step is to refine the Virtual Entity. So far, the virtual side has been an accurate map of the physical world, but the Virtual Entity can offer more functionality. If this is the case, then this step adds the necessary resources. Since they don't have any physical counterpart, only Resource and ServiceResource nodes should be added to the actual model from this point on. The smart room example does not offer any additional virtual services. Therefore a refining of the virtual side is not necessary.

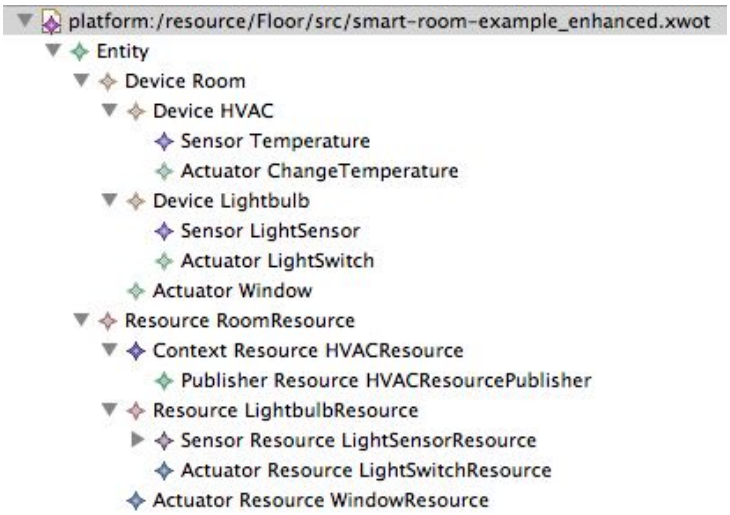


Fig. 9.14.: A smart room with booth, the physical and the virtual side modeled

At this stage the resource design is finished and should not change anymore. Accordingly, the model can now be translated into code or at least a code skeleton. The translation is straightforward; each SensorResource, ActuatorResource, ContextResource and ServiceResource is translated into a REST resource. Resources are either translated into a REST service and a REST resource if their `Composed` property is false. Otherwise they are translated into a REST resource. It is important to understand that the outcome of this translation is not one single RESTful service. Rather, it respects the WoT vision of multiple small and independent devices. The translation to code generates at least one "big" RESTful web service representing the Entity. Additionally, for each Resource where the `Composed` property is false, it creates another RESTful service to be deployed directly on the smart device. The "big" RESTful web service representing the Entity then simply acts as a proxy for the smaller ones, composing the Virtual Entity. Reusability is the biggest advantage of this architecture. Actually, if the translation creates skeletons for already existing web services these can be ignored. Instead the existing ones can be reused. Only the remaining code skeletons need to be implemented.

9.3.2. Data Modeling

After successfully modeling the Entity and its associated physical and virtual side, it is now time to think about stored and exchanged data. Representations are a core concept of RESTful web services. Clients interact with a resource over its representations, be they HTML, JSON, XML or any other format. Since these are the only point of contact between a REST service and its clients, their design is crucial for a successful web service. The commonly used formats, JSON and XML can be modeled with XSD (XML Schema Definition). This is a widely used approach to model XML data and is standardized by the W3C [WEB69]. Although, it would be possible to think directly in JSON or XML, from a software engineering point of view this is not a recommended practice. This is easily illustrated by comparing the representation modeling with the REST API modeling. In subsection 9.3.1, instead of directly writing executable code of the RESTful web service, we first modeled the API by creating an instance of the xWoT meta-model. The same approach should be applied to the representation modeling where the XSD language plays

the role of the meta-model. For instance, Listings 9.2 and 9.3 show respectively the JSON and XML representations of the temperature sensor attached to the HVAC smart device, discussed in the example use-case in this section.

```
{"temperature": {"@units": "celsisus","@precision": "2","#text": "24.00"}}
```

List. 9.2: JSON representation of the temperature reading of the HVAC smart device

```
<temperature units="celsius" precision="2">
   24
</temperature>
```

List. 9.3: XML representation of the temperature reading of the HVAC smart device

JSON and XML are the two most popular representations of a resource, although they all share the same underlying data. In the current example, this is a temperature reading, a floating-point value. A temperature can be expressed in many ways the Celsius, Kelvin or Fahrenheit scales being the most commonly used ones, but there are others like the Rankine scale. However, it is safe to say that the smallest float representation more than covers the capabilities of commonly used thermistor sensors in consumer devices. This discussion shows that it is also important to communicate the scale together with the value to allow users to interpret the returned value. Finally, sensors often return floating point values with far too many decimal values. In these cases, it is important to make an educated choice about the required level of precision of the sensor and also the needs of the clients. Therefore, a temperature sensor is represented by: (1) a floating point value corresponding to the sensor reading, (2) an indication of the used scale and (3) a hint about precision. Now that it is clear what needs to be represented, it is time to decide how this information is to be represented. This is particularly the case for XML representations where a piece of information can be packed either as an element or as attribute to an element. Which one to use is a controversial discussion. Google returns over 1M results, Bing over 6M and Yahoo over 11M responses to this question. Though, discussing XML document engineering is outside the scope of this thesis. Listing 9.4 shows how the above requirements can be modeled with XSD.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified"
    targetNamespace="thermistor" vc:maxVersion="1.1" vc:minVersion="1.0"
    xmlns:vc="http://www.w3.org/2007/XMLSchema-versioning" xmlns:th="thermistor">

    <xs:element name="temperature">
        <xs:complexType>
            <xs:simpleContent>
                <xs:extension base="xs:long">
                    <xs:attribute name="units" type="th:unittype" use="required"/>
                    <xs:attribute name="precision" type="xs:integer" use="required"/>
                </xs:extension>
            </xs:simpleContent>

        </xs:complexType>
    </xs:element>
    <xs:simpleType name="unittype" final="restriction">
        <xs:restriction base="xs:string">
            <xs:enumeration value="celsius"/>
            <xs:enumeration value="fahrenheit"/>
```

```
            <xs:enumeration value="kelvin"/>
            <xs:enumeration value="rankine"/>
        </xs:restriction>
    </xs:simpleType>
</xs:schema>
```

List. 9.4: XSD modeling a representation for a temperature reading

There is yet another advantage for first modeling the representations with a language like XSD instead of writing XML or JSON. Using an intermediary format allows translating it into many different types of code. JAXB [1] is a Java framework widely used to deserialize and serialize JSON and XML data to Java POJOs (Plain Old Java Objects). Therefore, it is frequently adopted in RESTful Java web services. This is only one functionality of JAXB, the `xjc` tool which comes bundled with JAXB allows the creation of POJOs from XSD files. Listing 9.5 shows how this tool transforms the earlier `thermistor.xsd` to Java classes. Similar tools exist for other programming languages. Python, a quite popular WoT language, offers generateDS [2] and PyXB [3] with similar functionalities.

```
xjc -d src -p ch.unifr.softeng.thermistor thermistor.xsd
```

List. 9.5: Creating Java POJOs from XSD files

Since the format of the representations is now well-defined, what remains is the definition of stored data. This step is not always necessary. For RESTful web services representing bare hardware (i.e only made of Resources, Actuator-, Sensor- and ContextResources) there is no need to store any data in a persistent manner on the server side. However, for other types of resources it is. Of course, general purpose ServiceResource are likely to need some data to work with, but a PublisherResource already needs some database. Since clients can (de)-register for some kind of event (see Subsection 9.4), this information needs to be stored on the server side.

Which approach is most suitable to model the stored data greatly depends on the type of data. Thus, a primary analysis is important to decide upon the data characteristics and their quantity. As a result, it should be clear whether the data can be stored on the smart device or if this is already outside the capacity of the smart device. Another important characteristic of the data is whether it is suitable for a relational database or it is better to use document storages and NoSQL storages. Discussing all these approaches is clearly outside the scope of this thesis, so the discussion is limited to the case of a PublisherResource which needs to track its subscribers plus some additional information.

According to Subsection 9.4, a client can subscribe for notifications by invoking the PublisherResource. As soon as the sensors register a change in its value, the PublisherResource iterates through the list of subscribers and sends a notification to each one. Furthermore, upon subscribing, the client can define his own events, in the form of a predicate, for which a notification should be sent. If he does so, this client will only get notified if the predicates evaluates to true. Figure 9.15 shows one approach to modeling this situation in an ER (Entity-Relation) model. The ER schema assumes the same role as the XSD file for the representation modeling. It acts as a standardized definition format, which can later be translated into various programming languages. The implementation

[1] https://jaxb.java.net/
[2] http://pythonhosted.org/generateDS/
[3] http://pyxb.sourceforge.net/

of the persistence layer API differs from programming language to programming language, yet most are able to reverse engineer classes from a database schema.

Fig. 9.15.: ER Model for the underlying data structure of PublisherResource

9.3.3. Implementation

As pointed out, the Entity (of Interest) is the core concept of the meta-model. According to definition 14, it reflects the point of view the system architect adopts during the modeling phase. Once this point of view has been chosen, it cannot change without severe consequences for the rest of the design. Although the model of a smart room and a smart home share the smart light bulb device, they would not be modeled in the same way. In the first case, the topmost element is the room, composed of windows, HVAC and finally, the smart light bulb. Accordingly, the meta-model compiler creates the following services:

1. one RESTful service representing the use-case, here a smart room,
2. one RESTful service for the smart light bulb
3. one RESTful service for the smart HVAC device
4. one RESTful service for the smart window.

In the second case, the topmost element is the smart home, which is composed of rooms. Each room decomposes further into windows, HVAC and finally, the smart light bulb. Accordingly, the meta-model compiler creates the following services:

1. one RESTful service representing the use-case, here a smart home,
2. one RESTful service for the smart rooms composing the smart home,
3. one RESTful service for the smart light bulb
4. one RESTful service for the smart HVAC device
5. one RESTful service for the smart window.

Both models create similar service skeletons. This is not surprising as in the end both rely on the same sensors and actuators. Therefore, upon implementing the second use-case, it is not necessary the reimplement the created service skeletons for the smart light bulb, the smart HVAC device, the smart window and the smart room. Instead, since these services behave exactly the same as those already created and deployed in the first scenario, these can be reused and integrated in the second scenario. This also fits the philosophy of the xWoT where the world is full of small smart devices ready to be used in different applications. Where software engineering has mandated code reusability for

decades, the xWoT mandates reusability for smart devices. As reusable blocks of the xWoT, they are true components as defined by Cox [9] and introduced in Chapter 6. Whereas for the first example, 4 services need to be implemented, for the second, only one new service needs to be implemented. The rest is simply reused from the previous example. Such an approach greatly speeds up the remaining implementation work and also brings a whole bag of benefits related to component-based architectures.

Code reusability is highly coupled to the way the code is structured. The above discussion shows that for xWoT modules code reusability is not meant for business logic implementation details, but rather for the module as a whole. It proposes an approach where the atomic structure is a full xWoT component. Having solved this first problem, the discussion turns to how the meta-model helps to implement the code necessary for responding to incoming requests. In its simplest form, the REST web service offers the basic CRUD HTTP verbs, each mapping to another functionality of the component. Consequently, the component needs at least four functions, each responsible for one HTTP verb. Figure 9.4 shows the rough overview of the xWoT meta-model, containing the most important elements system architects are concerned with. So far the different types of Resource have only served to distinguish them and to introduce a formal terminology for the xWoT. However, the xWoT meta-model compiler also uses this information to generate the most complete skeletons possible. Although, the presented meta-model of Figure 9.4 captures the main aspects of the xWoT it lacks some important RESTful concepts. The concept of REST resources as a mean of partitioning the space into smaller parts was discussed when the different *Resource classes were introduced. Nevertheless, a SensorResource does not act in the same way as an ActuatorResource or a ServiceResource.

Each resource reacts to a set of requests. It makes most sense for an Actuator Resources to react to `PUT` request, whereas Sensor Resources react to `GET` requests. In terms of any programming language, this means that a resource offers for each request a corresponding function, internally handling the request. Therefore, each HTTP verb [rfc2616] maps to at least one function in the final code. To bind a given function to an HTTP request, different frameworks use different approaches. The Jersey framework uses annotations before each function, other impose naming conventions. Additionally, each function needs to know the kind of content a client is sending (`Content-Type` header) and which type of content a client is expecting (`Accept` header). The xWoT meta-model takes care of this and allows this behavior to be modeled in a very fine-grained manner. It defines the *Method* class containing the parameters: (1) a *MethodOperation* (one of the available HTTP verbs), (2) the type of input data *MethodInput* if any and (3) the type of output data *MethodOutput*. Therefore, each combination of HTTP verb with `Accept` and `Content-Types` leads to one function for the final fulfillment of the exact request. Accordingly, for a resource to respond to a `GET` request with either an `XML` or an `HTML` representation, some code similar to the snippet of Listing 9.6 is needed. This listing contains two methods, both treating `GET` requests. The first returns an XML representation of the requested resource, whereas the second returns an HTML representation. This listing shows the application of the Method class in lines 3 and 12 where they translate in this case to Java methods. Moreover, lines 1 and 10 define that both methods treat `GET` requests. These lines are the results of the compilation of the MethodOperation class. Finally, lines 2 and 11 define which `Accept` headers trigger which method. Therefore, these lines are the result of the MethodInput and -Output compilation.

```
1   @GET
```

```
2     @Produces({MediaType.TEXT_XML, MediaType.APPLICATION_XML, MediaType.
          ↪ APPLICATION_JSON})
3     public Response getUserXml(
4             @QueryParam("start") @DefaultValue("0") int startpartnerships,
5             @QueryParam("size") @DefaultValue("5") int sizepartnerships){
6         AbstractJaxb rep = getUser(startpartnerships, sizepartnerships, Response.Status
              ↪ .OK);
7         return rep.buildResponse(MediaType.TEXT_XML, null, 0, null);
8     }

10    @GET
11    @Produces({MediaType.TEXT_HTML})
12    public Response getUserHtml(
13            @QueryParam("start") @DefaultValue("0") int startpartnerships,
14            @QueryParam("size") @DefaultValue("5") int sizepartnerships){
15        AbstractJaxb rep = getUser(startpartnerships, sizepartnerships, Response.Status
              ↪ .OK);
16        return rep.buildResponse(MediaType.TEXT_HTML, "/WEB-INF/view/user", 0, null);
17    }
```

List. 9.6: Two methods answering `GET` requests

The reader will spot the presence of method parameters in Listing 9.6. In a REST architectural style, a resource is identified through its associated URI. Furthermore the HTTP verb defines the action carried out by the server, whereas the remaining headers fine-tune the input and output, hence, the methods associated with each request are generally parameterless. However, if the HTTP request contains a body, this data needs to be stored somewhere. Also, commonly used query parameters need to be stored in such a way that the method can access them. Like methods, resources can also have parameters. This is the case for URL parameters when they are used to select one item from a list as in `http://example.com/lightbulbs/1/`. Clearly, the last part of the URL is variable, each pointing to another one of the many available light bulbs. However, in terms of code, it does not make sense to write code for each individual light bulb. Instead, the same code can be reused, under the assumption that it is possible to pass this URL parameter to the code.

This final situation is resumed in Figure 9.16. Besides the already discussed elements from Figure 9.4, this model contains the necessary elements to model methods responsible for dealing with incoming RESTful actions, their associated URL and query parameters. To adapt to this situation the meta-model includes a definition of the most common HTTP methods in the `MethodOperation` enumeration. This is not an invention of the meta-model, rather part of the official HTTP definition [rfc2616, Section 9]. The same applies to the definitions of the input format (through the `Content-Type` header) and the output format (through the `Accept` header). The enumerations `MethodOutput` and `MethodInput` take care of this aspect by defining the most common content types in RESTful services. It is however possible (and very likely) to extend this list with future content types. Furthermore, each VirtualEntity is composed of *Method* instances, each one responsible for one available combination of `Method` with `MethodInput` and `MethodOutput`. Finally, the meta-model includes definitions of the two types of parameters: (1) `QUERY` to capture URL parameters and (2) `TEMPLATE` to represent variable URL path segments.

Although, there are other meta-models for RESTful services like Schreier's structural meta-model [73] or Alarcón et al. with their *REST Service Description Model* [1], this

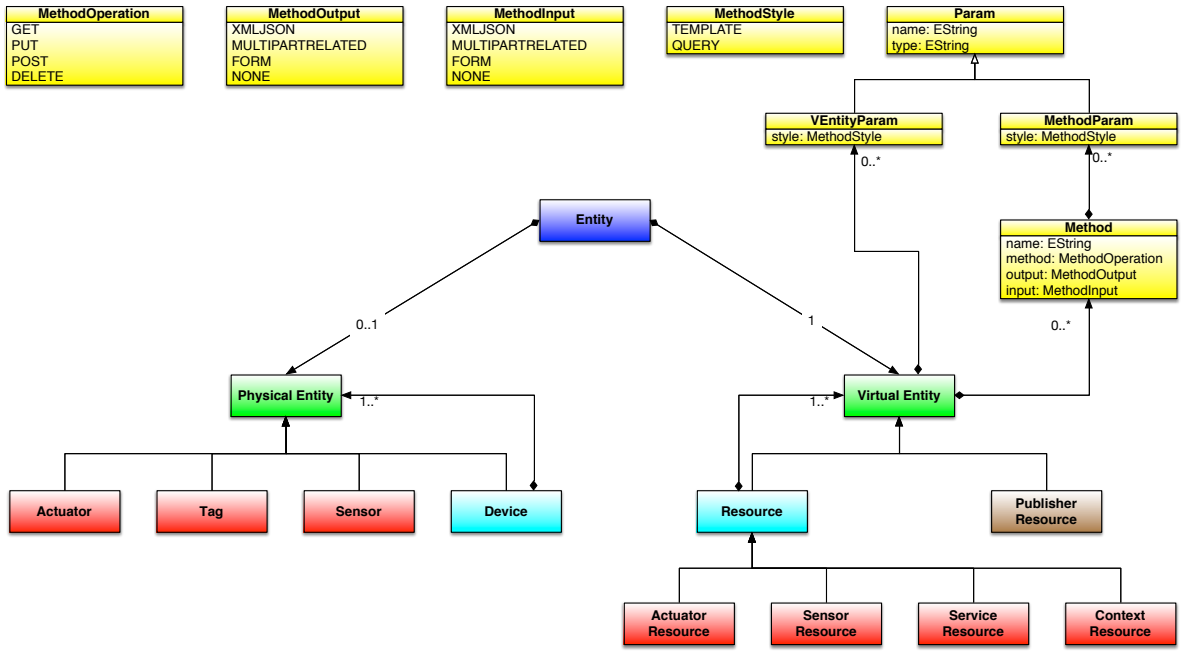


Fig. 9.16.: Detailed meta-model for the xWoT

meta-model is tailored to the xWoT. Alarcón et al. focus more on the links between the resources while Schreier's meta-model is quite complete and aims to model the whole spectrum offered by RESTful web services. While it would be possible to integrate these meta-models into the current xWoT meta-model, this would only artificially bloat the meta-model. On the other hand, integrating other approaches like the IoT-A's reference architecture would be almost impossible. The focus of the latter is too different from the aim of the xWoT meta-model. As will be seen in Section 9.4, the current xWoT meta-model captures all the aspects required to build a fully functional yet, generic, model compiler.

9.4. Component Generation

The methodology supporting users during the creation of new xWoT components is highly coupled to the xWoT meta-model itself. Whereas Section 9.2 formally introduced the meta-model and its associate terminology, Section 9.3 defines the associated methodology. Equipped with these tools, users can easily develop new xWoT compliant components. Up to now, the meta-model has only played a passive role by checking the validity of the built model. Most of the work is done by the users themselves. This section shows how the xWoT meta-model can be enhanced with tools to automate some of these steps.

Before discussing concrete tools, some properties of the meta-model and also the global vision of the xWoT need re-considering. As suggested in Section 8.3 and more precisely in Subsection 8.3.3, one of the goals of the xWoT is to close the gap between primitive smart devices and more complex constructions. To achieve this goal, besides modeling raw sensors, actuators and tags, the xWoT allows combinations of these to build *virtual devices* (called hubs in Subsection 8.3.3). On the one hand, this has the benefit that platforms like cosm or pachube are replaced by a virtual device fully respecting RESTful

guidelines with the advantage that it looks and acts like any other smart device, so the interactions are seamless. This leads, to the `RoomResource` back in Figure 9.14, which is not attached to any sensors or actuator but which rather groups other devices. It is still interesting to be able to address such an embracing resource. On the other hand, creating such bloated virtual devices breaks the original vision of the WoT where the world is full of small intelligent pieces of hardware that can communicate with each other.

To overcome this limitation and keeping the benefits of virtual devices, the xWoT meta-model introduces a clever mechanism which satisfies both needs at the same time. When modeling the physical side, each Device has a `Composed` property. This property already appears in the discussion on how Entities should be modeled in Subsection 9.3.1. It allows the distinction to be made between devices with a physical counterpart in the form of a smart device (sensor, actuator and tag) and devices with no attached smart devices but grouping several of them. By default, this property is set to `false` indicating that the model represents a physical smart device. This flag not only serves to distinguish the two types of Devices in the model, it has also implications for the model's compilation. To fulfill the requirement of individual, small and deployed smart devices, the model compiler creates one service skeleton for each Device with the `Composed` property set to false. Furthermore, if the model starts with a Device with this flag set to true, the compiler creates an additional service representing this composed Device. Since the model compiler mostly works on the virtual side of a model, the `Resource` class and its children also have this attribute. During the translation of the physical to the virtual side, the attribute is simply copied. Therefore, if a Device has this property set to `true`, the corresponding Resource (or subclass) has its `Composed` property also set to `true` and vice versa.

Reconsidering the smart room model in Figure 9.14 as already outlined in Subsection 9.3.3, the model compiler creates one (device)-service for the HVAC Context, one for the Lightbulb Resource and one for the Window Actuator. These three service skeletons will later run directly on their corresponding smart device. Yet, the compiler creates an additional (nodemanager)-service representing the Room Resource. Although this service contains the full use-case hierarchy, a call to `http://example.com/room/havc/` would be delegated the HVAC (device)-service created earlier. Therefore, all the requirements are met: the WoT has its many independent devices which can join or leave a place and, the xWoT has its service(s) representing the modeled scenario making platforms like cosm (almost) obsolete. Additionally, this approach highly encourages the reuse of already deployed components.

One of the benefits of the meta-model are the conventions that come with it. This is true for any meta-model, not just the xWoT meta-model. These conventions and standards are the foundation for tools working with instances of the meta-model. The xWoT meta-model comes bundled with a few tools. Subsection 9.2.4 suggests the adoption of the composite pattern for both the digital and the virtual side of a Thing. Furthermore, it requires a relaxed one-to-one mapping between the physical and the virtual world. These two constraints allow to automatically add the virtual part for any physical side modeled with the xWoT meta-model editor. This is the aim of the first tool: `physical2virtualEntities`. It takes two parameters: the input file containing an xWoT meta-model instance and an output file which will contain the augmented model. Following the definitions in Subsection 9.2.4 this scripts iterates over the devices composing the physical side and build up the virtual side of the smart device. Most steps are fully auto-

matic, however some require user intervention. The script cannot tell whether a Device containing an Actuator and a Sensor should be translated into a Resource containing a Sensor- and an Actuator Resource or whether it should translate it into a single Context Resource. Therefore, the program asks, when necessary whether a Sensor/Actuator couple should be grouped together into a Context. Additionally, the program also asks for each Resource's associated URI. Although this could be derived from the Resource name, it is very likely that the user will tweak this later. By requesting this information during the creation of the virtual side, the user does not need to modify the generated model. Listing 9.7 shows part of the interaction between the user and the scripts when applying them to the model in Figure 9.14. The result of this application is a new model where the Physical Entity is the same as in the input and the Virtual Entity is the generated part. Although the model in Figure 9.9 is done by hand, both the physical and the virtual parts of a generated model, would look almost the same (only the Resource names will differ since generated by the tool).

```
1  ruppena@tungdil:~$ physical2virtualEntities -i smart-room-example.xwot -o smart-room-
       ↪ example_augmented.xwot
2  Specify URI for xwot:Device Room: room
3  I have found the following Nodes:
4  0: <Component name="HVAC" xsi:type="xwot:Device">
5        <Component name="Temperature" xsi:type="xwot:Sensor"/>
6        <Component name="ChangeTemperature" xsi:type="xwot:Actuator"/>
7     </Component>
8  1: <Component name="Lightbulb" xsi:type="xwot:Device">
9        <Component name="LightSensor" xsi:type="xwot:Sensor"/>
10       <Component name="LightSwitch" xsi:type="xwot:Actuator"/>
11    </Component>
12 2: <Component name="Window" xsi:type="xwot:Actuator"/>
13 Is there a ContextResource? [y/n]?n
14 Specify URI for xwot:Device HVAC: hvac
15 I have found the following Nodes:
16 0: <Component name="Temperature" xsi:type="xwot:Sensor"/>
17 1: <Component name="ChangeTemperature" xsi:type="xwot:Actuator"/>
18 Is there a ContextResource? [y/n]?y
19 Combining Node Temperature and Node ChangeTemperature into ContextResource
20 Specify URI for ContextResource TemperatureChangeTemperature: hvac
21 Has this ContextResource a publisher? [y/n] ?y
22 Specify URI for xwot:Device Lightbulb:
23 ...
24 ruppena@tungdil:~$
```

List. 9.7: Applying the `physical2virtualEntities` tool to the use-case of Subsection 9.3.1

The tools provided also take special care of the publishing mechanism. Besides actuators, sensors and services, the meta-model also defines a special kind of resource responsible for pushing information (mostly events) back to clients. Figure 9.7 shows where this Resource sits in the meta-model and Subsection 9.2.4 describes the possible combinations. Therefore, the `physical2virtualEntities` tool asks for each Sensor Resource and each Context Resource whether a Publisher Resource should be added. Listing 9.7 exhibits this behaviour in line 21 where the `TemperatureChangeTemperature` Resource gets a publisher. Nothing more needs to be specified for this to work. On the generated model (similar to the one in Figure 9.7) the Context Resource now contains a child node, the Publisher Resource.

```
ruppena@tungdil:Test$ model2Python -i smart-room-example_enhanced.xwot
INFO - Start processing
INFO - Creating Server: REST-Servers/NM-Room_Server
INFO - Creating Server: REST-Servers/_HVAC_Server
INFO - Creating Server: REST-Servers/_Lightbulb_Server
INFO - Creating Server: REST-Servers/_Window_Server
INFO - Successfully created the necessary service(s)
```

List. 9.8: Applying the `model2Python` tool to the use-case of Subsection 9.3.1

The second tool bundled with the meta-model, `model2Python`, is a model compiler. It takes as input a valid xWoT model and creates as output REST service skeletons ready to be deployed on the individual devices. Listing 9.8 shows that the tool generates 4 different services according to the `Composed` properties set on the different Resources. Services prefixed with `NM` are application scenario services or node manager services, the result of a Resource having the `Composed` property set to `true`. In the current use-case, there is only one such a service representing the Entity. All other services have the `Composed` property set to `false`, meaning they are device services and are prefixed with `_`. The lefthand side of Figure 9.17 shows the generated skeleton structure for the `_HVAC_Server`. Besides some documentation and bootstrap scripts, each REST resource has its own python module. The `_HVAC_Server` is composed of the HVAC resource plus the publisher, hence, the HVAC resource gets its own python module. Furthermore, the publisher is translated into two modules, one representing the Publisher Resource and the other the subscribed users. Why two python modules are necessary to represent an xWoT publisher will be discussed later. Additionally, the project contains a skeleton bridging the gap between business logic code and the underlying hardware sensors and actuators (`Hadware_Monitor.py`). The remaining modules implement a very basic announcing mechanism, which can be used for the automatic discovery of smart devices. Unlike the first tool, this one runs without human intervention. Since everything is defined in the model, the compiler simply translates it into code.

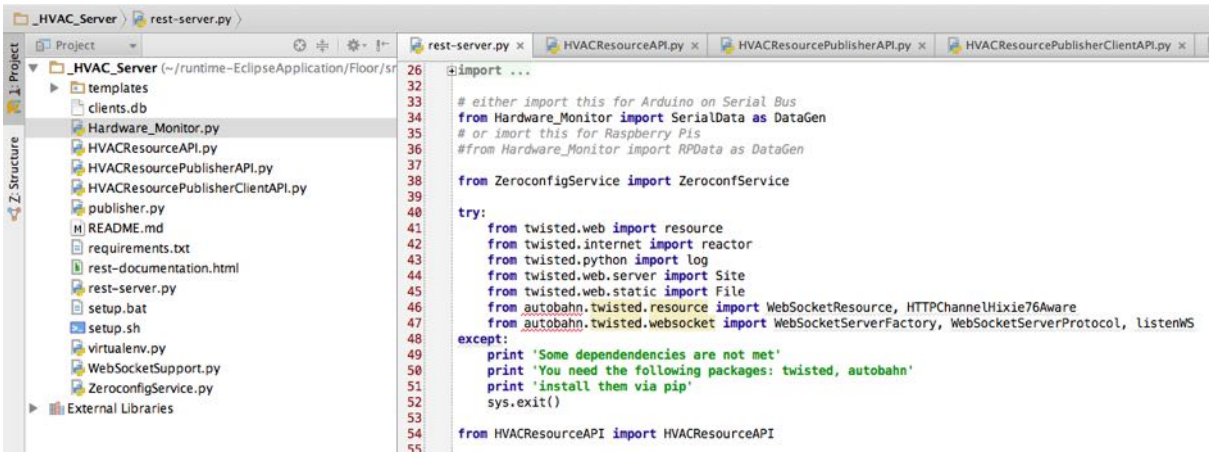


Fig. 9.17.: Generated REST service skeleton for the HVAC resource

As for the `physical2virtualEntities`, the `model2Python` compiler also takes special care of the Publisher Resource. During the first phase of generation, the `physical2-virtualEntities` creates the Publisher Resource and sets its properties to some default values. Since the Publisher Resource class inherits all the properties of the Resource class, it also inherits the `Uri` attribute (set to `pub` during the first phase). By design, a child class

can only extend its father but not restrict him, so it is possible to adapt this property prior to the compilation. However, its default value is one of the properties of the meta-model and other models should not change it. Subsection 8.2.2 explains a few approaches to push information back to clients. The subsection concludes that depending on the needs, one or the other approach might be better. For obvious reasons, the meta-model can't decide which approach is the most suitable one. Instead of asking the user for advice, the model compiler translates each Publisher Resource into two pushing mechanisms:

(1) A WebSocket connection is deployed under `pub/` (note the trailing slash).

(2) Clients register for events by sending a `POST` request to `pub` containing all the necessary information so that the server can later contact the client. This implements an inversion of control whereby the service becomes the client of the user supplied service. The information joined to the `POST` request upon subscribing for notifications includes at least an URL and an `HTTP` method (either `PUT` or `POST`). Additionally, for each client, the service creates a sub resource `pub/{id}` where `id` uniquely identifies a given client. A client can use this URL to modify (`PUT`) or cancel (`DELETE`) his subscription later.

```
import sqlite3
import requests
import json
import logging

class Publisher():
    def __init__(self):
        self.__database = 'clients.db'

    predicate = """
    def testEvent(*values):
        if values[0] > 30:
            return true;
        return false;
"""

    def publish(self, values):
        """Entry point for external scripts to send a notification to all subscribers """
        clients = self.__executeQuery("Select * from Subscriber where resourceid=1 order by id")
        for client in clients:
            #TODO do this in a new thread for each client
            predicate = client["predicate"]
            execute predicate
            if testEvent(values):
                self.__updateClient(client, values)

    def __updateClient(self, client, values):
        """Updates each individual client by sending the corresponding request"""
        #TODO update client with a REST request
```

List. 9.9: Predicate execution to test whether a notification needs to be sent

Choosing this approach gives each xWoT device the best of all pushing mechanisms. If a client is interested in very frequent events over short periods of time, he can connect

over a WebSocket. If instead, the client is more interested into less frequent events, he can subscribe for them. Ideally, upon subscribing, the client sends, together with the other information, a predicate limiting the number of notifications. For each generated event, the server then evaluates for each client the corresponding predicate to determine whether a notification is sent or not. Listing 9.9 shows a partial python code for how such a predicate could be implemented. This is, of course, only a very simplistic approach. Executing strings as code is dangerous especially if the string is submitted by users. Yet, this simple example shows how notifications can be handled from an architectural point of view. Programmatically, it would surely be a better approach to implement this with some custom mini language, which can then be interpreted. This still lets users submit a string, which is executed on the server. However, imposing such a mini language removes almost all programmatic constructs, therefore the threat would be limited. Such a language can be very simple and only contain basic comparators. For an even stronger approach, instead of giving the users the full power of predicates, something like regular expressions would be sufficient to limit the events. The most suitable approach would be to define a Domain Specific Language (DSL) tailored to this domain of application.

The way the meta-model compiler works is quite similar to Model Driven Architecture (MDA) works [B19]. Starting from an abstract description of the SUS complying to the meta-model, a tool chain first translates this early model into a more sophisticated model. This first step alone conforms to the definition of MDA: translating descriptions (the model) into code. Here the generated code is again a model, augmented with the virtual side of the modeled entity. In the second step, the tool chain translates this second model into ready to use code skeletons. Again, by transforming the model into code (RESTful web services), this tool perfectly fits the definition of MDA. Generally, MDA asks for a DSL (Domain Specific Language) to describe a system in a platform independent manner, the task of the xWoT meta-model. Being expressed in Ecore, the meta-model is platform independent. Furthermore, the availability of Eclipse on various platforms guarantees that the xWoT meta-model can be used on all supported platforms to model xWoT entities. Additionally, the tool chain is written in Python, liberating it from any specific OS (Operating System) . Hence, not only can most platforms be used to express xWoT models, they can also compile them to xWoT service skeletons.

9.5. The Web as a Container

Chapter 6, Section 6.4 in the discussion of software components, Definition 4 asks for a life cycle of components. The lifecycle has an influence on the current capabilities of the component and is imposed by the supporting container. Furthermore, according to Definition 12, the xWoT is a component based approach, where the components are the basic bricks composing the xWoT. Therefore, deployed xWoT components have an associated life cycle. Regarding these components, two different and disjoint life cycles can be identified: (1) A *development life cycle* taking care of the component creation. (2) A *runtime life cycle* starting with the deployment of the component on the smart device and remaining active throughout the lifespan of the smart device.

The *development life* cycle is the first cycle of each xWoT component. Starting with an Entity, the system architect first considers what a smart device would look like in the real world. For example, to create a smart door, requires a door. Furthermore, a door can be

locked and unlocked, opened and closed. Therefore, the system architect concludes that a smart door needs two actuators to fulfill this requirement. It might also be of interest to know whether the door is unlocked and whether it is open or closed. Thus, two sensors track these two physical properties. In the model, the system architect will finally model the door as composed of two devices each having one sensor and one actuator. One device is responsible for representing the open and closed property, whereas the other is responsible for representing the locked and unlocked property. This reasoning results in a first xWoT model reflecting the physical properties of the smart object. This model is then refined with the aid of the provided tool chain and as result the model is enhanced with a virtual side representing the smart object in the virtual world. Here, the system architect can fine tune the translation by adding more resources or by influencing the hierarchy of the generated ones. Of course, it is also possible to rename the resources of the associated URIs.

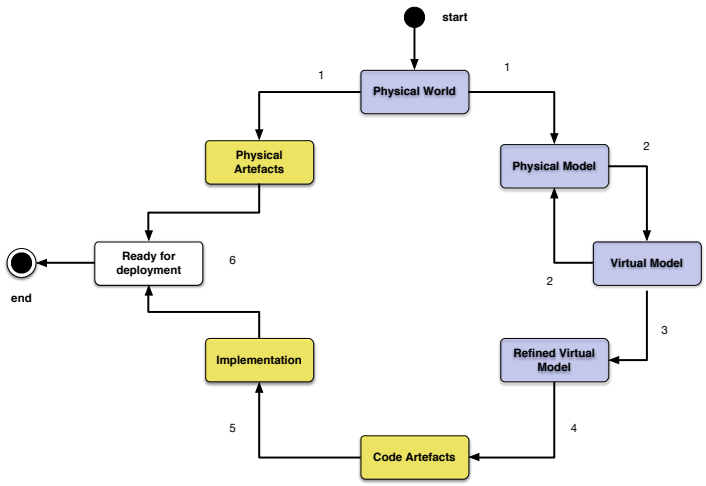


Fig. 9.18.: Schema of the development life cycle

As soon as the model reaches a stable state, with the aid of the compiler, it is translated into code artifacts. These skeletons represent the outer structure of the future component, but to be fully functional the developers need to fill the code gaps. This means defining the input and output formats as well as wiring up the code with the underlying hardware of the different sensors and actuators. Once this task is completed, the smart device is considered *ready to deploy*. In this state, the hardware is built according to the physical model. The different sensors and actuators are physically grouped to Devices, which in turn are placed accordingly. Moreover, for each Device as well as for the Entity, a RESTful web service is ready and the code sits on the corresponding Device. This step is the last step in the development cycle. The Entity is considered stable and once deployed it can serve incoming requests covered by the runtime lifecycle.

Code is always subject to change as are xWoT smart devices. With changing requirement or technological enhancements, either the RESTful web service or the underlying hardware is likely to change. To incorporate such changes, either the code, the hardware or both need some modifications. It seems that the life cycle in Figure 9.18 does not handle such changes. Yet, as pointed out in Section 6.3, such modifications lead to a new version of the API deployed under a new URI, so to incorporate changes after the release of the initial version, the development lifecycle starts again. Of course, some steps can

be copied from the initial lifecycle; nonetheless the new life cycle eventually leads to a new xWoT component ready to be deployed. Since this life cycle describes the different states of a smart device throughout the development phase, it is highly coupled with the development flow shown in Figure 9.10.

The *runtime lifecycle* takes care of all aspects of the component's life cycle once it is ready to deploy. The final state of the development life cycle thus corresponds to the initial state of the runtime life cycle. If a component is ready to be deployed this means its code has been finished and tested. To deploy the code on the component, one can either copy the code or fetch it from a repository like git. If the code is compressed (zip) or packaged (tar), it needs to be extracted/unpacked at its final destination. The tool chain provided with the meta-model translates xWoT models into Python code. Before executing the code, the necessary dependencies need to be installed. Setuptools [WEB13] is the Python way of installing Python software into an existing OS. A corresponding setup scripts tells setuptools what dependencies are needed and where to install the tools. Whereas setuptools is great for installing components into the OS, there are simpler approaches for dependency handling. The `pip` package manager can easily install missing package in an existing python installation. Additionally, it is a good practice to create separate Python environments for separate projects. In software engineering, this approach has become more and more of a standard these days. Ruby makes heavy use of this pragma. Ruby uses `rvm` to manage separate Ruby environments and Python uses `virtualenv`. This approach only requires one command to create a new Python environment to which all missing dependencies are added through `pip`. Listing 9.10 shows the first few deployment steps for a component available in a git repository. Of course, line 7 is repeated for each necessary dependency.

```
1  ruppena@tungdil:~$ git co https://example.com/git/Component.git
2  ruppena@tungdil:~$ cd Component
3  ruppena@tungdil:Component$ python virtualenv.py xwot
4  New python executable in xwot/bin/python
5  Installing setuptools.............................done.
6  Installing pip...........................done.
7  ruppena@tungdil:Component$ xwot/bin/pip install autobahn
8  ...
9  ....
10 ruppena@tungdil:Component$ xwot/bin/python rest-server.py
```

List. 9.10: Steps executed during Deployment

The RESTful web service is now ready on the physical device but not yet running, that is, the component is ready and installed in its environment. In the stopped state, no clients are served and neither the actuators nor the sensors are working. To activate the virtual side of any smart device, the component needs to be `started`. This is as simple as running the `rest-server.py` module taking care of the rest. If at some point the service is stopped, it will no longer serve incoming requests and the virtual façade of the smart device disappears. Therefore, the component is again in the stopped state. Additionally, a component can be undeployed, simply by removing all the code artifacts and the RESTful web service from the hardware. Usually, this is only necessary to deploy a new version of he API.

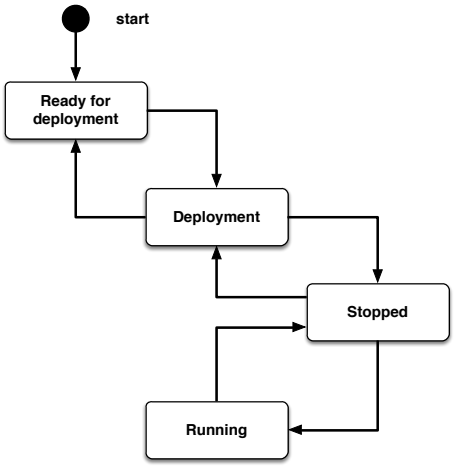


Fig. 9.19.: Schema of the runtime life cycle

Every time a component is in the running state, yet another life cycle kicks in. This is the life cycle of the supporting container, in this case, the Web. At first, the component is ready and waiting for clients and just occupies some memory but does nothing more. As soon as a client connects, some memory is allocated to this client and the request is served. There are two major approaches for serving clients: (1) *Blocking requests:* only one client can use the connection with the web server at any given time. This approach is simple to implement but rather limited. (2) *Non-Blocking requests:* allow the simultaneous connection of multiple clients, each getting some memory allocated and all requests are served in parallel. Obviously, this approach is superior, but implementing frameworks tend to be bigger. Whether a RESTful web service is blocking or not, often depends on the chosen web framework, which in turn, may depend on the underlying hardware capabilities. In our case, the adopted framework is Twisted [WEB65], which is asynchronous by nature and thus allows for non-blocking requests. When the request is answered, the occupied memory is freed up.

Although the web serves as a container for any RESTful component, it only gives the rough structure of how interactions happen. This cycle is far less complex than comparables ones using Java application servers with memory handling, messages queues, database pools, security policies etc. However, the component itself also influences the life cycle in part. As soon as an xWoT component is running, if it has a least one publisher, the latter also runs. However, by their nature, publisher resources have a slightly different life cycle. Subsection 9.4 discussed how Publisher Resources get translated into code. On the one hand, a WebSocket endpoint is created, listening for incoming WebSocket requests. As soon as at least one client is connected, the WebSocket starts broadcasting events to its subscribers. To keep the connection with the clients up, a heartbeat message is send regularly. The amount of time between two heartbeat messages is implementation-specific. Heartbeat messages are only sent if no other modification was pushed to subscribers in the given amount of time. If the last client unsubscribed from the WebSocket, it pauses until another client connects. The second publisher endpoint works by sending the event as a REST request to a "client server". Since the xWoT component plays the role of a client, the components life cycle has no special effect on these outbound events. For both pushing mechanisms, the component itself needs to be aware of an event happening. How this is implemented depends on the underlying hardware and the communication

between the code and the hardware (see the `Hardware_Monitor.py` module). If an event happens, the `publisher.py` module sends it to all WebSocket subscribers. Furthermore, for each subscribed client, it checks whether his alert condition is met, and if so, a `PUT` or `POST` request, depending on the client's preferences, is sent.

When we discussed software component in Chapter 6, the life cycle was one of the key element in the definition of a component. The xWoT component fulfilled this requirement by relying on the Web as a container and therefore depending on it for its runtime life cycle. This chapter has shown that the runtime life cycle is quite simplistic compared to other component's life cycle, like Java Stateful Session Beans. This is mainly because the supporting container is the Web, having only a small life cycle. However, a closer look shows we have seen that an xWoT component is supported by several associated life cycles, each taking care of a different aspect. The different publishing mechanisms play an important role here.

10 Validation

10.1. Introduction

So far the discussion has covered the historical foundation of the Web and its associated success story throughout its history, from the early days to Web 3.0. Additionally, we have seen how Model Driven Architectures can help developers create new applications by adopting a Domain Specific Language tailored for a given domain. In this case, the DSL is the xWoT meta-model. It has introduced a formalism for naming the different elements of the xWoT and their relationship. With the help of an Eclipse plugin, it is possible to create valid instances of this meta-model which serve as input for the different tools. The latter compile the model either into more sophisticated models or into code skeletons.

This chapter discusses use-cases to show how the xWoT meta-model and its associated tools can be used in real life situations. Section 10.2 starts by introducing three small use-cases each underlining different aspects of the xWoT. Section 10.3 takes a look at a slightly more complicated scenario to show that the xWoT meta-model can efficiently model any situation. Finally, the chapter concludes with a vision of how the xWoT will perform and evolve.

10.2. Some Small Examples

This section discusses and showcases some simple small use-cases. Some of them already served as examples throughout this thesis and will now be examined in detail. The approach remains the same for all use-cases:

1. *Introduce the use-case.* Discuss the use-case and its scope. This also includes a short rationale for why the use-case is within the scope of the xWoT. Furthermore, this discussion also clearly defines the boundaries of the current use-case.
2. *Design the UML use-case diagram.* From the initial discussion, it is possible to derive a use-case diagram exposing the functionality of the smart device. Although some aspects are shared between different use-cases like the publisher mechanism, all use-case diagrams mention them.
3. *Model the Physical Side.* The physical side of the entity can be modeled based on the description and the use-case diagram. Possibly, this physical side only partly covers the use-case model since not all involved actions translate to something physical. This represents the first step of the activity diagram of Figure 9.10. The outcome of this step serves as input for the following ones.
4. *Launch first tool.* With the aid of the `physical2virtualEntities` tool, the physical model is augmented with its virtual counterpart. Although, this step involves human intervention to choose URIs, nothing else is done here.
5. *Enhance the generated model with virtual only resources.* After the automatic translation of the physical to virtual resources, those still missing from the use-case diagram are added to their corresponding places.
6. *Generate the code skeletons.* Once the model is completed and with the help of the `model2Python` tool, the code skeletons are generated, one for each device service and one for the application scenario service.
7. *Build the hardware.* Wire the necessary sensors and actuators to a breadboard and implement the necessary logic to pilot the device. Ideally, the code has methods to read sensor values and to modify actuators.
8. *Fill in the code gaps.* Following the recipe in Figure 9.10 and Subsections 9.3.2 and 9.3.3, fill the gaps in the generated code skeletons to build a functional RESTful web services.
9. *Deploy and run.* In this last step the software is deployed on the hardware and installed in the physical world. The system is now ready to be used.

The main focus of these use-cases is the xWoT meta-model and its associated tools,therefore, not all use-cases will go through all of the steps above. Building the hardware side of a smart device, for example, is outside the scope of the meta-model as well as defining the inputs and outputs with the help of XSD schemas. Therefore, some use-cases are limited to a preliminary analysis of the specific requirements, the modeling as an xWoT compatible model and some basic code gap filling.

The connection with the hardware is a central piece of the generated code. Basically, there are two approaches to interacting with hardware: (1) Exchange messages over a serial bus, the preferred way when dealing, for example, with Arduino prototypes. (2) Low level input output with signals directly on pins. This approach is similar to what is done on the Arduino directly to interact with the connected sensors and actuators. However, this

approach is also the only choice when dealing with GPIO (General-purpose input/output) pins like the ones available on a Raspberry Pi. If the communication with the raw sensors and actuators is done directly over the digital/analog pins where the hardware is connected, it is sufficient to respect the hardware specification to read sensor values and write to actuators. Generally, this implies a mixture of the correct timing plus reading/writing the right sequence of raising and falling signals. Often, the hardware vendors or the community provide ready to use libraries abstracting from the raw hardware and proposing high-level functions to interact with the hardware. Therefore, using this approach, the hardware integrates into code just like any other code segment. On the other hand, if the software communicates with the hardware over a serial bus, no additional libraries are needed. Yet, the serial bus is just a (duplex) channel between two endpoints quite like a WebSocket connection. In order to communicate, the peers need to agree on a message format. The firmata protocol is one such example of a message format [WEB17].

A hardware mock simplifies the hardware side during the use-cases. This is quite a frequent approach in software engineering. As long as something is not yet implemented but already specified, it is sufficient to provide a dummy implementation sending back fixed responses. A mock for a login service would for example always return `true`. This allows developers depending on this functionality to continue developing their own code. The same approach can be applied to hardware. Instead of building the full hardware first, it is possible to mock it up. A prerequisite to mock hardware is to already know the concrete sensor and actuator layout (and potentially also their corresponding pins). Using a mock up for the hardware also has another benefit. Testing the business logic code with real sensors and actuator can be painful. Not only is it necessary to bring the system back (by hand) to the initial state between different runs but sometimes it can also be quite frustrating to make a sensor change its value (imagine a gravity force sensor). If they are mock sensors and actuators however, bringing them into a given state is just a matter of a few clicks. Mocking up hardware is especially easy when the hardware communicates with the rest of the code over a serial bus, as in this case it is sufficient to define the possible messages, so that a piece of code can then send such prefabricated messages on the serial bus.

```
1 {"humidity":"45", "electricity":"1"}
```

List. 10.1: JSON message exchanged over the serial bus

To speed up the development of xWoT smart devices we have developed a general-purpose hardware mock-up based on serial exchanges. To keep things simple we opted for our own message protocol instead of adopting something like firmata. Of course, this may limit the usefulness of the mock hardware but it has shown that this approach is sufficient in the examined use-cases. All exchanged messages are JSON strings reflecting the state of the hardware. Each element in the JSON string represents one element of the circuit and its associated value. Listing 10.1 shows such a message for a smart air humidifier composed of a humidity sensor and a power switch. Listing 10.1 describes the full state of the system: the current humidity is of 45% and the device is switched on. Each element of the JSON string designates one component of the electrical circuit, thus, the `humidity` element designates a DHT11 sensor, whereas the `electricity` stands for a relay module. Although very simple, this message format has the advantage of being easily debuggable since it is human readable. Furthermore, since JSON (de)serializers are available for

many platforms and programming languages, its implementation in code is very easy and straightforward. Such JSON messages can not only be sent from the hardware to the code to inform the latter about the state of the hardware, but also in the other direction, from the code to the hardware to influence actuators.

```
1 ruppena@tungdil:~$ screen /dev/ttyACM0 9600
2 humidity": "34.00"}
3 {"temperature": "24.00", "humidity": "34.00"}
4 {"temperature": "24.00", "humidity": "34.00"}
5 {"temperature": "24.00", "humidity": "34.00"}
```

List. 10.2: Reading from a serial connection

On Unix systems, when a device like an Arduino with a serial interface gets connected, the OS allocates a new device in the `/dev` folder. From this point on, programs can open this file and read and write on the serial bus. Therefore, having a serial port file seems to be a requirement when mocking-up hardware. However unlike a LaTeXdocument, files sitting in `/dev` are not normal files and therefore they can't be opened and read like standard files. Listing 10.2 shows the content of such a serial device file dumped with a serial console applications `screen`. The listing also shows a problem which can occur when reading them: messages can get truncated when no special measures are taken. Also the listing can give the impression that lines 2 to 5 are the content of this device file. However, this is false, the device file never contains anything. If nobody is there to listen to the serial bus, then the messages just disappear. Accordingly, the hardware mock-up also needs such a special device file when starting. However, these file are automatically created and setup by the OS as soon as the hardware is plugged in. To overcome this limitation, a serial connection between two parts can be simulated with `socat`. This command line tool was originally developed as the equivalent of `cat` for sockets. Consequently it can `cat` the input of one socket into another one. Finally, the socket can be anything like a TCP endpoint but also a virtual serial device. Listings 10.3, 10.4 and 10.5 show how `socat` can be used; first to create two linked virtual serial devices (Listing 10.3) and second how these two sockets can be used for communication between two processes (Listings 10.4 and 10.5).

```
1 ruppena@tungdil:~$ socat -d -d PTY,link=$HOME/COM1,b9600 PTY,link=$HOME/COM2,b9600
2 2015/01/21 18:58:29 socat[17441] N PTY is /dev/pts/14
3 2015/01/21 18:58:29 socat[17441] N PTY is /dev/pts/15
4 2015/01/21 18:58:29 socat[17441] N starting data transfer loop with FDs [3,3] and
    ↪ [5,5]
```

List. 10.3: Creating two virtual serial devices

```
1 ruppena@tungdil:~$ echo "Hello World"
    ↪ > COM2
```

List. 10.4: Writing on one side of the virtual serial devices

```
1 ruppena@tungdil:~$ cat COM1
2 Hello World!
```

List. 10.5: Listening on the other side of the virtual serial devices

Now all the necessary tools are in place to build a hardware mock-up for quicker development and testing purposes. It might seem a poor choice at first glance to limit the

mock hardware to serial data exchanges with a message format as defined previously. However, in a well-designed code, communication with the hardware is encapsulated in a class. This is also the case for the generated skeletons. Back in Figure 9.17 (page 173) the `Hardware_Monitor` module was briefly shown, taking care of all interactions with the hardware. Now, in a situation where the final hardware does not communicate over a serial bus, it is sufficient to slightly adapt the code here to read and write the values without changing the modules' public interface. This way, only the inner guts of this module needs to adapt to the final hardware. Based on these considerations, we have built a generic hardware mock-up as a single page web application. This seems to be the best compromise between easy deployment and platform independence. Figure 10.1 shows the virtual breadboard of the mock hardware. The menu on the left contains various prefabricated sensors and actuators which can be activated on the breadboard space on the righthand side. Each menu item corresponds to a full device.

The upper menu contains a few sensors to build simple scenarios. The middle menu contains actuators. These two menus merely exist for testing purposes. It is unlikely that the hardware side of a smart device would only be composed of a single actuator or sensor. The bottom menu contains more complex devices built from a combination of sensors and actuators. The breadboard space on the right side can accommodate several devices simultaneously, as long as each uses a different virtual serial port. This allows simulating different devices at once. Since the model compiler generates one RESTful web service for each non-decomposable device it makes sense that the mock hardware should also accommodate multiple devices together. This is also demonstrated by Figure 10.1, where the breadboard contains two devices one called *Meteo Sensor* and representing a rudimentary meteo-station and the second the *Smart Light Bulb* of Subsection 10.2.1.

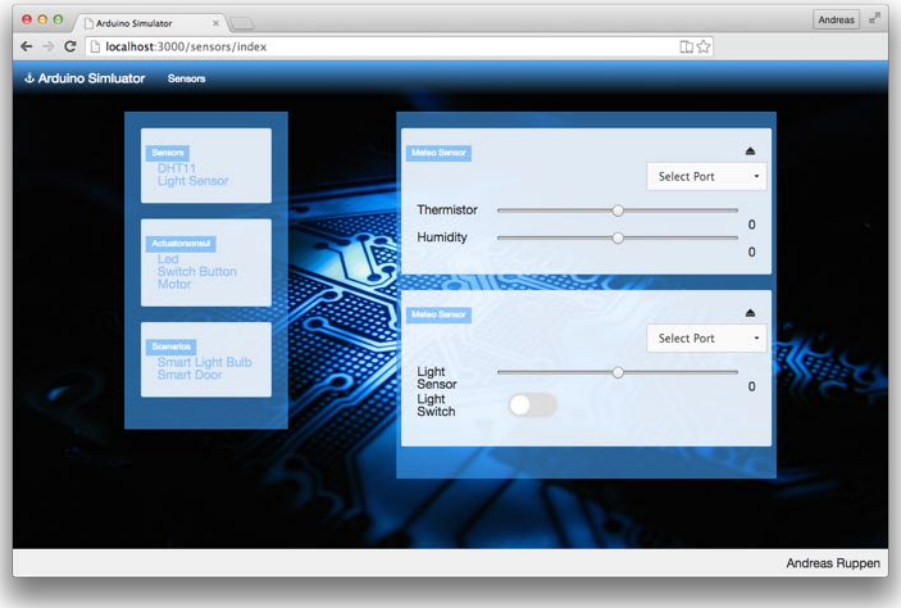


Fig. 10.1.: Hardware Mock Application running in a Browser

New devices can be easily added to the mock application code. Since the application is written in Ruby with the Rails framework, it embraces the Rails MVC structure in particular, the Rails partial rendering mechanism for HTML pages. To add a new device is sim-

ply a matter of creating a new file in `views/sensors/` and giving it a name respecting the Rails partial rendering naming conventions (it needs to start with an underscore). Thus, to create a LED smart device, it is sufficient to create a new file called `_led.html.erb` and add it to the menu in `views/sensors/index.html.erb`. Listing 10.6 shows the content of the `_led.html.erb` file defining what the LED (Light-emitting Diode) breadboard looks like. Lines 7 to 16 are necessary for all devices and print a drop-down menu to select the virtual serial port to which this device is connected. The device definition itself happens in lines 17 to 23. Each component of a device gets an `id` which is composed of a literal plus a random number `arduinoNumber`, which is only used for internal logic. The literal part however composes the JSON messages this device sends and accepts. Additionally, each device has a name, defined on line 5. The remaining elements of the listing are the same for any new device and serve either for internal purposes (`onchange` attribute) or are CSS attributes. Therefore, in the case of a simple LED, defining such devices is a one liner (line 20).

```
1  <div class="row arduino<%=arduinoNumber%>">
2    <div class="panel callout radius">
3      <div class="fi-eject right"
           ↪ onclick="removeArduino('arduino<%=arduinoNumber%>')"></div>
4      <div class="row">
5        <span class="label">LED</span>
6      </div>
7      <div class="row">
8        <select id="serialPortSelector" class="com1 columns large-4 right"
             ↪ onchange="updateComPort('arduino<%=arduinoNumber%>', this)">
9          <option value="null">Select Port</option>
10         <option value="COM1">COM1</option>
11         <option value="COM3">COM3</option>
12         <option value="COM5">COM5</option>
13         <option value="COM7">COM7</option>
14         <option value="COM9">COM9</option>
15       </select>
16     </div>
17     <div class="row">
18       <span class="columns large-3">LED </span>
19       <div class="columns left switch round large-4">
20         <input id="led<%=arduinoNumber%>" type="checkbox"
               ↪ onchange="xwotcallback('arduino<%=arduinoNumber%>')" class="xwot
               ↪ actuator">
21         <label for="led<%=arduinoNumber%>"></label>
22       </div>
23     </div>
24   </div>
25 </div>
```

List. 10.6: Defining a new smart device for the Hardware Mock

At this stage, all the necessary tools are in place to build xWoT smart devices and follow the steps enumerated earlier in this subsection. UML use-case diagrams are useful for the requirements analysis. The xWoT meta-model and its associated tools support the developer during the design phase and build code skeletons. The mock hardware allows the required hardware to be quickly simulated instead of building it. Finally, input and output formats are defined in XSD for which just any modern IDE (Integrated

Development Environment) contains an editor with syntax highlighting. The same IDE can also be used to fill the code gaps in the generated skeletons.

10.2.1. Smart Light Bulb

The first use-case considers a smart light bulb. Subsections 8.2.3 and 8.4 have already taken a smart light bulb as examples and discussed various aspects related to the xWoT and the meta-model. This subsection however, concentrates on a simpler version of the smart light bulb. Unlike previously, there is no presence sensor, just the raw light bulb translated into the virtual world. Furthermore, in contrast with previous discussions of a smart light bulb, this subsection shows all aspects, from the analysis to the final build. The primary analysis concludes that in the physical world, a light bulb can be switched on and off. Additionally, by looking at the light bulb, a person can conclude whether it is currently on or off. To make a light bulb smart, these properties and actions need to be transported to the virtual world. Figure 10.2 shows all possible interactions. On the left are the three possible interactions: *Switch Light on*, *Switch Light off* and *Get State*. Moreover, in the physical world, it is possible to notice when a light bulb is switched on or off without staring at it. This phenomenon of passive observation translates to a publisher mechanism in the virtual world. The right hand side of the use-case diagram shows how this publisher works. Both, the *Switch Light on* and the *Switch Light off* use-case can generate an event, each of the same type. The bottom right part shows the possible interactions with the publisher. These will remain the same for any xWoT publisher and are given by the xWoT model compiler.

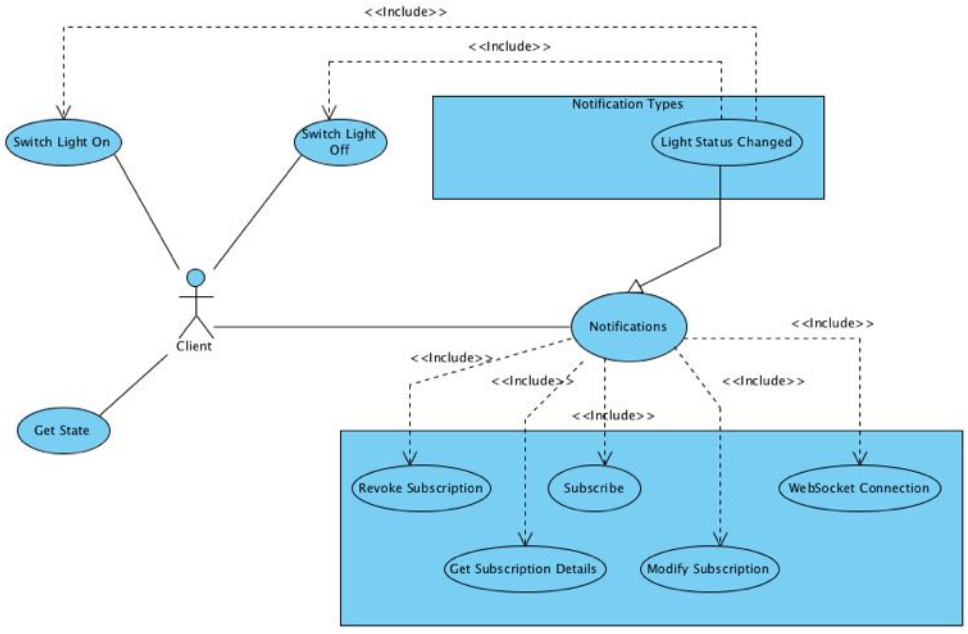


Fig. 10.2.: Use-Case diagram of a simple Smart Light Bulb

Now the use-case has been explained, a model of the physical side of the smart light bulb can be created. This is merely a one-to-one translation of the use-case to physical devices. The topmost element of the smart light bulb model, the Entity, is the smart light bulb. Although there are thousands of light bulbs, and a dozen in a house, only one needs to be modeled. This is sufficient to render all the light bulbs smart since the outcome of

this process can later be deployed on each individual light bulb. On the physical side, the smart light bulb is made of a *Light Bulb Device* grouping an actuator *OnOff* to switch the light bulb on and off and a sensor that measure its current *State*.

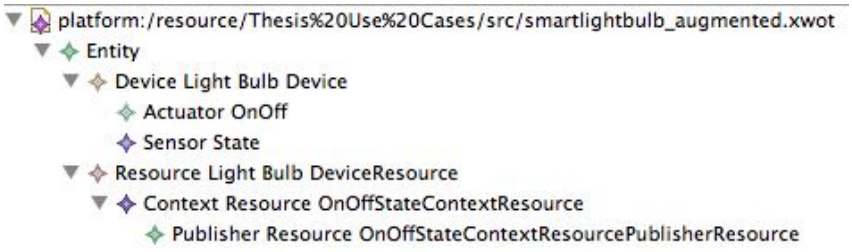


Fig. 10.3.: xWoT model for the smart light bulb

This initial physical model then serves as input for the `physical2virtualEntities` script constructing the virtual side of the smart light bulb. In this case, it is impossible for the property observed by the sensor to change without an action by the actuator. Since the output of the sensor is directly coupled with the action of the actuator, they form a `Context Resource`. Furthermore, the passive observation capability of the real world that was translated as the *Notification* use case gives birth to a `Publisher Resource`. Figure 10.3 shows the outcome of these first two modeling steps.

Since the discussed model completely describes the system, there is no need to add any other virtual-only resource. Therefore, Figure 10.3 represents the final xWoT model for the smart light bulb, which can now be compiled into code skeletons. The `model2Python` compiler generates only one device service and no associated node manager service. Given that the scenario is only made up of one non-decomposable device, the generated service represents exactly one smart light bulb and there is no need to represent anything else. Listing 10.7 shows that the generated service has a structure similar to the one presented in Figure 9.17. The difference lies in the different `*ResourceAPI` modules that are created. The compiler creates the `OnOffStateContextResourceAPI`, which stands for the smart light bulb and accepts `GET` requests for querying the state of the light bulb and `PUT` requests to influence this state. Additionally, the two modules for the publisher associated with this resource are also created.

The next step is all about the hardware. Instead of building it with real sensors and actuators, mock hardware applications are used. The code chunk creating a smart light bulb device is quite small and easy to understand. Although the model explains that the physical side of the smart light bulb is composed of a sensor and an actuator, the mock set-up only contains a switch button. This is possible since the sensor and actuator form a context. Therefore, the code necessary to represent the smart light bulb is similar to the example discussed in Listing 10.6. It outputs and consumes the following message format: `{"light":"on"}` or `{"light":"off"}` depending on the current state or action to execute. Building the circuit with actual hardware is also straightforward. Figure 10.4 shows the electrical wiring necessary to drive a LED from a raspberry pi. We chose the combination of a raspberry pi with a connected LED instead of an actual 220V light bulb simply for safety reasons. The chances are high that the electronic circuit in the Philips Hue light bulbs for example, will be quite similar (plus the ZigBee hardware for communication), thus validating the model.

```
ruppena@tungdil:src$ ll REST-Servers/lb_Server/
total 464
-rw-rw-r-- 1 ruppena staff 4.3K Jan 21 10:13 Hardware_Monitor.py
-rw-r--r-- 1 ruppena staff 3.7K Jan 22 11:45 Light Bulb DeviceResourceAPI.py
-rw-r--r-- 1 ruppena staff 3.7K Jan 22 11:45 OnOffStateContextResourceAPI.py
-rw-rw-r-- 1 ruppena staff 4.6K Jan 22 11:45
    ↪ OnOffStateContextResourcePublisherClientResourceAPI.py
-rw-r--r-- 1 ruppena staff 4.1K Jan 22 11:45
    ↪ OnOffStateContextResourcePublisherResourceAPI.py
-rw-rw-r-- 1 ruppena staff 407B Oct 1 10:03 README.md
-rw-rw-r-- 1 ruppena staff 3.5K Oct 1 10:03 WebSocketSupport.py
-rw-rw-r-- 1 ruppena staff 1.4K Oct 1 10:03 ZeroconfigService.py
-rw-r--r-- 1 ruppena staff 36K Dec 12 11:13 clients.db
-rw-r--r-- 1 ruppena staff 2.5K Jan 16 16:40 publisher.py
-rw-rw-r-- 1 ruppena staff 236B Dec 12 11:28 requirements.txt
-rw-rw-r-- 1 ruppena staff 11K Jan 22 11:45 rest-documentation.html
-rwxrwxr-x 1 ruppena staff 6.1K Jan 22 11:45 rest-server.py*
-rwxrwxr-x 1 ruppena staff 71B Oct 1 10:03 setup.bat*
-rwxrwxr-x 1 ruppena staff 741B Oct 1 10:03 setup.sh*
drwxrwxr-x 5 ruppena staff 340B Dec 12 11:31 templates/
-rwxrwxr-x 1 ruppena staff 112K Oct 1 10:03 virtualenv.py*
```

List. 10.7: Generated smart light bulb skeleton

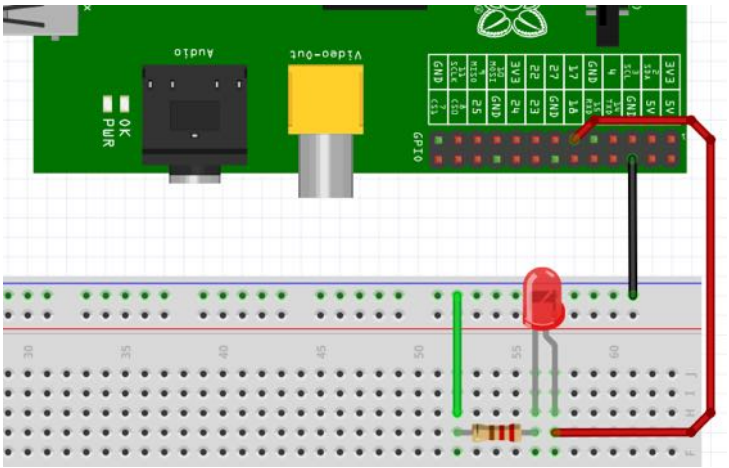


Fig. 10.4.: Electronic wiring of a smart light bulb

Finally, the gaps in the code need to be filled. In the case of a simple smart device, this is done rapidly. The publisher module and its associated module don't need any modification. Only the context resource needs to be adapted to return the state of the smart light bulb in an appropriate manner and to parse the input JSON data. For this, a new HTML template takes care of representing the smart light bulb as an HTML page. In this example, the JSON and XML message are quite simple and able to adopt the same format, which is also used for communication between the RESTful web service and the underlying hardware. Therefore, if a client requests a JSON representation, the server responds with a message like `{"light":"on"}`. The last step is to adapt the `Hardware_Monitor` module so that it can parse the incoming messages and modify the state of the light bulb if requested. Again, only minor changes are necessary to parse the incoming messages and translate them into commands for the hardware. Through the provided `setup.sh` script, the RESTful web service can easily be deployed to the hardware (for example a Raspberry Pi). Listing 10.8 shows how the final code is deployed and run. To keep the focus on the important parts, some messages about installing the required dependencies have been removed from the listing (line 10). After line 20, the smart device is ready and can serve incoming requests.

```
1  ruppena@tungdil:lb_Server$ ./setup.sh
2  be sure to install first
3  apt-get install libavahi-compat-libdnssd1
4  New python executable in xwot/bin/python
5  Installing setuptools.............................done.
6  Installing pip...........................done.
7  Collecting setuptools from https://pypi.python.org/packages/3.4/s/setuptools/
       ↪ setuptools-12.0.4-py2.py3-none-any.whl#md5=062ffc9b0b1b4b4ff1ade2fd1d6664f8
8    Using cached setuptools-12.0.4-py2.py3-none-any.whl
9  Installing collected packages: setuptools
10 [....]
11   Running setup.py install for pyserial
12     changing mode of build/scripts-2.7/miniterm.py from 644 to 755
13     changing mode of /Volumes/home/ruppena/Desktop/lb_Server/xwot/bin/miniterm.py to
           ↪ 755
14 Successfully installed autobahn-0.9.5
15 backports.ssl-match-hostname-3.4.0.2 certifi-14.5.14 ponydebugger
16 pybonjour-1.1.1 pyserial-2.7 requests-2.5.1 six-1.9.0 tornado-4.0.2
17 twisted-14.0.2 zope.interface-4.1.2
18 ruppena@tungdil:lb_Server$ xwot/bin/python rest-server.py -d ~/COM2 -s 1
19 INFO Peparing Serial Connection. Please stand by...
20 INFO Up and Running
```

List. 10.8: Deploy and run the smart light bulb

10.2.2. Smart Door

The previous use-case, although very basic, shows a full example of how to create a new smart device including the required hardware. This subsection examines a similar example. However, the hardware side and the virtual side in particular, will be slightly more complex. Whereas for the smart light bulb the virtual side was only made of a context resource which translated into one single REST resource, this time the virtual

side delivers a more complex URL hierarchy. This thesis has referred to a smart door several times (see for example Listing 5.9) without giving more detail of what such a smart door should look like. It seems quite natural that a door in the physical world can be opened or closed. This is also how we mostly use a door: we open it to enter a room and close it behind us. Furthermore, a door can be locked with a key and also unlocked. Obviously, before opening a door, it has to be unlocked and a door can only be locked if it has previously been closed. Once again, somebody can gather a door's open/closed state simply by looking at it. Figure 10.5 resumes this situation in a UML use-case diagram. Compared to the use-case diagram of the smart light bulb, this one has more elements and will therefore translate into a richer xWoT model, with more REST resources. Again, the model's left side reproduces the passive observation of the door's state. As previously, this passive observation translates into a publisher resource in the virtual world. This time however, the scenario needs two types of observable events: (1) when the door is opened or closed and (2) when the door is locked or unlocked. It makes sense to distinguish these two types of event to allow subscribers to listen only to one or the other. Moreover, if a subscriber is interested in both event types, he can simply subscribe to both. The bottom right of Figure 10.5 is exactly the same as before and simply shows the predefined xWoT subscription actions that are possible on any publisher resource. The model's right hand side contains the use-cases specific to this example. Each described action translates into a use-case. Furthermore, the *Get State* use-case allows a client to query the current state. Since the example contains two observable properties, two separate use-cases are necessary. However, by using inclusions in the UML model, we show that they have something in common. In terms of REST resources, this means that each use-case translates to one Resource reflecting either the open/close state or the locked/unlocked state. The base use-case still also gets translated into a Resource giving access to both statuses at once.

Based on this use-case diagram, we can now model the physical side of the smart door as an xWoT model. The topmost element of the model, the Entity, is the door itself. It is composed of an actuator responsible for opening and closing the door and a sensor measuring whether the door is opened and to what degree. Additionally, the door needs an actuator to (un)lock it and a sensor returning whether the door is (un)locked. The first pair of actuator and sensor form one device and the second pair of actuator and sensor form a second device. Therefore, the physical side of the Entity is made of two Devices. This physical model can be turned into an augmented model where both the physical and the virtual side are modeled. With the help of the `physical2virtualEntities` tool, a basic structure for the virtual side can be constructed.

Figure 10.6 shows the situation after the execution of this tool. Each device of the physical side is translated into a resource on the virtual side, as are their child devices. One could argue that both actuators should translate into a Context Resource. However, the locked property depends on the closed property and vice versa. Therefore, the dependency between the actuator and its sensor is not strong enough to combine them into a Context Resource, although this is a matter of personal taste. Even though the tool makes an educated guess about how the physical side should be translated into the virtual one, it sometimes fails or the translation is incomplete, as in this case. Yet, this is not the tool's fault, nor the model's. The `physical2virtualEntities` tool correctly translates the physical side into an adequate virtual side. However, according to the use-case of Figure 10.5 there is one use-case where it is possible to query the overall state of the

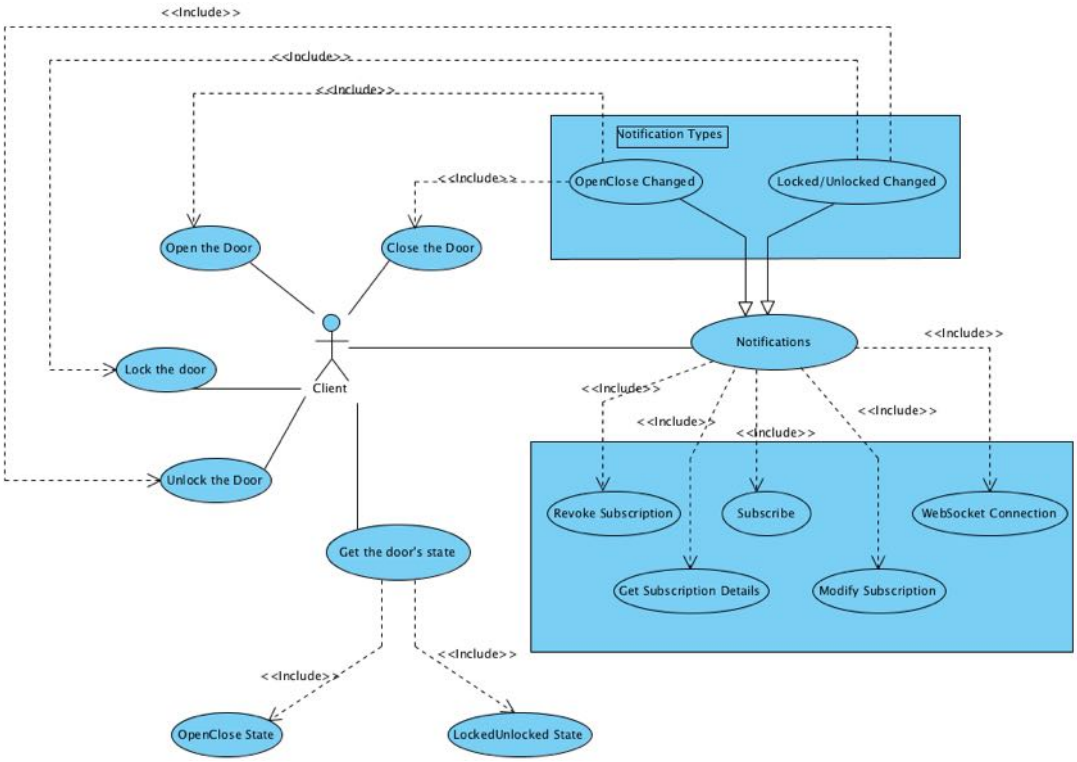


Fig. 10.5.: Use-Case diagram of a Smart Door

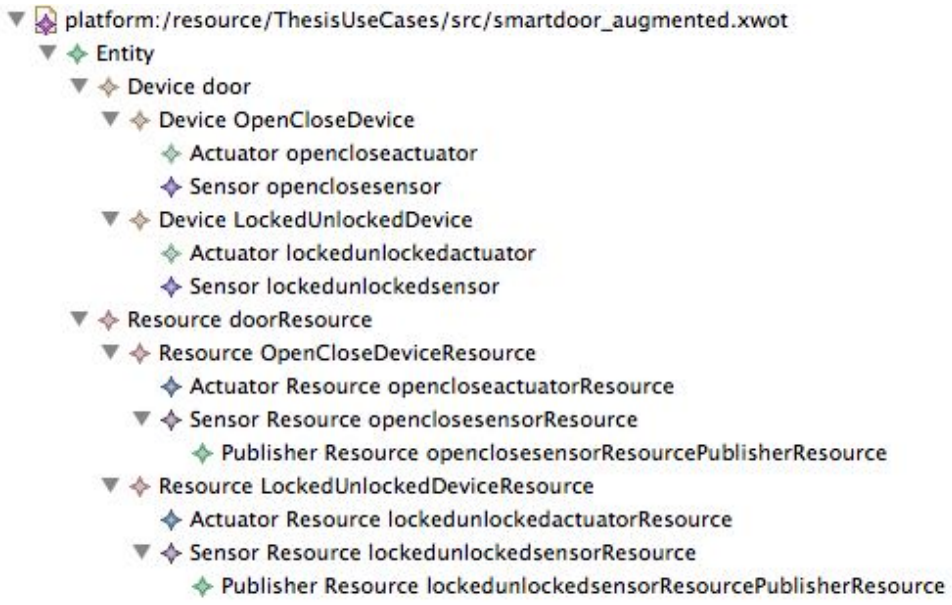


Fig. 10.6.: Automatic translation of the physical to the virtual side

system with two subsequent use-cases to query each state individually. Although the present translation allows querying each state individually, it is impossible to query the global state of the system. Therefore, it is necessary to tweak this initial Virtual Entity until it functions as required. Figure 10.7 shows the final result of these efforts. Instead of two resources, the virtual side is now made of three, where the third resource makes the global and individual door states available. This resource contains two children, each representing one state. This leads to a situation where two URLs point to the same resource. Nonetheless, according to Richardson [B18] this is not forbidden, but care is required later when filling the gaps in the code, that one of the two resources is defined as the canonical one (like the one for the device), while the other as a symlink (via the redirect mechanism) to the former. Additionally, Figure 10.6 shows that the door is a device with two child devices. Although physically there are two separate devices, they act like one. It is unlikely that any other scenario relies on only one or the other of the two devices without relying on the door itself. Therefore, the `composed` property of the `door` device is set to false to indicate that this is a logical grouping of physically separated devices. As a result, the model compiler will produce one RESTful web service for the door encompassing the open/close and the locked/unlocked mechanism.

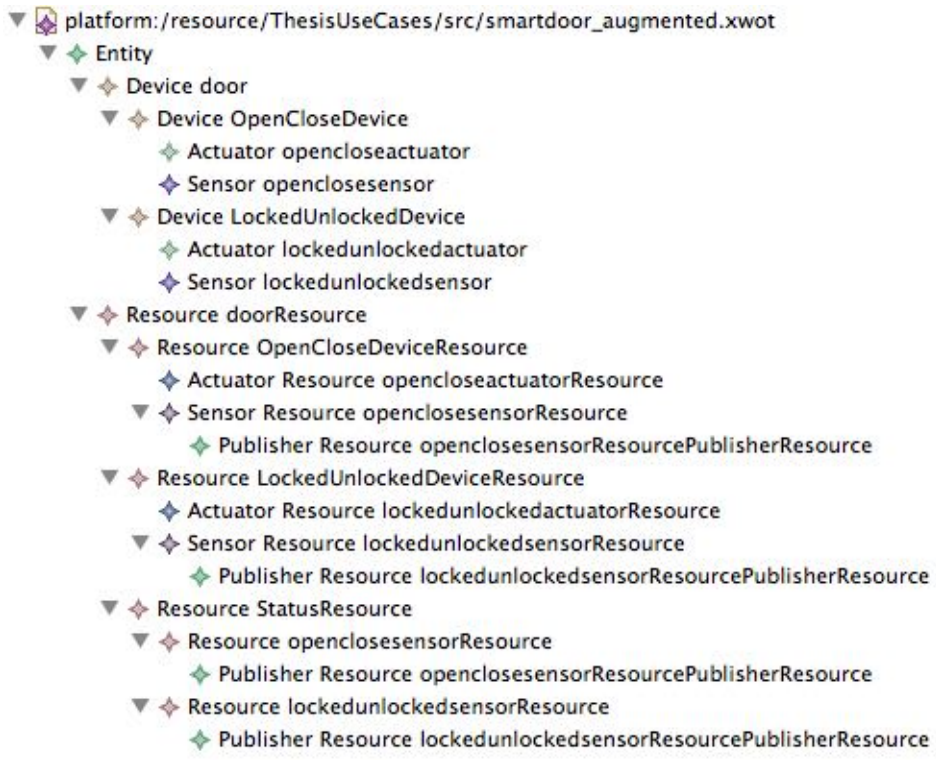


Fig. 10.7.: Tweaking of the generated virtual side

```
ruppena@tungdil:src$ ll REST-Servers/
total 0
drwxrwxr-x 3 ruppena staff 748B Jan 23 15:04 _door_Server/
```

List. 10.9: Generated code skeletons for the smart door

At this stage, the model perfectly fits the use-case and is ready to be converted into code skeletons via the `model2Python` command line tool. Listing 10.9 shows the generated code skeletons. As for the previous use-case one single service skeleton is generated,

`_door_Server`, representing the smart door and containing the enumerated resources of Figure 10.7. Once the code gaps are filled, the service will be deployed directly on the door, thus making it available to the virtual world.

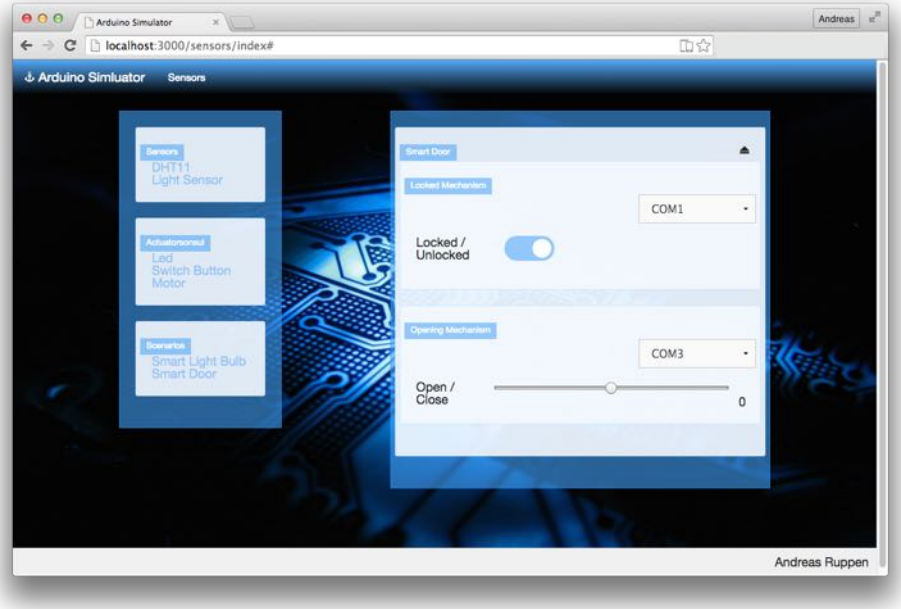


Fig. 10.8.: Hardware mock for the smart door

Building the electronic circuit for a real smart door is outside the scope of this thesis. Therefore, the hardware mock application is used to simulate how the hardware would react. Figure 10.8 shows the hardware mock-up of the smart door. In contrast with previous mock-ups, this one is composed of two mock devices, one running on the virtual serial port `COM1` and the other on `COM3`. Through this separation, they can be used as separate devices and it is possible to talk to each one separately. This also reflects how the physical side is built, with two disjointed physical devices.

10.2.3. Smart Door Revisited

How a single door can be augmented and represented in the virtual world was discussed earlier. In the process, a device with a bunch of sensors and actuators is attached to the door. This device delivers information to and accepts command from a RESTful web service automatically generated from the virtual entity description in the corresponding xWoT model. The compilation of this model leads to one single service to be deployed directly on the device attached to a door. The last step of the approach discussed on page 182 can be repeated for several doors resulting in a number of smart doors. Once the coupling of hardware and xWoT service is deployed to these doors, each has its own representation in the virtual world and therefore can be manipulated either in the physical or the virtual world.

If this vision of several smart doors in a building is pushed a little further, we can try to model more than just a bare door. Modeling a full building would require too much space to discuss. Therefore, instead of a full building, we limit the present use-case to a building composed of several floors, each having several doors and several lights. The

use-case diagram of this example is a combination of the use-case diagram of the smart light bulb and the use-case of the smart door. Additionally, floors group the doors and lights and the building is composed of several floors, so it is possible to know and modify the state of a whole floor or building. Furthermore, it is possible to ask a given floor which doors are open and it is also possible to open/close all the doors on the same floor at once. Of course the same is true for the whole building. Figure 10.9 resumes this train of thought visually.

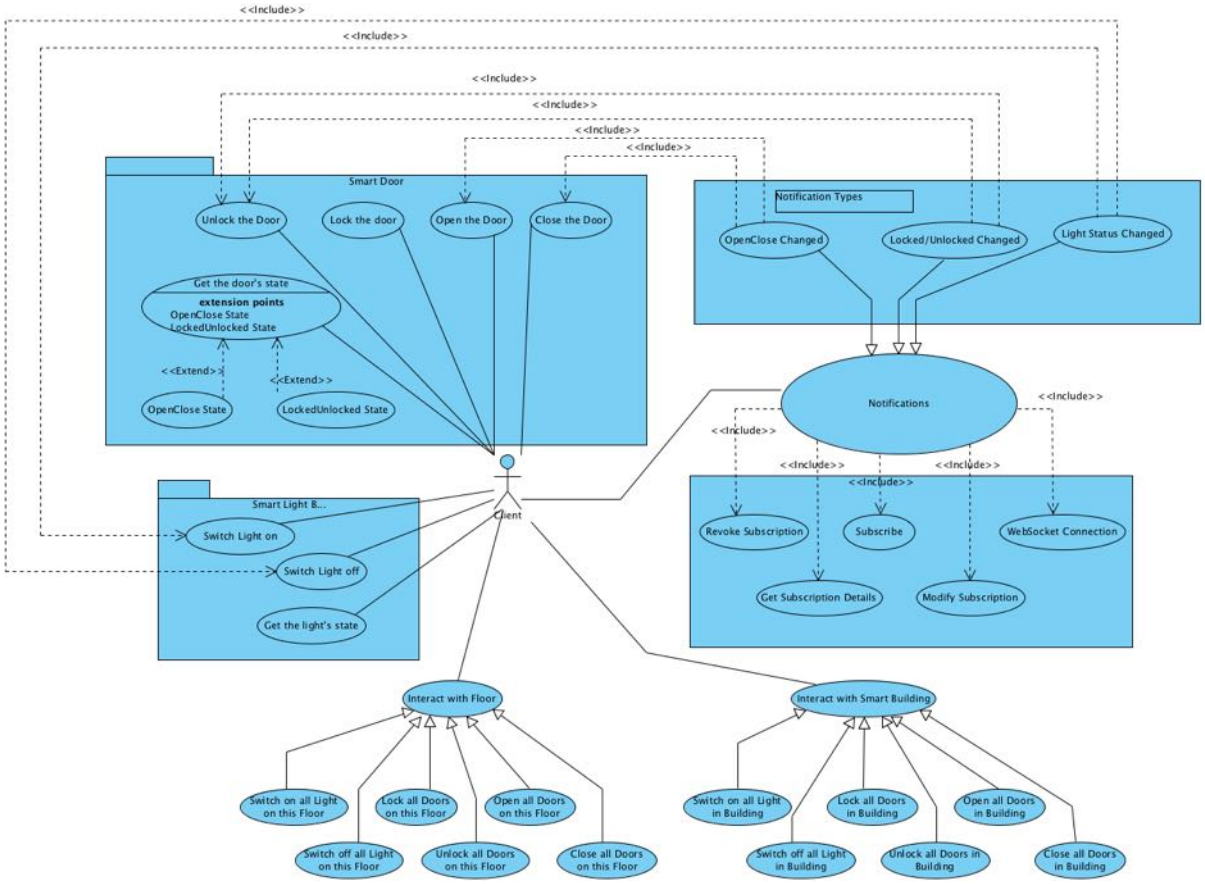


Fig. 10.9.: Use-Case diagram of a smart building

As before, this use-case diagram guides the modeling of the physical side of the smart floor. It is interesting to note that the sensors and actuators are only attached to the doors but none are attached to the floors. Whereas the physical entity for a door is exactly the same as before, each door is now attached to a floor and these are grouped together as a building. Both the floor and the building are represented as devices. However, these devices are now used to physically group the different doors together into floors and then into a building. Therefore, the `floor` as well as the `smartBuilding` both have their `composed` flag set to true, whereas the `door` and the `light bulb` keep this flag set to false to indicate a logical grouping of devices. The top part of Figure 10.10 shows the physical model of such a smart building. There is the `building` itself, the `floor` as well as the `smart door` and `smart light bulb` with the hierarchy as depicted in Figure 10.10. Unfortunately, there are no model `floors`, `doors` and `light bulbs` as they don't appear naturally in the physical world. As will bee seen later, this decision has an influences on the generated virtual entity, which needs some work in order to become a meaningful RESTful API.

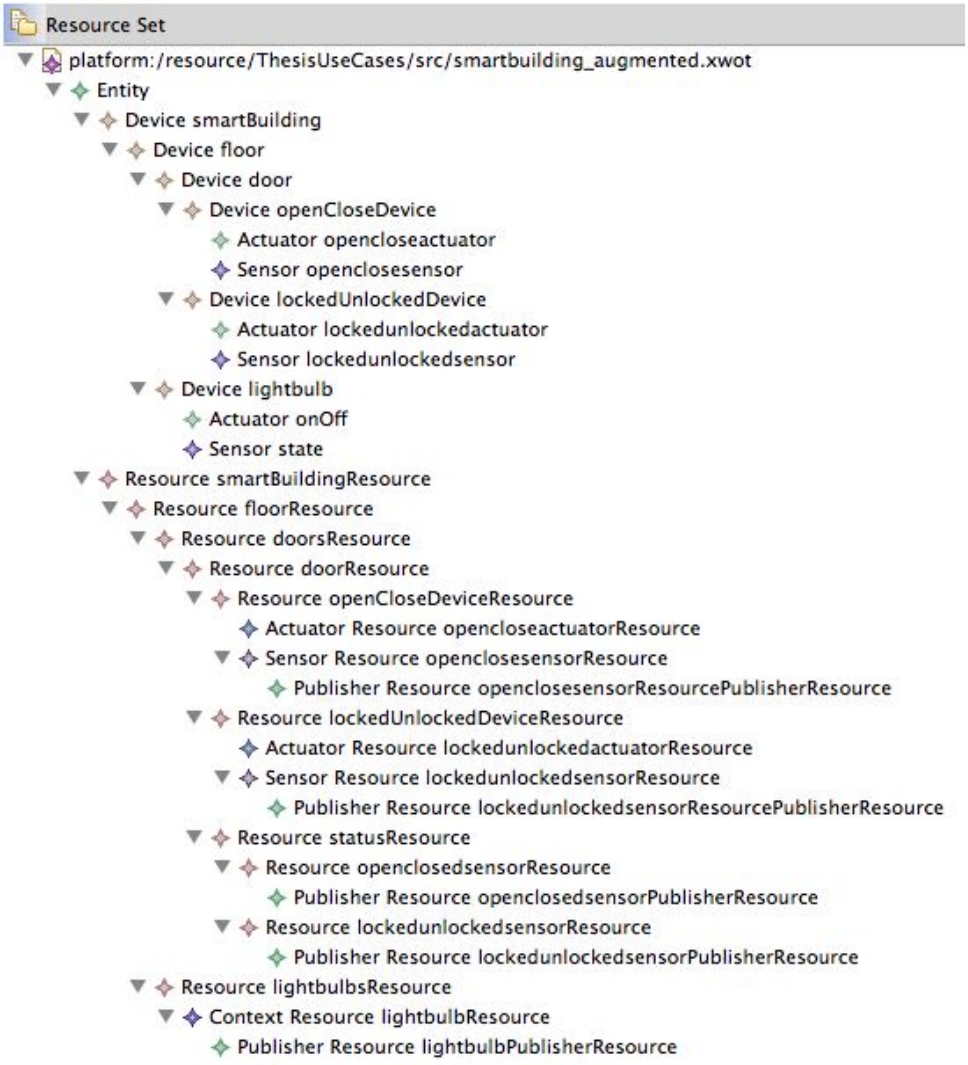


Fig. 10.10.: xWoT model of a smart building

If the physical model is translated via the `physical2virtualEntities` tool, the result is not convincing. First, the virtual door lacks the `statusResource`. Since this is a purely virtual resource, it needs to be added by hand to the generated model. This issue has already been discussed for the smart door use-case and the same approach ca be used again. Second, the generated resources hierarchy exactly reflects the physical hierarchy. This leads to URIs like `http://example.com/sb/5/3` where the first integer represents the floor identified by the number 5 and the second integer represents either the smart door with id `3` or the smart light with this id. Already this reflection shows that the generated structure lacks consistency. Besides, in RESTful services, when dealing with a group of objects, sometimes it is necessary to introduce an aggregation layer. Therefore, instead of attaching the `doorResource` and the `lightbulbResource` directly to the `floorResource`, two aggregation resources are created: `doorsResource` and `lightbulbsResource`. The former aggregates all doors in a given floor whereas the latter aggregates the light bulbs in the same floor. Normally, such an aggregation layer would be necessary to group all the floors together. However, since the floors are the only child resource of the smart building this is not necessary and would just bloat the API. However, if there are other resources at the floor level, such an aggregation resource becomes mandatory. Consequently, the new URIs look like `http://example.com/sb/5/doors/3` to access, for example, the door with id 3 on the floor with id 5 and `http://example.com/sb/7/lights/8` to access the light with id 8 on floor with id 7. Not only does this aggregation level make the API more readable, it also prevents conflicting ids between smart doors and smart lights.

Now that the virtual model is fine-tuned it can be turned into code skeletons. As before, the `model2Python` model compiler takes care of this. Since some of the resources have their `composed` flag set to `true`, more than one service skeleton will be generated. This is also as expected: one skeleton for the smart door, another for the smart light and finally, the last for the smart building. Listing 10.10 shows the different service skeletons generated by the model compiler. The `__int:doorid__Server` and the `__int:lbid__Server` service skeletons implement the smart door and the smart light respectively. These two services are deployed on the corresponding hardware. This combination of hardware and software is copied so that each door and each light gets its own hardware and software. These services (and also the necessary hardware) is the very same as discussed in previous use-cases. Suppose that somebody has already implemented these use-cases. Then smart doors and smart lights are already deployed. Although, they were foreseen for a different use-case, the xWoT meta-model ensures they look the same as the skeletons generated in this use-case. Therefore, instead of reimplementing these services and deploying a second piece of hardware on each door and light bulb, these smart devices can be reused. The `NM-_sb_Server` is the node manager service and represents the smart building use-case. It represents the Entity and therefore contains the overall resource hierarchy as defined in the model. However, most of its resources are not canonical but refer to other resources available on one of the other generated services.

Listing 10.10 also contains three other node manager services: one representing a single floor plus two others aggregating, respectively, the smart doors and smart lights. Since these are purely virtual resources, their functionality can also be directly implemented in the `NM-_sb_Server`. The choice is up to the developer whether the canonical URIs are on the `NM-__int:floorid__Server`, the `NM-_doors_Server` and the `NM-_lbs_Server` or on the `NM-_sb_Server`. However, since all these services need to be deployed on some extra server (they dont have any associated hardware) it makes most sense to only

```
ruppena@tungdil:src$ ll REST-Servers/
total 0
drwxrwxr-x 3 ruppena staff 1.0K Jan 29 11:03 NM-__int:floorid__Server/
drwxrwxr-x 3 ruppena staff 884B Jan 29 11:03 NM-_doors_Server/
drwxrwxr-x 3 ruppena staff 510B Jan 29 11:03 NM-_lbs_Server/
drwxrwxr-x 3 ruppena staff 1.0K Jan 29 11:03 NM-_sb_Server/
drwxrwxr-x 3 ruppena staff 1.1K Jan 29 11:03 __int:doorid__Server/
drwxrwxr-x 3 ruppena staff 612B Jan 29 11:03 __int:lbid__Server/
```

List. 10.10: Generated Code Skeletons for the smart building use-case

implement the `NM-_sb_Server` and skip the three other services.

Therefore, supposing that the smart lights and smart doors already exist, only one RESTful web service needs to be implemented to complete this scenario. This shows yet another aspect of the xWoT meta-model where the model compiler tries to produce the most valuable xWoT components not only for the current use-case, but also for future use-cases which might rely on some of these components. This not only greatly speeds up development but also ensures that for one door, there is exactly one virtual representation and not a dozen different ones for each use-case. Again, this vision is inspired by the physical world where physical components serve a goal but also can be used for other means.

10.2.4. Medical Records

The previous examples showcased various combinations of sensors and actuators and how they translate into a physical model, and finally, into executable code. Furthermore, the discussion about how the global system state translates into the physical world (see Subsection 10.2.2) nicely illustrates how the virtual side of a smart device can offer more than what is possible in the physical world. In this example this approach is pushed even further by completely eliminating the physical side. It seems that sometimes it can be difficult to grasp the physical manifestation, for several reasons: for example, if the entity is moving through sensors installed at fixed locations, it is difficult to model the physical side through a combination of sensors and actuators since these are not attached to the entity itself. Moreover, if the physical manifestation is a *phenomenon* which can be measured only by observation but not by attaching devices in a permanent manner, modeling the physical side through a combination of sensors and actuators is not accurate either. This is for example the case when the Entity is a person. First, depending on the context, a person can be an individual; it can be a client, a customer, a patient, a caregiver etc. Depending on the context, different information about him are of interest and some cannot be measured. A first name for example or a gender is an attribute of a person. There is no sensor that can be attached to a person to *measure* his name. For other properties, a person behaves like a moving entity through fixed sensors. This is, for example, what happens to get the height, the weight or the blood pressure of a person. Although this is a measurable quantity, no person comes packaged with a height or weight sensor. Instead, some external instruments are used to measure the person's height etc. This example further investigates such a scenario where the Entity cannot be augmented with sensors and actuators.

Regarding the huge quantity of fabricants producing various fitness trackers (eg. Fitbit, Nike Fuel etc.) e-Health and e-Nursing is a trending topic and all major vendors launch their products to track the user's health [WEB52, WEB4, WEB19]. All these data need to be stored somewhere. Commonly, in a hospital environment, this is done in medical records. Historically, this is a paper file containing sheets with personal data, analyses, consultations etc. More and more hospitals and also caregivers are switching to electronic systems. Imagine now that this medical record is an xWoT resource. It can participate in mashup applications either as data provider or data receiver and push the barrier to e-Nursing one step further. Section 10.3 will investigate a larger use-case involving the medical record discussed herein.

In contrast to the previous examples, the approach is slightly different this time. The medical records, although having a physical representation, are a purely virtual good. The physical and virtual side can be linked together through a tag (NFC, QR, RFID, barcode etc.), but a modification of the physical side will have no effect on the virtual side and vice versa. Since the two are disconnected, the model has only a virtual side. Therefore, only the steps dealing with the virtual side remain. Also the `physical2virtualEntities` tool is useless in this example. Even though its application would not harm, its outcome would be empty. Therefore, the modeling process directly starts with the virtual side of the entity. Since there is no physical anchoring, there are no guidelines on how to model the virtual entity. It is up to the architect to choose an appropriate design. Therefore, during modeling he must already think about the resulting hierarchy and if it makes sense to represent the entity this way. Also, since he is free of any physical constraints, the architect should take into consideration the usability of the resulting hierarchy. This sometimes involves digital information having a different hierarchy from its physical counterpart (sheets of paper in this case). For example, whereas in the physical file, all records are stored in temporal order, the virtual side can offer more types of ordering. Another point where use-cases based on a physical device diverge from purely virtual use-cases is the modeling approach. Whereas for the former, the physical device imposes its structure to the virtual side, the latter merely asks to model directly the final API. In this example, this means that the content of such a medical records needs to be defined. Instead of starting with a blank sheet, what the physical file based medical records look like and the type of information they contain has to be considered. Five types of information can be identified:

1. General information about the patient like his name and gender but also information about insurance and family members or emergency contacts.
2. Data about the patient's lifestyle. This encompasses information like the height and weight but also factors directly influencing the patient's health like his drinking behavior, if he smokes or takes drugs.
3. Medications need to be tracked to ensure that there are no nasty side effects between two prescribed drugs. Having the medication grouped together greatly speeds up this process. The medication not only contains information about what drugs are prescribed, but also the start and end date as well as the quantity.
4. Analyses are part of the daily business of any hospital or any medical practice. They range from a simple blood analysis to determine the blood group before an operation to long term tracking of the heart rate for example. Storing them all together can save money and time since it is easy to find past analyses.

5. A short résumé for each consultation. This is the place where the caregiver can make notes about what he did or what he plans to do in future consultations. Since the xWoT supports the property of connectedness, a consultation contains links to medication if the caregiver prescribes drugs and links to analyses if they are needed for future consultations.

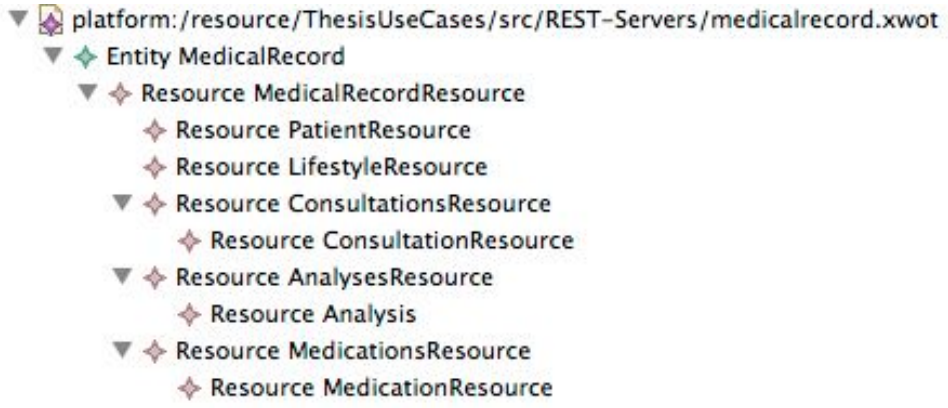


Fig. 10.11.: Translation of the requirements into resources

These requirements also dictate the available resources and how they are modeled. Figure 10.11 shows the results of these efforts. Each of the five types of information gives birth to a resource. Since a patient can have several consultations, analyses and medications, these resources are all composed ones, the top level resource giving access to a list of consultations (for example) whereas the child resources give access to individual consultations. The model compiler allows a variable path definition and correctly translates it into code. The ConsultationsResource, for example, is associated with the `consultations` URI, whereas the ConsultationResource is associated with the `{id}` URI. The special syntax with a parameter name between curly brackets indicate that the URI is not a string but a variable and should be treated as such by the compiler. The last decision is whether the hierarchy of Figure 10.11 should translate into one RESTful web service or instead, whether each of the five types of information gives birth to an individual web service with one node manager service representing the use-case. Both approaches are doable. For the sake of this example, let the `composed` properties be `false` and generate a single RESTful web service. The outcome of the `model2Python` compiler is a single RESTful web service containing the modeled resources and their hierarchy. The result of this process is shown in Figure 10.12 containing on the left the structure of the generated code skeleton and on the right the `ConsultationsResource`. This resource also defines the URL schema of its child resources, where on line 90, the compiler has translated the `{id}` URI into a numeric variable and delegates requests to it to the `ConsultationResource` module.

At this stage, the code gaps can be filled with the required business logic and then the service is ready and can be deployed. Implementing the business logic is also slightly different from the previous examples; instead of dealing with hardware and low level programming, purely virtual resources deal with information available in a database or file system. Since they can represent anything, they can contain quite large business logic. In this example, the business logic is reduced to some SQL queries to fetch the requested information from some database or Customer Relationship Management (CRM).

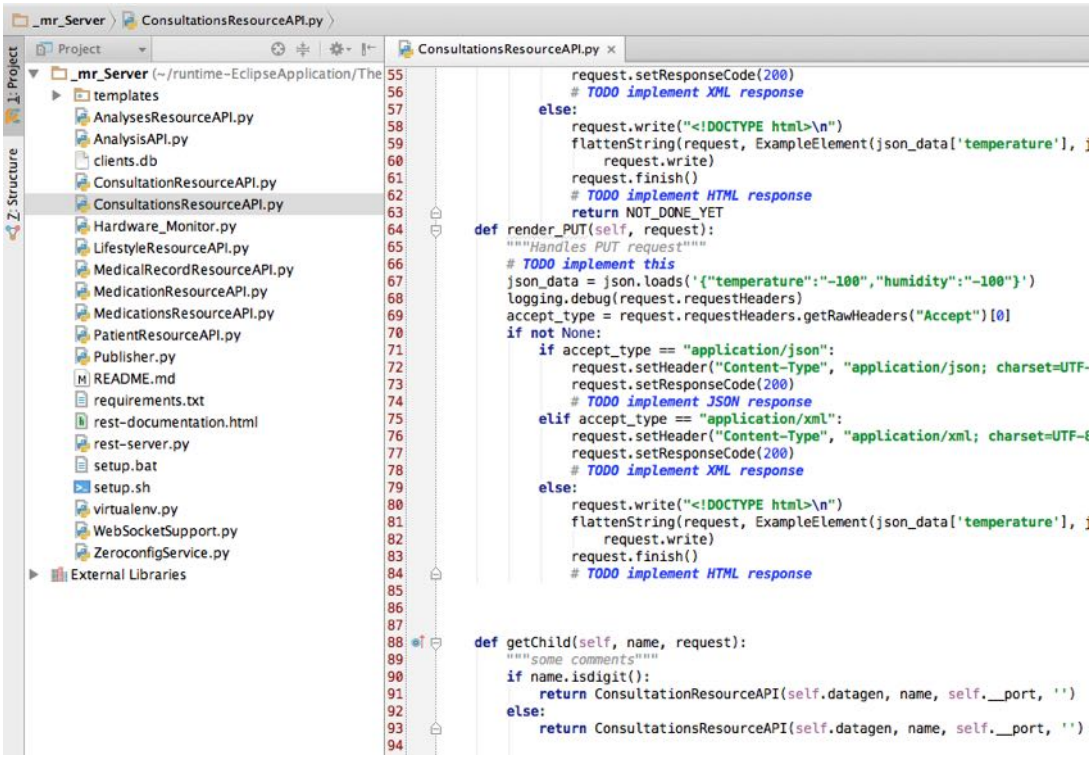


Fig. 10.12.: Generated REST web service

As such, the `ConsultationResource`, for example, executes a query in the database, with the parameters of the `WHERE` statement adapted to the right patient and the selected consultation. Still, the input and output format need to be defined, likely with and XSD schema as for the previous examples. Figure 10.13 depicts how a patient is represented. It contains the attributes discussed during the requirement-engineering phase plus other not discussed any further now. Although XSD is only used to define valid XML files, they also define JSON representations since XML can easily be mapped to JSON. Since the defined attributes are judged to be important, they should also appear in the HTML representation. Therefore, the XSD file also influences the HTML output. More specialized representations however need special engineering. The `ConsultationsResource`, for example, could offer an `ical` output. Of course, such a representation needs special care.

Another use-case that would fit this example is the parcel tracking and delivery problem we discussed in [71]. In this case the parcels are the entity of interest. As a parcel travels, it visits different scanners, which are the devices in this scenario measuring the parcel. However, the devices are never attached to the parcels; instead they are attached at a fixed places in space. A parcel is only identified through a *tag*. An use-case where such sensors form the hardware of a service and the tags serve as input parameters for the sensors is illustrated in the next section.

10.3. A Bigger Example — eHealth

The previous section discussed various small examples highlighting different aspects of the xWoT modeling environment. The first example showcased a simple combination of one sensor and one actuator forming a context. The second example showed how the

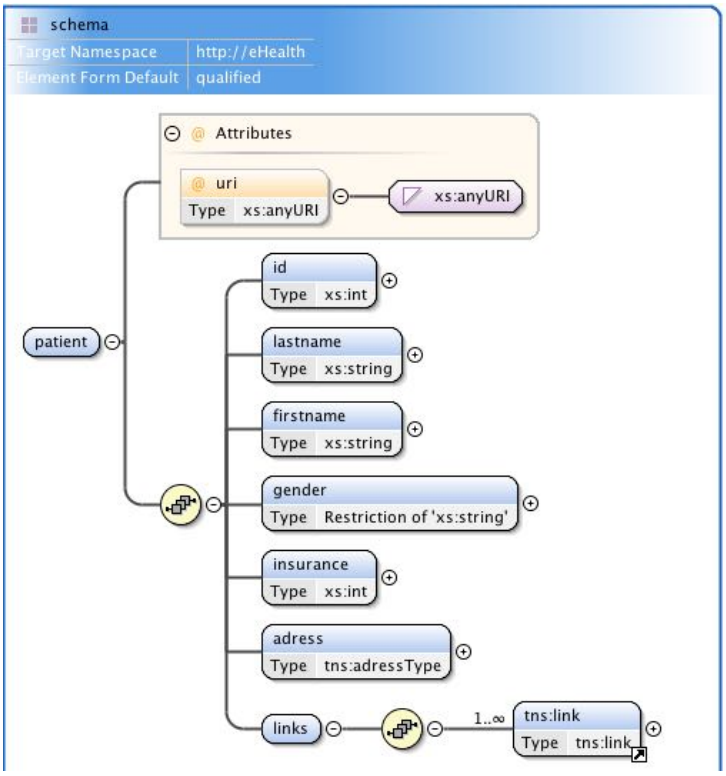


Fig. 10.13.: XSD file defining the XML and JSON representations for the Patient Resource

generated model could be enhanced with additional virtual resources, making the virtual side richer than the physical. In a combination of these two use-cases, we discussed a situation that needs a node manager to represent the entity and showed that in these situations, the model compiler generates several independent service skeletons running either on dedicated hardware or some generic server. Finally, the last example introduced a model containing only virtual resources and explained how the xWoT meta-model handles entities lacking any physical device. This section concentrates on a bigger use-case combining the previously considered aspects and approaches to model a full environment. Some entities are similar to the smart light bulb example or contain enhanced virtual representation. Others need a node manager to represent the entity and break down into independent deployable xWoT components, whereas some aspects require a mashup composed of other xWoT components.

Regarding the xWoT modeling approach depicted in Figure 9.10 and re-examined in Section 10.2, not all the steps will be carried out. Many of the involved devices would require a deep knowledge of how to build them and need very specialized sensors and actuators. The focus of this thesis is the meta-model and its associated tools and not an introduction to electronics. Thus, the assumption is that these devices implement some low level connection (like a serial interface) to which it is possible to connect the generated RESTful web services. Therefore, this section focuses on modeling the use-case and translating it into code skeletons. This involves many different resources, some of them being purely virtual, others simple sensors or a combination of sensors and actuators.

The most notable difference compared to the other use-cases examined is the number of actors involved. Whereas previously, there was only one client, the consumer of the

smart device, here there are *patients*, *caregivers*, *visitors*, *administrative staff*, *cleaners* and many more. All together, they ensure that the hospital is running, that somebody is taking care of the patients, their surgeries and postoperative care. Although it is possible to model all this in one big use-case, this would miss the essential parts. Therefore, in this use-case the focus is on *patients* and what they are doing during a stay at the hospital. Depending on the situation (e.g. emergency, pregnancy, surgery) the process may vary. Yet without loss of generality, it is true that a patient enters the hospital at some point, gets treatment according to his condition and returns home as soon as he is better. To keep the example manageable, we will focus on a standard stay at a hospital either for examination or a surgery. Having a closer look at this flow, a more fine-grained process can be identified. Upon arriving at the hospital, the patient needs to check in like in a hotel. If it is his first stay, the *secretary* creates a new *medical record* and fill in the general information. At this stage, the patient also gets a badge identifying him. This can for example take the form of a wristband with his name or containing a NFC tag linked to his medical record. After admission, the patient is transferred to his room. When the physician visits, he completes the missing information in the medical record. This mainly concerns the patient's *life style factors*: smoking, drinking, drugs, height and weight as well as *medical pre-conditions*. To ensure the best possible care, often *analyses* are needed. They range from simple blood type determination to more complicated analyses. Figure 10.14 shows an extract from the form used to execute different analyses and gives an insight into the quantity of possible analyses.

Depending on his condition, a patient often needs some *drugs* to help during the convalescence. Again, the type and quantity of drugs a patient needs can vary and depend on his medical condition. It is important to avoid (1) overdoses and (2) unwanted side effects. This is only possible if the global medication state for a patient is known. Furthermore, drugs are prescribed for a given amount of time and need to be taken at precise intervals (only evening, before meal etc.) and in a given quantity. This information is just as important and needs to be stored together with the prescribed drugs.

From an architectural point of view, *caregivers*, *patients* and *drugs* behave the same way. Although they deliver information, like a sensor, they are not connected to the hardware delivering this information to a RESTful web service. Instead this information needs to be entered and stored somewhere so that later, a RESTful web service can expose this information as a resource. Yet, *caregivers*, *patients* and *drugs* are not completely disconnected from their virtual representations. Through tags, they are linked to their virtual counterpart. Other elements though can be turned into real smart devices. For example, the machine executing different analyses on the patient's blood can be turned into a smart object by hooking up a REST service directly on the machine. Also a heart rate monitor can be rendered smart simply by taking the reading of the sensor and making it available over a RESTful web service. Basically, these machines are sensors (and actuators), which are either directly or indirectly attached to a patient to observe a given property. As such, they can be turned into smart devices embedded in the xWoT if they offer the sensor readings not only on a screen but also over a RESTful API. In this sense, this scenario is similar to the parcels and scanners discussed at the end of Subsection 10.2.4.

The similarity between the parcels example and this use-case also becomes clear from the use-case diagram in Figure 10.15. All use-cases include the *patient* as the central element (although there are other processes in a hospital like cleaning, which has no

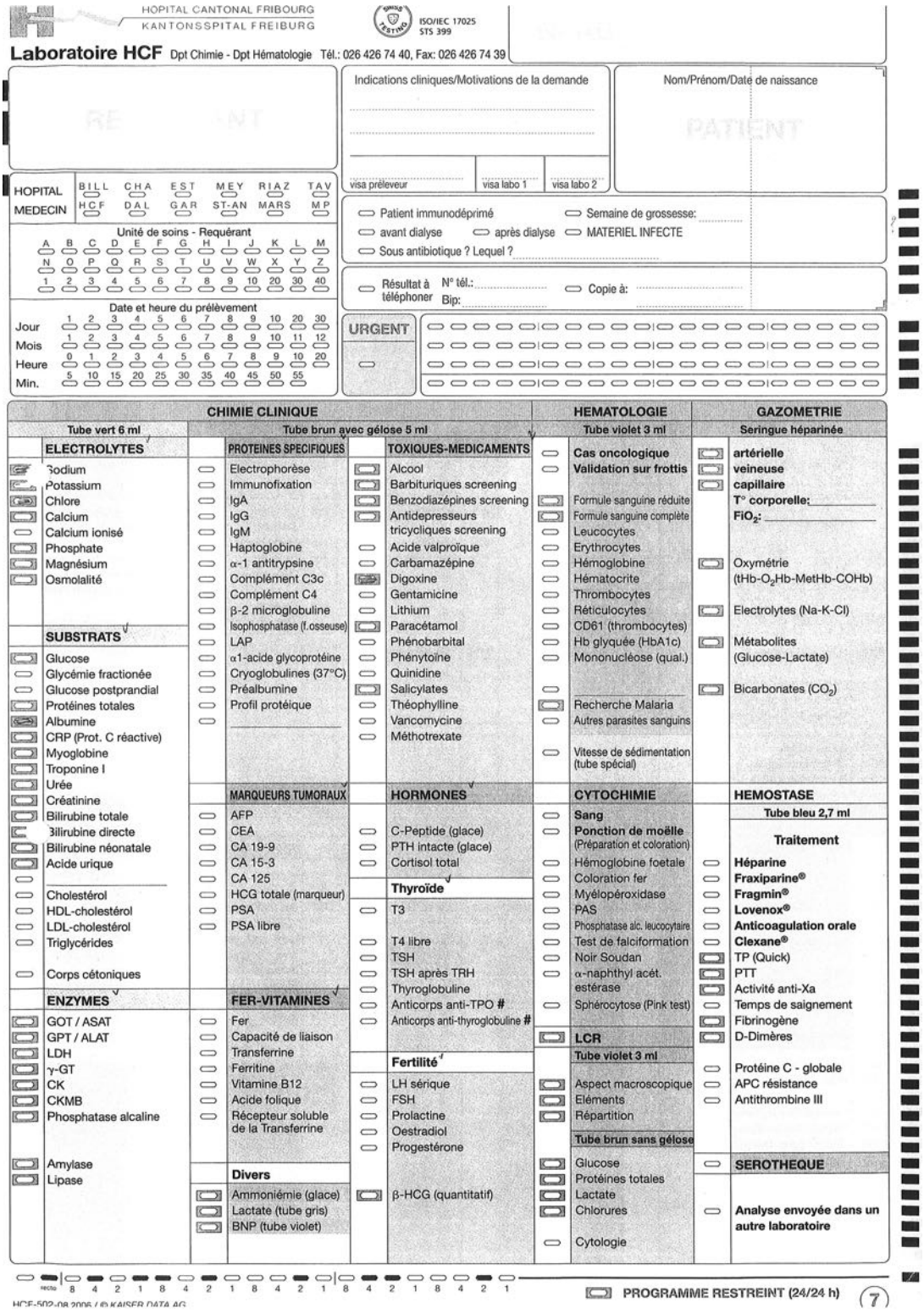


Fig. 10.14.: Analysis Sheet

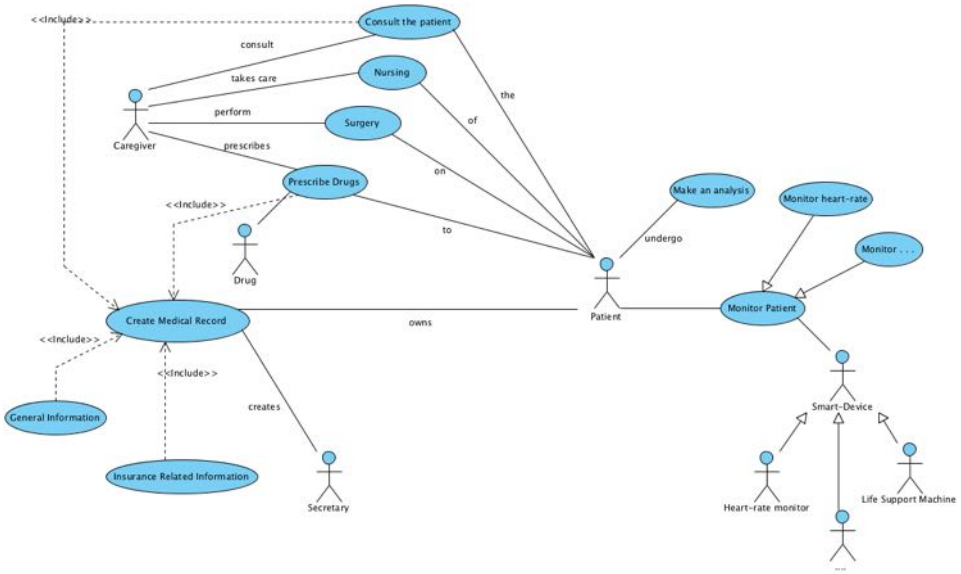


Fig. 10.15.: Overview of the eHealth use-case

connection to patients). Either, they measure something on the patient like the heart rate machine or they act on the patient like the physician during surgery. Unlike for the previous example where each use-case was self-contained, here some use-cases require some mashup layer exploiting the underlying xWoT elements. The monitor heart rate use-case on the bottom right of Figure 10.15 only makes sense if the heart rate sensor can report its values somewhere. The same applies to the analyses and consultations. Therefore, this use-case on the one hand implies the modeling and creation of several xWoT components and on the other hand it needs a mashup layer implementing the business processes taking advantage of these components.

10.3.1. Physical Devices

As an example of medical machines involved in this use-case we will analyze a life support system helping the patient to breathe (mechanical ventilation) and controlling his heart rate and pressure through an Automated external defibrillator (AED). The device needs at least one sensor to capture the patient's breathing and an actuator to help his breathing if required. Additionally, the device needs sensors for monitoring the heart rate and blood pressure and also a defibrillator in case of emergency. Consequently, we can conclude that the life support machine is made of two devices, one responsible for the breathing and the second responsible for the heart, each having its own sensors and actuators as discussed. The top part of Figure 10.16 shows how this description translates into an xWoT physical model. The last question is whether the `lifesupporsystem` device has the `composed` flag set to true or not. Setting it to true would later lead to individual service skeletons for each device plus one node manager service skeleton for the `lifesupporsystem`. However, since physically this is one big machine, it is better to set this property to false and only generate one service representing the life support machine as a whole.

Once the physical side is modeled, the virtual side can automatically be generated with the `physical2virtualEntities` script. Figure 10.16 already contains the result of this operation. Since the life support machine is relatively simple regarding its devices, there

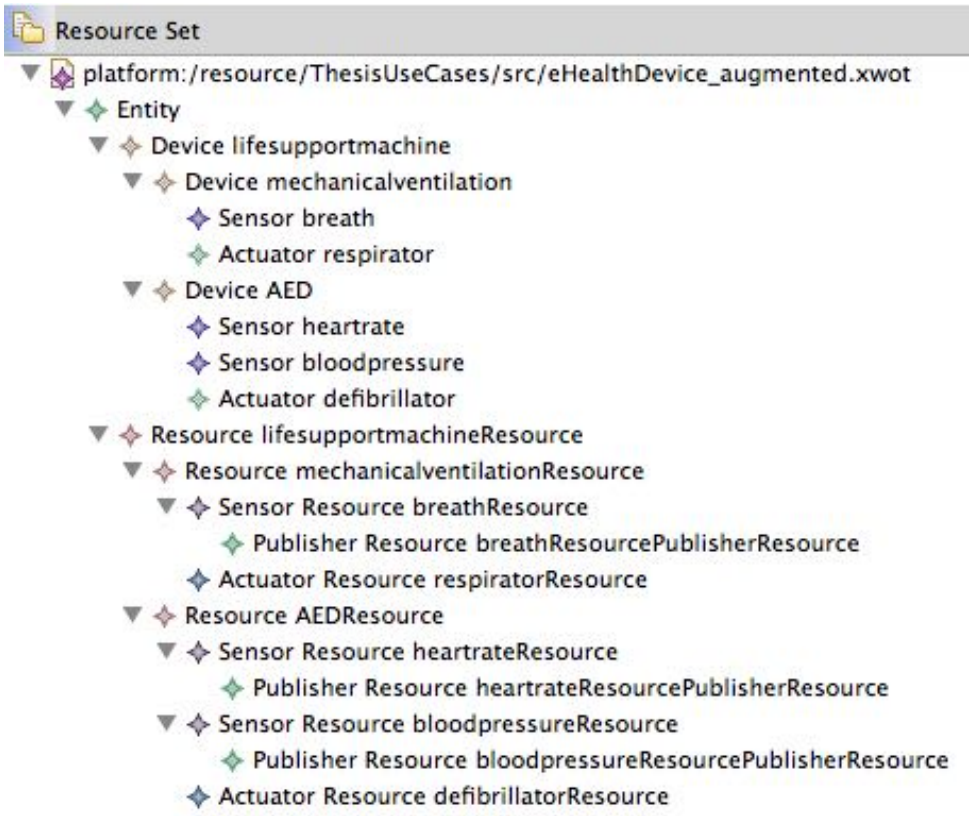


Fig. 10.16.: Physical and virtual model of a life support machine

is no need to change or add anything. The default translation from the physical to the virtual word is fine. Figure 10.16 shows that all the sensors also contain a publisher resource. The previous example discussed how, when using a publisher, a client is more interested in the fact that a change has taken place than the new value. This is an example where this assumption is false. The publisher can still be used to detect a change (a cardiac arrest) but mashup applications may be more interested in getting a feed of measures that can be saved for later consultation.

The model is now ready to be turned into code skeletons via the `model2Python` compiler. As shown in Listing 10.11, the compiler correctly translates the model into one unique xWoT service. The generated code contains the usual gaps that need to be adapted to the current hardware. This involves some coding in the `Hardware_Monitor` module, which is vendor specific. Additionally, the inputs and outputs need to be defined. Measures can generally be characterized by four attributes: (1) the measured value, (2) the scale, (3) the precision of the sensor and (4) a timestamp. These attributes completely describe most sensor readings, thus an adapted output format for a sensor contains all this information. Since XML allows elements, values and attributes we can put the scale, precision and timestamp can be put into attributes and the element's value used for the sensor reading. The name of the element is in its simplest form the name of the sensor. Such a representation seamlessly translates into JSON (although it does not differentiate between attributes and element value) and HTML. Actuators are more difficult to describe since their input depends on the actuator. A switch for example only needs a binary value whereas a brushless motor accepts degrees. However, the format should be kept as simple as possible. To keep things simple, the different publisher

resource use the same format as their corresponding resource.

```
ruppena@tungdil:src$ model2Python -i eHealthDevice_augmented.xwot
INFO - Start processing
INFO - Creating Server: REST-Servers/_lsm_Server
ruppena@tungdil:src$ ls -l REST-Servers/
total 0
drwxrwxr-x 3 ruppena staff 986B Jan 27 20:57 _lsm_Server/
```

List. 10.11: Compile the Life Support Machine into code skeletons

The approach discussed in this subsection can be used for any device in this scenario. Whether it is a smart thermometer or a life support machine does not matter. The end result is an environment where all machines of interest are equipped with a RESTful API and thus, seamlessly embedded in the xWoT. On their own, these smart eHealth devices have no big value. Nevertheless, it is easy to imagine a scenario where instead of printing the measured values directly on the attached screen, a caregiver can get them in real time on his portable device. Therefore, a mobile device can serve as output for many smart devices. Since all sensors propose a publisher, getting real time information on such a distant screen is quite an easy task. This also opens smart device to all kind of screens, ranging from full size tablets to smart watches. In [72] we show how a hospital could take advantage of such smart devices to build an alert escalation system. In this paper we showed that the values measured by smart devices can be checked by some service to decide whether a given measure is in between acceptable boundaries or represents a danger to the patient's life. In the second case it is important to alert a caregiver and to ensure that at least one caregiver acknowledges the alert and checks the patient's condition.

10.3.2. Virtual only Resources

The different physical devices are one part of Figure 10.15. Yet, there are many other resources that neither are a device nor can be augmented with devices. This is for example the case for the `patient` and the `caregiver`. Both are of central interest. The caregiver is generally the actor initiating the use-case, whereas the patient is the receiver of the use-case. It is thus important to give them a virtual counterpart and embed them in the xWoT. This can easily be achieved with virtual only resources. Such resources have no physical device attached and the physical and virtual sides are only loosely coupled through tags. Regarding the use-case diagram 10.15 the following objects will be modeled as virtual only, since their connection with their physical counterpart is too weak to take their physical manifestation into consideration:

- Caregiver,
- Patient and
- Medical Record.

These three entities will become top level resources. Again, it is arguable whether each of these resources is to be modeled as an entity and get its own xWoT service or whether they are direct children of the hospital entity and are also attached to the latter. Since all these services require a dedicated server to run them, it is less cumbersome to create only

one hospital service exposing the hospital as root resources having the other resources as direct children. Finally, to ensure that only one service is generated, the hospital resource must have its `composed` property set to `false`. This analysis leads to the hierarchy model in Figure 10.17.

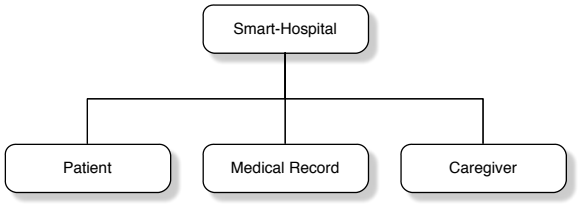


Fig. 10.17.: Initial Hierarchy of the Smart Hospital

Yet, the use-case diagram on Figure 10.15 still contains some elements that are neither devices nor represented in the hierarchy of the Figure 10.17. These are mainly:

- Lifestyle factors,
- Preconditions,
- Medications,
- Analyses and
- Consultations.

Although in the physical world and the physical medical record file, most would be represented as a child resource of the medical record, this hierarchy can be optimized in the virtual world. Whereas it makes sense to attach the consultations and medications to the medical record, lifestyle factors can be attached to the patient. Commonly, they don't change depending on the medical condition of a patient and are therefore interpreted like a patient's name or social security number. Preconditions might as well be attached to the medical record as to the patient. Although they are subject to more frequent changes than the lifestyle factors, it still makes sense to keep them separate from the medical records and attached directly to the patient. Figure 10.18 incorporates these observations by attaching the lifestyle factors as well as the preconditions to the patient resource. Finally, the last element is the analyses. Unlike consultations or lifestyle factors, analyses are based on scientific measures. Usually, specialized hardware is use to conduct such analyses. Therefore, the question whether analyses are a top level resource, a standalone service or attached to some of the already discussed resources is legitimate. Although analyses only make sense in the context of a given patient, they are carried out by sensors and other specialized hardware. Therefore we choose to make the analyses a top level resource. The REST requirement for connectedness permits keeping all these resources together in a consistent manner. An analysis, for example, contains a link to the patient to which the analysis relates. Additionally, the analysis also links to the caregiver who ordered it. This is only one example where the connectedness helps to build the bigger picture out of individual resources. In reality, many more links between resources are implemented.

The hierarchy on Figure 10.18 now guides the modeling of the virtual hospital. For most parts, the hierarchy directly translates into `Resources`. For others some tweaking is necessary. The patients, caregivers and medical records being groups, each one needs an additional aggregation layer in between them and the smart hospital resource. Upon

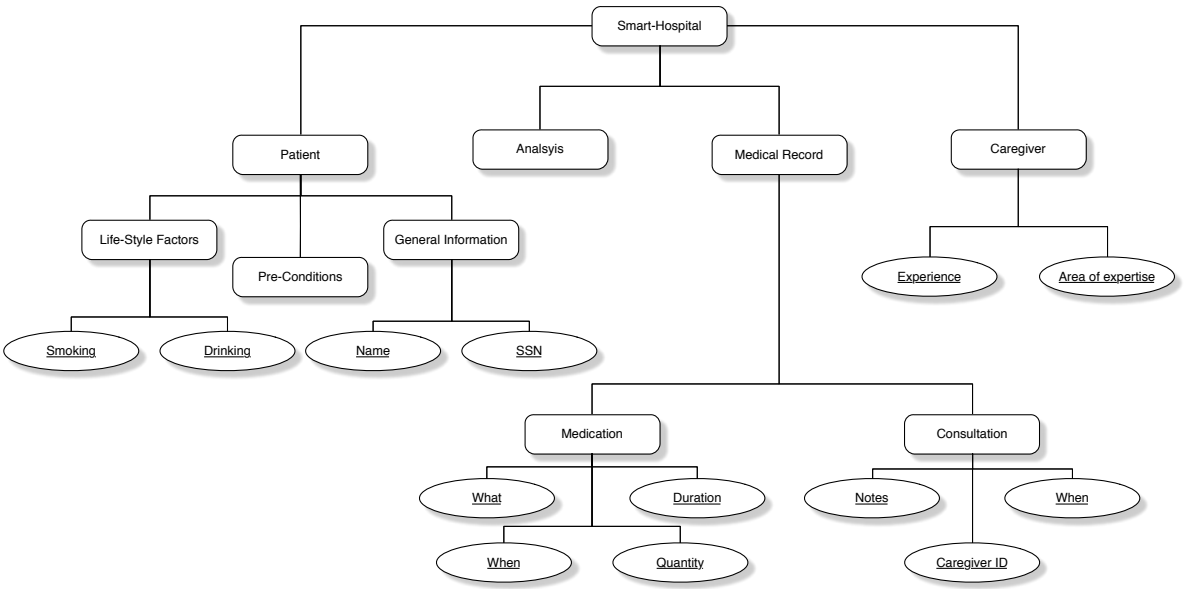


Fig. 10.18.: Final Hierarchy of the Smart Hospital

translation of the hierarchy into an xWoT model, all the elements with cardinality greater than 1 need to be packaged into an aggregation resource. This is the case for medications, analyses and consultations. Additionally, the hierarchy on Figure 10.18 contains squares and circles. Squares translate to resources whereas, circles represents the attributes of the resource they are attached to. These attributes are in no way complete but rather a starting point to create the XSD files later. As such, they serve as examples to designate what type of information each resource represents.

```
ruppena@tungdil:src$ model2Python -i hospital.xwot
INFO - Start processing
INFO - Creating Server: REST-Servers/_hospital_Server
INFO - Successfully created the necessary service(s)
ruppena@tungdil:src$ ll REST-Servers/
total 0
drwxrwxr-x 3 ruppena staff 1.0K Jan 29 17:37 _hospital_Server/
```

List. 10.12: Generated Smart Hospital Service Skeletons

The resulting model is too big to fit on one figure, yet Figure 10.19 shows a partial view of the model. As discussed in Section 9.4, the meta-model also allows the set of HTTP methods the resource supports to be associated with each resource. Figure 10.19 shows that all resources support the `GET` method to fetch information from them. Furthermore, aggregation resources are used to create new child elements, so they also support the `POST` method. All resources dealing with atomic information must allow their manipulation. Consequently, resources like `caregiverResource` also implement the `PUT` method for modifications as well as the `DELETE` method for deletion. At this stage, the model does not need any further tweaking and is therefore ready for the compiler. Listing 10.12 shows the compiler output as well as the enumeration of the generated service skeletons. Since all resources have their `composed` flag set to `false` only one xWoT service skeleton is created. This is exactly what is required.

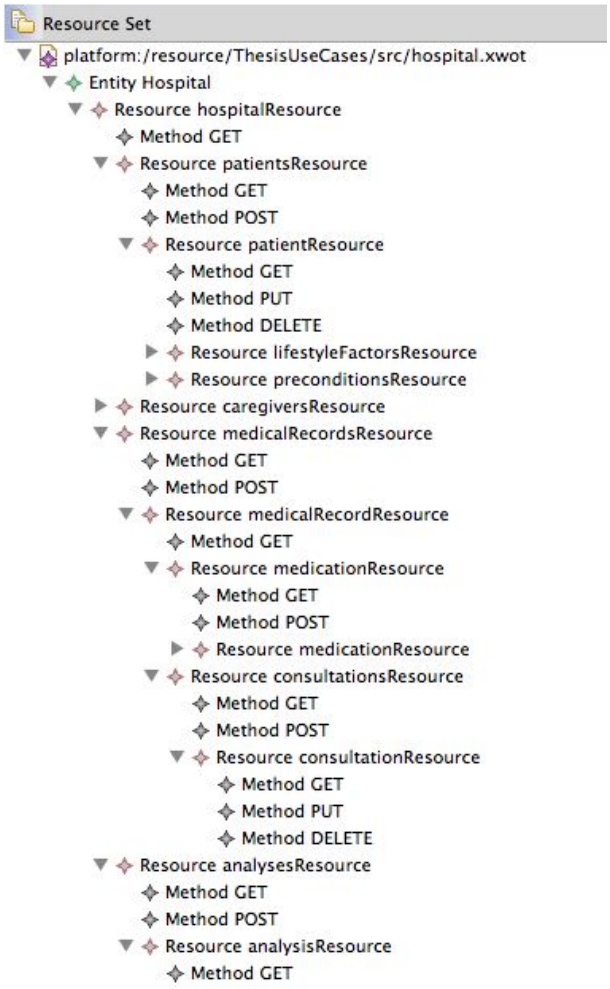


Fig. 10.19.: The smart hospital modeled as an xWoT model (partial view)

The last remaining steps in the modeling process are the definition of the input and output formats, the code gap filling and the deployment. All resources in this hierarchy are purely virtual and therefore they don't get the information they deliver from sensors and actuators. Instead, the information is stored in some database. What type of storing system is used, and how the connection to this system is implemented does not matter. So, it might be possible that some information is stored in a local database and other information is pulled from a CRM. Usually, `GET` requests execute one or several select queries (or their equivalent for a CRM (Customer Relationship Management) system). The result of these queries is processed to fit the defined output format. On the other hand, `POST`, `PUT` and `DELETE` modify the underlying database with either insert, update or delete queries. In his Bachelor Thesis [49], Mathieu fully describes the implementation of such a system with proper inputs and outputs in JSON, XML and HTML (see Figure 10.20) and the proof that the current model leads to a nice usable RESTful API.

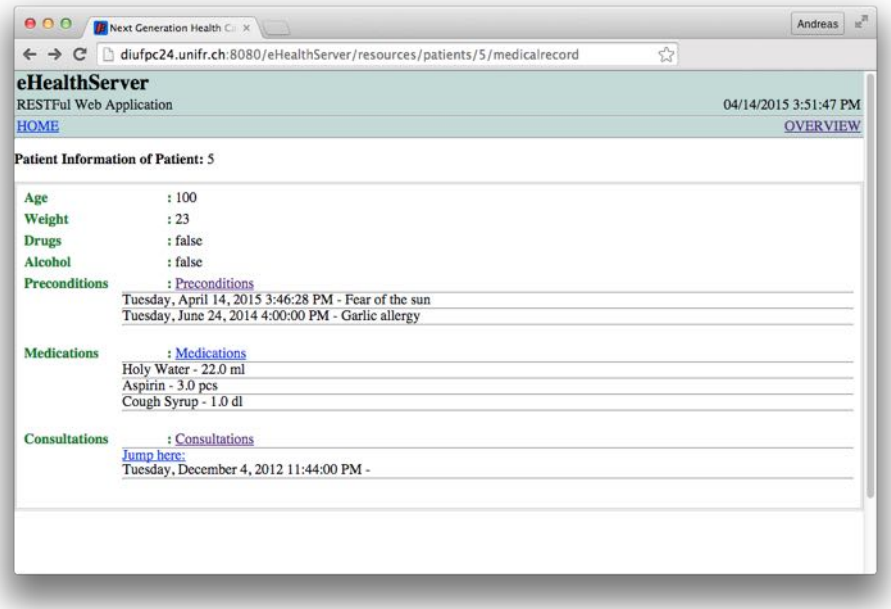


Fig. 10.20.: HTML representation of a Medical Record

The inputs and outputs for analyses are quite difficult to design due to the variety of different types of analyses, so, the most generic approach is chosen. Besides some fields used for connectedness like the `caregiverid` and the `patientid` and some meta-data about the analysis like the `startdate` and `enddate` the format defines a generic `input` field which can contain anything. Mostly, its content will be some base64 encoded binary data. For the RESTful web service the format does not really matter. The machine executing the analysis should be able to read and interpret this information. However, the input can also be empty. As will bee seen in Subsection 10.3.3, the smart hospital also contains a bunch of mashup applications for different needs. If the initiator of an analysis is a mashup application, the `input` element contains nothing. The generated output is also difficult to define. Yet, in the introduction to the current use-case, we stated that an analysis is quantifiable and executed by sensors or other specialized hardware. If the outcome of an analysis were not quantifiable (for example when the caregiver diagnoses a broken arm), then the result should rather be stored in the medical record.

Therefore, it is safe to assume that an analysis produces values. Here again, the defined format is very generic. The `results` contain all outcomes of an analysis. As shown in Figure 10.14, sometimes one analysis consists of a multitude of measures; each is stored in its own `result` element. Finally, a result only contains the numerical value of the current measure, yet due to the variety of possible measures, it is mandatory to specify the `units` and to later distinguish the different results by a unique `key` for each measure. Besides, the `value` element contains a `timestamp` attribute. This information can be used to show any time-dependent evolutions of a given measure. In fact, for some diseases it is important to show that a given value is within a predefined threshold before stopping the treatment.

Discussing the `Patient` and `Caregiver` resources would only bloat this section. Both contain barely more information than any classic CRM system. The RESTful API adopts the resource hierarchy explained in Figure 10.18 and exposes for each child resource the corresponding information. Again, where this information is fetched and stored highly depends on the already deployed infrastructure in the hospital. Yet, in the simplest case they simply use some relational database that the REST API can query.

10.3.3. eHealth Mashup

At this point the smart hospital already has many different smart devices like life support machines, blood analysis equipment, automatic drug dispensers, etc. There are also some purely virtual resources representing objects which cannot be rendered smart through the addition of sensors and actuators like the medical records file or the patient. In this subsection we will further investigate how a smart hospital can take advantage of all the deployed resources and make the nursing smart. We will first examine a sort of manual mashup application linking resource dynamically. The second part introduces processes and shows how processes, mashups and xWoT components go hand in hand to provide a better service to both patients and caregivers.

The previous subsection briefly discussed a smaller use-case where a caregiver used his tablet as a distant screen to plot the measures from one of the different smart devices monitoring a patient's medical condition. Such an application becomes reality by the implemented pushing mechanism in each xWoT device. In its simplest form, some WebSocket client is sufficient to implement this scenario. For example, Figure 10.21 shows what such a distant monitor for taking a patient's temperature over a short interval of time could look like. The monitor can be used to plot various sensor information; only the WebSocket message parsing needs to be adapted. However, such a simple parsing can be prepared for all sensors in the form of regular expression. It is then sufficient to choose the right expression according to the sensor to be monitored.

Although Figure 10.21 is not really what is usually considered a mashup application (only one sensor is consumed) it shows the trend of what is possible. This simple scenario can be extended to create a fully blown mashup application. If instead of just quitting the application, the caregiver is given the choice to save the results for later consultation, the mashup has already integrated two resources. Upon confirmation, the mashup application then takes all the readings and saves them, for example, as a new analysis for the patient. Although this might not be useful for a simple temperature plotter, for ECG (Electrocardiogram) and EEG (Electroencephalogram) this is the standard procedure, especially since the measures are usually not taken by the physician himself but rather by

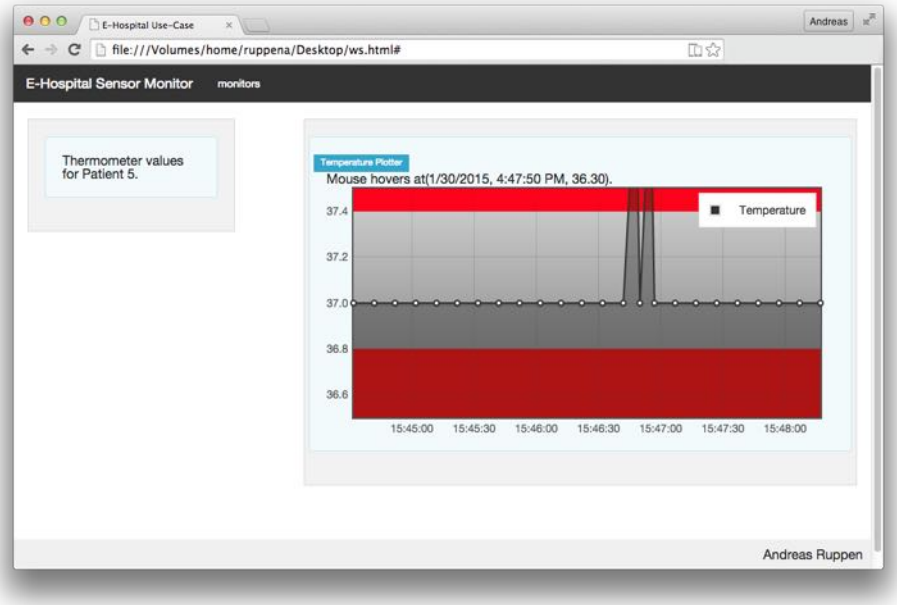


Fig. 10.21.: Pushing real-time of a smart clinical thermometer

a nurse. In the case of the smart clinical thermometer, saving the information for later consultation would lead to a new analysis resource similar to Listing 10.13. This simple example shows how mashup applications can enhance the nursing process. Although this is a quite simple example, there are no limits as to how such a complex mashup could look like. For instance, imagine a heart rate mashup takes into account the patient's age to adapt the desirable interval for his heart rate. Mixing this information helps the caregivers to gain valuable information at a glance, which would be difficult to find out in the physical world.

Section 8.3 discussed the nuance between mashup applications and hubs and concluded that sometimes it is better to propose a given functionality as a hub so that other clients can build upon this service. Additionally, Subsection 8.2.1 classified the different types of services of interest for the xWoT. Among them, business processes were identified as industry standard and an approach to embed them into the xWoT was discussed. Business processes are a perfect example of a situation where a mashup should be replaced by a hub. Behind the scenes, such a business process is very similar to what could be accomplished with a mashup application. Consider the following example: A patient needs different measures of his EEG at predefined intervals over a period of time. Since, the measures can be taken with a sophisticated machine, no human intervention is necessary to take the raw measures. Therefore, it is desirable for the patient to stay at home and does not occupy valuable resources in the hospital. Yet, at some point of time, a physician needs to analyze the different EEG plots. Therefore, each measure is stored as a new analysis. Since such scenarios are standard procedures, they can be modeled once in BPM and then deployed whenever needed and for any patient. Other benefits of business processes compared to mashup applications is that they can: (1) Run in parallel. Several business processes can be executed at the same time, either on the same patient or on different patients. (2) Run in the background. They don't block a terminal until the measures are terminated. Therefore, to build a smart hospital it also makes sense to consider business

processes exploiting the capabilities of the different available smart devices.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<analysis xmlns="http://domain.ehealth.digsim.homelinux.org"
    ↪ uri="http://localhost:8080/AnalysisServer/resources/analyses/112829951/">
    <id>112829951</id>
    <caregiverid>1</caregiverid>
    <patientid>5</patientid>
    <results>
        <result uri="http://[...]/analyses/112829951/results/555/">
            <value key="temperature" units="celsius"
                ↪ timestamp="1422629685">37.2</value>
        </result>
        <result uri="http://[...]/analyses/112829951/results/556/">
            <value key="temperature" units="celsius"
                ↪ timestamp="1422629695">37.1</value>
        </result>
        ...
    </results>
    <input>......</input>
    <startdate>2015-01-30T15:54:45+02:00</startdate>
    <enddate>2015-01-30T15:54:55+02:00</enddate>
</analysis>
```

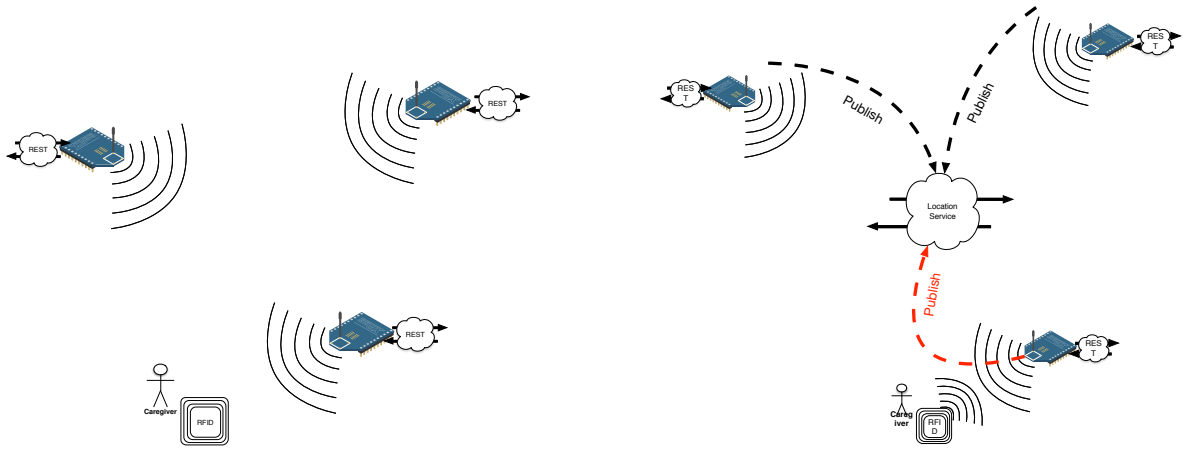
List. 10.13: XML representations of an Analysis

10.3.4. Location Service — Tags Revisited

Throughout this thesis we defined the physical artifacts as actuators, sensors or tags. Whereas it seems easy to model either a sensor or an actuator with the meta-model, things get a little bit more problematic when it comes to tags. According to the definition introduced earlier, tags are passive attributes. Therefore, they neither generate data nor accept commands. Instead, they identify objects and serve as input for tag readers. Consider a use-case involving tags, tag sensors and services still in the hospital environment. Each day, a given number of caregivers are on duty. Unlike in an office environment where each employee has a fixed working place, caregivers move around in the hospital. When an emergency occurs, it is important to know which caregiver is most suitable to give aid. This decision depends on the specialty of the caregiver, but also on his location. To build such a system, the hospital needs to track the caregivers and store their location somewhere. Like the patients, caregivers can be identified through RFID tags. The hospital places RFID readers at strategical points in the building. Such points include the entry to operating rooms, main entrances and doors between the different departments. Therefore, each time a caregiver passes one of these points, the associated RFID reader scans the tag and knows which caregiver has just passed.

At first glance, modeling this use-case seems straightforward; the caregiver resource gets a new attribute `location` containing the last known position. However, such a model would require the caregiver to have some sensor attached to him reporting his position. Yet, the caregiver only has his tag, the sensors being installed at fixed locations. Therefore, the better approach is to model the sensor, capturing the different tags passing by as depicted in Figure 10.22a. Using the meta-model this requires a Sensor on the physical

side plus a corresponding Sensor Resource on the virtual side leading to a simple REST API.



(a) RFID tag reader providing a distributed location service

(b) Centralized Location service as publisher endpoint

Fig. 10.22.: RFID tags in the hospital

Each of these RFID sensors responds to `GET` requests containing the last scanned RFID tags. Therefore, to know the position of a caregiver, it is necessary to send `GET` requests to *all* deployed RFID sensors, which is cumbersome. Furthermore, this does not guarantee that one of the sensors will respond with the caregiver's location. If the caregiver has not moved for some time, chances are high that no RFID holds old enough data to find the caregiver. Therefore another approach is needed. The requirements remain the same: the system must be capable of reporting the last location for each caregiver. Furthermore, it must respect xWoT design principles. The easiest way to achieve this goal is to implement a service providing this exact information like the one in Figure 10.22b. This is a purely virtual service with no physical counterpart. It takes its information out of a database containing for each caregiver his last known location. To fill in the database, the location service is subscribed to all RFID sensor Publisher Resources. Each time one of the RFID sensors scans the tag of a caregiver; it produces an event, which is pushed to the location service. The latter stores this new information in its local database. From this point on, the location service is able to answer `GET` requests, returning locations associated with each caregiver. Respecting the HATEOAS principle, the caregiver resource mentioned earlier should be modified to contain a link pointing to his location served by the location service.

11
Conclusion and Outlook

The main goal of this thesis is to introduce the xWoT together with a meta-model to support software engineers in the task of creating new smart device ecosystem. This chapter summarizes the main contributions of the present work and concludes by considering future work and further research questions.

11.1. Summary of Contributions

This chapter returns to the major contributions of this thesis as defined in Chapter 1, of which there are four: (1) An extension of the classic WoT to include service components allowing the integration of algorithms and business processes working on sensor data or managing actuators. (2) A clean publishing mechanism compatible with RESTful architectures. (3) A meta-model to define structure and to name the important elements of the xWoT. (4) A component based approach whereby the meta-model defines the shape of the building blocks of the xWoT and allows automatic generation of their code skeletons. Each contribution is reviewed and its impact on the current WoT discussed.

The xWoT

The classic WoT is all about sensors and actuators and how they can be embedded in the Internet. It is no secret that in the near future, the number of things participating in the web will outnumber the number of human participants [WEB63]. This leads to a flood of information that no human can deal with anymore. Big Data applications [B11] and Linked Open Data [WEB30] can handle huge quantities of information, where humans are rapidly overwhelmed. Meyer et al. propose a solution where smart devices can link to algorithms, which are input compatible with smart devices [50]. Although, this seems to work, the integration is not seamless. Things outside the WoT suddenly cooperate with services outside the WoT. This is where the xWoT tries to unify the different types of resources. The xWoT proposes to make no difference between a resource standing for a

physical device or just for some algorithm. By analyzing the different types of services, we conclude that there are only 5 classes of services: (i) Short Living, (ii) Real-time, (iii) Delayed, (iv) Decomposable-Delayed and (v) Business processes.

Smart devices are usually in one of the first two categories. The three others are reserved for computationally intensive resources. Since the xWoT combines smart devices with other RESTful web services, a user cannot distinguish between the two anymore. This comes in handy for aggregating data. The smart home use case in Subsection 10.2.3 contained smart light bulbs and smart doors, all deployed independently. However, the example also implemented a `smart floor` and `smart building` layer. In this use-case, these two resources can be seen as aggregation of several smart doors and smart light bulbs. In this way, it becomes possible to get the overall states or updates from several smart devices by invoking just one, single resource. Such an aggregation of resources have no physical equivalent, yet they are useful to a point where the WoT cannot ignore them.

The next three categories are also important for the xWoT. Components in these categories are generally only loosely coupled to their physical counterpart, if any. The examples presented throughout this thesis show how these services can be seamlessly integrated into the xWoT. Furthermore, the examples show that mashup applications are not the solution to all problems. For the xWoT, it is better to provide business logic, as xWoT component are embedded in the Web of Things.

A Publishing Mechanism

Smart environments rely on events. For example, `IFTTT` [WEB22], a popular IoT mashup editor, relies on some initial event to fire adequate reactions. This allows the creation of simple `if-then` scenarios like: `If: ,,door opens‘‘ then ,,Switch floor light on‘‘`. However, the way the system achieves this goal today is suboptimal; if an IFTTT recipe defines `if ,,the temperature drops below` 18°C`‘‘ then ,,increase the thermostat‘‘` then IFTTT continuously polls the website associated with the Netatmo Weather Station to check the current temperature. This situation could be tremendously improved if the Netatmo Weather Station could send a notification about events to the IFTTT service.

To improve the situation, the xWoT mandates for a clearly defined pushing infrastructure to come bundled with each smart device. Subsection 8.2.2 discussed several pushing mechanisms used in the web and their advantages and disadvantages. Two suitable approaches for the xWoT were selected: (1) WebSockets and (2) Webhooks. By default, each xWoT component comes equipped with both approaches, letting the client choose which method is suitable for the given use-case. The eHealth mashup of Subsection 10.3.3 showed for example, how a WebSocket could be used to implement a distant monitor continuously displaying sensor readings. The implementation of this example proved that the chosen approach works and is quite reliable. Therefore, a WebSocket application is well indicated to follow the evolution of a sensor in real-time over a relative short amount of time. However, for more isolated events, webhooks are more suitable. Additionally, using this second approach, a client can register for several distinct events and get notified about them.

Although it is already possible to add various publishing mechanisms to a smart device, the real strength of the approach here is its support from the meta-model. This has the advantage that publisher behaves in a predictable manner. Therefore, upon querying a

sensor, it is very likely that this sensor offers a sub resource `pub/` containing the publisher. Such conventions are useful for creating semi-automatic mashup applications.

A Meta-Model

The meta-model and its associated tools are the most notable outcome of this thesis. Erik Wilde perceived early on that the WoT lacks supporting tools and methodologies to guide system architects and engineers through the design of new WoT components. [79]. This absence of tools and conventions led to the Things crisis. It is possible to overcome this situation by accepting a common playground. The xWoT meta-model introduces some conventions each smart device has to adopt. The benefits, are manifold; by adopting the meta-model, engineers get the methodology to create new smart devices. Since each future xWoT component is designed as an instance of the xWoT meta-model, the associated tools take care of supporting the developers in their tasks. These tools can automatically generate missing parts but also implement skeletons for each required component. The developer can therefore concentrate on the core of the components, provide the business logic and wire the necessary electronics. Furthermore, all smart devices share some common mechanisms like the implemented publishing approaches. This makes the behavior of smart devices predictable and eases the creation of either mashup applications or business processes.

A Component-Based Approach for the xWoT

The natural output of the xWoT meta-model methodology are independent and reusable components. These are deployed directly on the smart device, with the advantage that each smart device operates on its own and can be reused in other scenarios. Additionally, this makes platforms like Xively almost obsolete. They still have some use when it comes to sharing things, but are no longer necessary for taking full advantage of smart devices. The components produced by the meta-model and its associated tools are the building blocks of the WoT and therefore also the xWoT. All other applications that take advantage of smart devices rely upon these building blocks. The meta-model is smart enough to create the necessary atomic components, which can be re-used in other scenarios. However, instead of limiting the developer to only primitive smart devices, the meta-model also embraces virtual devices, a composition of different physical devices. This duality allows both, a maximum reusability of the generated (device) components and design as well as implementation of interesting (node manager) components. Previsouly, creating a smart home like in Subsection 10.2.3 was limited to mashup applications. The meta-model breaks with this approach and allows for the creation of complex hubs, appearing to the client like any other smart device.

11.2. Future Work and Open Research Questions

The xWoT meta-model supports the vision of small, independently and deployed smart devices in all aspects of our lives, each providing a useful service on its own. However, they can be easily combined into novel applications and adapted to changing needs. Through the combination of purely virtual services and smart devices the xWoT can take advantage

of deployed smart devices to produce virtual devices. For a client, there is no difference between a service tied to a physical device and a service without any attached hardware. Again, this vision where compositions lead to new xWoT components participates in the spread of the xWoT. However much work remains to be done. The xWoT meta-model allows the re-consideration of how mashup applications are built today. Having the meta-model as a base allows a future with smart mashup applications to be imagined, where inexperienced users with no IT skills can create and adapt their own mashups. To achieve this visions several problems needs to be solved.

Event Multithreading

The REST services generated from the model support multiple clients at the same time. This is mainly due to the underlying HTTP framework, which allows two clients to subscribe for notifications at the same time. When the service creates an event and notifies the subscribers, it iterates through the list of subscribed clients and sends out the notification. While this approach is simple and works well in situations with only a few subscribers, it gets more complicated as the number of subscribers and subscribed events grows. Sending a notification involves opening a TCP connection, doing an HTTP handshake and sending the message. If this action were to take 1 second per client, this would lead to a high server occupation for a small number of subscribers, resulting in not all subscribers being notified simultaneously. In this case the generated code would greatly benefit from an asynchronous approach. The difficulty is finding a good compromise between the limited capabilities of the relatively cheap hardware and the gain in time.

Event Language

By providing publishing as a drag and drop component, the meta-model helps to provide a better pushing mechanism to smart objects. Whereas the use-case for the WebSocket endpoint is clear and its applicability has been shown through some examples, the approach based on Webhooks would gain in usability if the system had some predicate language allowing a user to define which events he is interested in as this removes another decision from the developer. Without such a language, the developer has to define what amount of change in a system leads to an event. However, this is an almost impossible task: for a temperature sensor, is a change of 0.2 degrees Celsius a big enough change, are 10 degrees a big enough change? Chances are high that there is not answer satisfying all needs. Furthermore, if the developers decide that an event has been generated, it is pushed to all the clients. In a world with millions of smart-devices this only leads to an unnecessary congestion of the network with useless notifications.

Discovery

Currently, mashup applications are not portable. They are tied to one given environment. Transporting them to another environment, even a similar one, requires a good amount of engineering. This is due to how mashup applications work: the data providers and data consumers (smart devices and other xWoT services) are mostly hardcoded. If one smart device changes, the mashup breaks. For some time now, the WS-* world has

discovered the power of late binding for services, which allows the creation of abstract business processes where the worker is chosen during the run time depending on the available services. Mashup applications would benefit in a similar way from such a late binding approach, as this would allow recipes to be created that users could download and deploy in their environment (as long as they have the compatible data providers). Most serious works in the domain of the WoT have briefly brought up the problem of discovery. A notable amount of papers, however, conclude by favoring mDNS (Domain Name System) as a discovery mechanism for the WoT, arguing that it is open-source and well established [rfc7252, 75]. However a true RESTful discovery should respect RESTful principles, as depicted by R. Fielding.

Semantics

A RESTful discovery mechanism allows deployed smart devices to be found. However, the discovery itself says nothing about the capabilities of the devices found nor their inputs and outputs. The only reliable way to define them is by injecting some semantics into the discovery. Yet, to separate the concerns and keep the protocol clean, semantics should be done in a separate layer, independ of the discovery. Furthermore, semantics would not only benefit the discovery of smart devices, but also the creation of mashup applications. Here semantics would allow the automatic preparation of the outputs of one components in a format suitable as input for the next component. Since semantics is used for several tasks, it should also be implemented outside the meta-model. However, it should take advantage of the definitions and conventions introduced by the latter.

11.3. Final Thoughts

This thesis analyzed the current situation and concludes that the WoT is in a crisis. The only way out of this crisis is to adopt a common playground and create truly re-usable smart devices. Through discussing the different aspects in this thesis, we have tried to give an answer to this problem and point the way out of the Things Crisis. Although the present work allows for an exit from this crisis, this is only the first step. Embracing the meta-model and its methodology opens new visions for the future of the WoT. This work allows a future to be imagined where a company not only sells sensors and actuators but also builds and sells the accompanying software for turning the hardware into a smart device. For clients this means that they no longer acquire raw hardware but a ready-to-be-deployed xWoT component.

Furthermore, in a world full of smart devices respecting the meta-model more sophisticated and intelligent mashup editors can be imagined. Through the discovery mechanism, such an editor can present the user with a selection of available devices. HTML 5 makes it easy to build fancy GUIs similar to what users are accustomed to from desktop application. Through drag and drop, the found smart devices can be combined into novel mashups. Semantics is necessary to seamlessly combine the output of one xWoT component with the input of the next one. All these ingredients are necessary to create an ecosystem of smart devices, so the WoT really will be their oyster.

Part IV.

Appendix

A
Common Acronyms

ADT Abstract Data Type
AED Automated external defibrillator
ANSI American National Standards Institute
API Application Programming Interface
APT Advanced Packaging Tool
ARPA Advanced Research Projects Agency
ARPANET Advanced Research Projects Agency Network
B2B Business-to-Business
BPEL Business Process Execution Language
BPM Business Process Modeling
BPMN Business Process Model and Notation
CERN Conseil Européen pour la Recherche Nucléaire
CGI Common Gateway Interface
CMS Content Management System
COM Component Object Model
CORBA Common Object Request Broker Architecture
CPU Central Processing Unit
CRM Customer Relationship Management
CRUD Create, Read, Update, Delete
CSS Cascading Style Sheet
DARPA Defense Advanced Research Projects Agency
DBMS Database Management System
DCOM Distributed Component Object Model
DIY Do-it-yourself
DNS Domain Name System
DSL Domain Specific Language
ECG Electrocardiogram
EEG Electroencephalogram
EMF Eclipse Modeling Framework
EMOF Essential MOF
EPC Electronic Product Code
EPCIS Electronics Product Code Information Services
EPFL École Polytechnique Fédérale de Lausanne
ER Entity-Relation
ESB Enterprise Service Bus

FTP File Transfer Protocol
GNU GNU is Not Unix
GPIO General-purpose input/output
GRDDL Gleaning Resource Descriptions from Dialects of Languages
GUI Graphical User Interface
HATEOAS Hypermedia as the Engine of Application State
HTML Hypertext Markup Language
HTTP Hypertext Transfer Protocol
HTTPS Hypertext Transfer Protocol Secure
HVAC Heating, Ventilation and Air Conditioning
IANA Internet Assigned Numbers Authority
IDE Integrated Development Environment
IDL Interface Definition Language
IMP Interface Message Processor
IP Internet Protocol
IRC Internet Relay Chat
ISP Internet Service Provider
IT Information Technology
IoT Internet of Things
JPA Java Persistence API
JSON JavaScript Object Notation
LAN Local Area Network
LED Light-emitting Diode
LIFO Last-in, First-out
M2M machine-to-machine
MDA Model Driven Architecture
MIT Massachusetts Institute of Technology
MOF Meta Object Facility
MVC Model View Controller
NAT Network Address Translation
NFC Near Field Communication
NSFNet National Science Foundation Network
OMG Object Management Group
OS Operating System
OSI Open Systems Interconnection
P2P Peer-to-Peer
PHP Hypertext Processor
POJO Plain Old Java Object
POM Project Object Model
QR Quick-Response
QoS Quality of Service
RDF Resource Description Framework
RDFa Resource Description Framework in Attributes
REST Representational State Transfer
RFC Request for Comments
RFID Radio Frequency Identification
RMI Remote Method Invocation
ROA Resource Oriented Architecture

RPC Remote Procedure Call
RSS Really Simple Syndication
ReLL Resource Linking Language
SGML Standard Generalized Markup Language
SOA Service Oriented Architecture
SOAP Simple Object Access Protocol
SQL Structured Query Language
SSE Server Side Event
SSL Secure Socket Layer
SUS System under Study
TCP Transmission Control Protocol
TLS Transport Layer Security
UDDI Universal Description Discovery and Integration
UDP User Datagram Protocol
UML Unified Modeling Language
URI Unified Resource Identifier
URL Uniform Resource Locator
VRP-TW Vehicle Routing Problem with Time-Windows
W3C World Wide Web Consortium
WADL Web Application Description Language
WSDL Web Service Description Language
WSN Wireless Sensor Networks
WWW World Wide Web
WoT Web of Things
XDR External Data Representation
XMI XML Metadata Interchange
XML eXtensible Markup Language
XML-RPC Extensible Markup Language Remote Procedure Call
XSD XML Schema Definition
XWoT eXtended Web of Things
xWoT extended WoT

B

License of the Documentation

C

Curriculum Vitae

Personalien	
Name, Vorname:	Ruppen Andreas
Geburtsdatum:	27 November 1981
Zivilstand:	Ledig
Heimatort:	Saas-Grund, VS
Adresse:	Mattenstrasse 2 3175 Flamatt
Natel N^{o}:	079 772 53 78
Email:	andreas.ruppen@unifr.ch
Ausbildung	
1996–1998:	zwei Jahre Kollegium Spiritus Sanctus in Brig (VS).
1998–2001:	drei Jahre Kollegium lycée collège des Creusets in Sion (VS) auf Französisch zur Erlangung der Maturität.
2001–2004:	drei Jahre Mathematik Studium an der EFPL in Lausanne (VD).
2004–2007:	drei Jahre Studium an der Universität in Freiburg (FR) zur Erlangung des Bachelors in Computer Science (zweisprachig deutsch-französisch) mit dem Nebenfach Mathematik.
2007–2009 :	drei Semester Studium an der Universität in Freiburg (FR) zur Erlangung des Master in Computer Science.
2010–2015 :	PhD in Computer Science an der Universität in Freiburg (FR) zur Erlangung des Doctor Scientiarum Informaticarum
Sprachkentnisse	
Französisch-Deutsch:	zweisprachig
Englisch:	verhandlunssicher, gute mündliche und schriftliche Kentnisse
Beruflich Erfahrung	
2005 :	Unterassistent im "Cours de mathématique spéciales" der EPFL für die Konzeption und Implementation eines Content-Management-Systems in PHP und MySQL.
2005:	Unterassistent im "Cours de mathématique spéciales" der EPFL. Die Aufgaben umfassten die Konzeption und Umsetzung einer Java basierten E-learning Software.

2006 :	Unterassistent im "Cours de mathématique spéciales" der EPFL. Das Pflichtenheft umfasste den Entwurf und die Implementation eines Java basieren Administratoren Interfaces für die E-learning Software des vergangenen Jahres.
2006 :	Unterassistent an der Universität in Freiburg bei der Gruppe "Software Engineering". Die Tätigkeit umfasste einerseits die Betreuung der Übungstunden des Kurses "Génie logiciel I", Mithilfe bei der Vorbereitung des Kurses und andererseits den Entwurf einer Software zur Verwaltung der Übungsserien.
2007–2008 :	Unterassistent im "Cours de mathématique spéciales" der EPFL. Weiterführung und Support der Projekte der vergangenen Jahre.
2007–2008 :	Unterassistent an der Universität in Freiburg bei der Gruppe "Telecommunications, Networks & Security". Der Aufgabenbereich umfasste das Betreuen und Vorbereiten der Übungsstunden für den Kurs "Telecommunikations" sowie das Vorbereiten und Durchführen der Praktika.
2009:	Software Engineer bei Fincons AG in Bern.
2010 – 2015 :	PhD bei der Forschungsgruppe Software Engineering an der Universität Fribourg.

Publications

[51]	Ruppen, A. and Meyer, S. "An Approach for a Mutual Integration of the Web of Things with Business Processes" *Enterprise and Organizational Modeling and Simulation, Springer Berlin Heidelberg,* ***2013****, 153, 42-56*
[52]	Meyer, S.; Ruppen, A. and Magerkurth, C "Internet of Things-aware Process Modeling: Integrating IoT Devices as Business Process Resources" *CAiSE 2013,* 2013
[69]	Ruppen, A. and Pasquier-Rocha, J. "A Meta-Model for the Web of Things" *Department of Informatics, University of Fribourg, Switzerland,* ***2013***
[72]	Ruppen, A.; Pasquier, J.; Wagen, J.-F.; Wolf, B. and Guye, R. A WoT approach to eHealth: case study of a hospital laboratory alert escalation system *Proceedings of the Third International Workshop on the Web of Things, ACM,* ***2012****, 6:1-6:6*
[71]	Ruppen, A.; Pasquier, J. and Hurlimann, T. A RESTful Architecture for Integrating Decomposable Delayed Services within the Web of Things *Parallel and Distributed Systems (ICPADS), 2011 IEEE 17th International Conference on,* 2011*, 860 -865*
[70]	Ruppen, A.; Pasquier, J. and Hurlimann, T. A RESTful Architecture for Integrating Decomposable Delayed Services within the Web of Things *Int. J. Internet Protoc. Technol., Inderscience Publishers,* 2011*, 6, 247-259*

[45] Latze, C.; Ruppen, A. and Ultes-Nitsche, U. A Proof of Concept Implementation of a Secure E-Commerce Authentication Scheme *Information Security South Africa Conference 2009, School of Tourism & Hospitality, University of Johannesburg, Johannesburg, South Africa, July 6-8,* ***2009****. Proceedings ISSA2009, 2009, 379-392*

D
Ehrenwörtliche Erklärung

Ich bestätige mit meiner Unterschrift, dass ich die Arbeit persönlich erstellt und dabei nur die aufgeführten Quellen und Hilfsmittel verwendet sowie wörtliche Zitate und Paraphrasen als solche gekennzeichnet habe.

July 14, 2015

Andreas Ruppen

E
References

Books

[B1] Gustavo Alonso, Fabio Casati, Harumi Kuno, and Vijay Machiraju. *Web Services: Concepts, Architectures and Applications*. Berlin, Germany: Springer-Verlag (cit. on p. 22).

[B2] P. Baran. *On Distributed Communications*. The Rand Corporation, Memorandum RM-3420-PR, 1964 (cit. on pp. 9 sq.).

[B3] Andrew Blum. *Tubes: A Journey to the Center of the Internet*. HarperCollins, 2012 (cit. on p. 10).

[B4] D. Briskorn. *Sports leagues scheduling: models, combinatorial properties, and optimization algorithms*. Lecture notes in economics and mathematical systems. Springer, 2008 (cit. on p. 116).

[B5] Bill Burke. *RESTful Java with Jax-RS*. 1st. O'Reilly Media, Inc., 2009 (cit. on pp. 59, 63).

[B6] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995 (cit. on pp. 4, 76 sq.).

[B7] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of software engineering (2. ed.)* Prentice Hall, 2003, pp. I–XX, 1–604 (cit. on p. 61).

[B8] B.L. Golden, S. Raghavan, E.A. Wasil, and Inc NetLibrary. *The vehicle routing problem: latest advances and new challenges*. Springer, 2008 (cit. on p. 117).

[B9] Cesar Gonzalez-Perez and Brian Henderson-Sellers. *Metamodelling for software engineering*. Wiley, 2008, pp. I–IX, 1–210 (cit. on pp. 4, 90 sq., 94, 97 sq., 100, 103).

[B10] Brian Henderson-Sellers. *On the Mathematics of Modelling, Metamodelling, Ontologies and Modelling Languages*. Berlin, Heidelberg: Springer Publishing Company, Incorporated, 2012. DOI: `http://link.springer.com/book/10.1007/978-3-642-29825-7/page/1` (cit. on pp. 4, 93 sqq.).

[B11] Judith Hurwitz, Alan Nugent, Fern Halper, and Marcia Kaufman. *Big Data For Dummies*. 1st. For Dummies, 2013 (cit. on p. 217).

[B12] L. Kleinrock. *Message Delay in Communication Nets with Storage*. Massachusetts Institute of Technology, Department of Electrical Engineering, 1962 (cit. on p. 9).

[B13] Serge Lang. *Undergraduate Algebra*. Springer Undergraduate Texts in Mathematics and Technology. Springer, 2001 (cit. on p. 96).

[B14] Peter Lubbers, Brian Albers, Ric Smith, and Frank Salim. *Pro HTML5 Programming: Powerful APIs for Richer Internet Application Development*. 1st. Berkely, CA, USA: Apress, 2010 (cit. on p. 129).

[B15] Bertrand Meyer. *Eiffel: The Language*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1992 (cit. on p. 76).

[B16] Bertrand Meyer. *Object-Oriented Software Construction*. 1st. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1988 (cit. on p. 76).

[B17] Bertrand Meyer. *Object-oriented Software Construction (2Nd Ed.)* Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1997 (cit. on p. 76).

[B18] Leonard Richardson and Sam Ruby. *RESTful web services*. 1st ed. O'Reilly Media, Inc., 5/15/2007 (cit. on pp. 59, 63, 69, 112, 117, 155, 193).

[B19] Thomas Stahl, Markus Völter, Sven Efftinge, and Arno Haase. *Modellgetriebene Softwareentwicklung: Techniken, Engineering, Management*. 2. Heidelberg: dpunkt, 2007 (cit. on pp. 153, 175).

[B20] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. 2nd. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002 (cit. on pp. 74, 79 sq.).

[B21] Andrew Tanenbaum. *Computer Networks*. 4th. Prentice Hall Professional Technical Reference, 2002 (cit. on pp. 21 sq., 27 sq., 46).

[B22] S. Ullmann. *Semantics: an introduction to the science of meaning*. Barnes & Noble, 1962 (cit. on p. 92).

Journal Articles

[1] Rosa Alarcón and Erik Wilde. "RESTler: Crawling RESTful Services". In: *Proceedings of the 19th International Conference on World Wide Web*. WWW '10. Raleigh, North Carolina, USA: ACM, 2010, pp. 1051–1052. DOI: 10.1145/1772690.1772799 (cit. on pp. 149, 159, 169).

[2] Rosa Alarcon, Erik Wilde, and Jesus Bellido. "Hypermedia-driven RESTful Service Composition". In: *Proceedings of the 2010 International Conference on*

Service-oriented Computing. ICSOC'10. San Francisco, CA: Springer-Verlag, 2011, pp. 111–120 (cit. on p. 109).

[3] Tim Berners-Lee and D. Connolly. *RFC 1866 – Hypertext Markup Language – 2.0*. RFC 1866 (Informational). 11/1995 (cit. on p. 13).

[4] Andrew D. Birrell and Bruce Jay Nelson. "Implementing remote procedure calls". In: *ACM Trans. Comput. Syst.* 2.1 (2/1984), pp. 39–59. DOI: 10.1145/2080.357392 (cit. on p. 23).

[5] Michael Blackstock and Rodger Lea. "WoTKit: a lightweight toolkit for the web of things". In: *Proceedings of the Third International Workshop on the Web of Things*. WOT '12. Newcastle, United Kingdom: ACM, 2012, 3:1–3:6. DOI: 10.1145/2379756.2379759 (cit. on pp. 3, 111).

[6] Francois Carrez. *Final Architectural Reference Model for IoT v3.0*. Tech. rep. The Internet-of-Things Architecture, 5/2013. DOI: http://www.iot-a.eu/public/public-documents/d1.5/at_download/file (cit. on pp. 147 sq.).

[7] A.P. Castellani, N. Bui, P. Casari, M. Rossi, Z. Shelby, and M. Zorzi. "Architecture and protocols for the Internet of Things: A case study". In: *Pervasive Computing and Communications Workshops (PERCOM Workshops), 2010 8th IEEE International Conference on*. 03/2010, pp. 678–683. DOI: 10.1109/PERCOMW.2010.5470520 (cit. on pp. 34, 111).

[8] Luc Clement, Andrew Hately, Claus von Riegen, and Tony Rogers. *UDDI Version 3.0.2*. Tech. rep. Oasis, 10/2004 (cit. on p. 27).

[9] Brad Cox and Bill Hunt. "Objects, Icons and Software-ICs". In: *Byte Magazine* 11.8 (08/1986), pp. 161–176 (cit. on pp. 80, 168).

[10] Suparna De, Payam M. Barnaghi, Martin Bauer, and Stefan Meissner. "Service Modelling for Internet of Things". In: *FedCSIS*. 2011, pp. 949–955 (cit. on p. 149).

[11] O. Dohndorf, J. Kruger, H. Krumm, C. Fiehe, A. Litvina, I. Luck, and F.-J. Stewing. "Towards the Web of Things: Using DPWS to bridge isolated OSGi platforms". In: *Pervasive Computing and Communications Workshops (PERCOM Workshops), 2010 8th IEEE International Conference on*. 03/2010, pp. 720–725. DOI: 10.1109/PERCOMW.2010.5470527 (cit. on p. 111).

[12] Joel Farrell and Holger Lausen. *Semantic Annotations for WSDL and XML Schema*. W3C Working Draft. World Wide Web Consortium, 2007 (cit. on p. 149).

[13] D. Fensel, R. Krummenacher, O. Shafiq, E. Kuehn, J. Riemer, Y. Ding, and B. Draxler. "TSC–Triple Space Computing". In: *e & i Elektrotechnik und Informationstechnik* 124.1 (2007), pp. 31–38 (cit. on p. 149).

[14] Roy Thomas Fielding. "Architectural styles and the design of network-based software architectures". AAI9980887. PhD thesis. 2000 (cit. on pp. 18, 27, 58 sq., 149).

[15] Aitor Gómez-Goiri and Diego López-de-Ipiña. "On the complementarity of triple spaces and the Web of Things". In: *Proceedings of the Second International Workshop on Web of Things*. WoT '11. San Francisco, California: ACM, 2011, 12:1–12:6. DOI: 10.1145/1993966.1993983 (cit. on p. 149).

[16] Aitor Gómez-Goiri, Pablo Orduña, and Diego López-de-Ipiña. "RESTful triple spaces of things". In: *Proceedings of the Third International Workshop on the Web of Things*. WOT '12. Newcastle, United Kingdom: ACM, 2012, 5:1–5:6. DOI: 10.1145/2379756.2379761 (cit. on p. 149).

[17] Martin Gudgin, Marc Hadley, Noah Mendelsohn, Jean-Jacques Moreau, Henrik Frystyk Nielsen, Anish Karmarkar, and Yves Lafon. *SOAP Version 1.2 Part 1: Messaging Framework (Second Edition)*. Tech. rep. 4/2007 (cit. on pp. 25, 27).

[18] D. Guinard, M. Fischer, and V. Trifa. "Sharing using social networks in a composable Web of Things". In: *Pervasive Computing and Communications Workshops (PERCOM Workshops), 2010 8th IEEE International Conference on*. 03/2010, pp. 702–707. DOI: 10.1109/PERCOMW.2010.5470524 (cit. on p. 111).

[19] Dominique Guinard. "A Web of Things Application Architecture – Integrating the Real-World into the Web". PhD thesis. Zurich, Switzerland: ETH Zurich, 8/2011 (cit. on pp. 1, 45, 108, 111).

[20] Dominique Guinard, Mathias Mueller, and Vlad Trifa. "RESTifying Real-World Systems: A Practical Case Study in RFID". English. In: *REST: From Research to Practice*. Ed. by Erik Wilde and Cesare Pautasso. Springer New York, 2011, pp. 359–379. DOI: 10.1007/978-1-4419-8303-9_16 (cit. on p. 1).

[21] Dominique Guinard and Vlad Trifa. "Towards the Web of Things: Web Mashups for Embedded Devices". In: *Workshop on Mashups, Enterprise Mashups and Lightweight Composition on the Web (MEM 2009), in proceedings of WWW (International World Wide Web Conferences)*. Madrid, Spain, 4/2009 (cit. on p. 108).

[22] 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". English. In: *Architecting the Internet of Things*. Ed. by Dieter Uckelmann, Mark Harrison, and Florian Michahelles. Springer Berlin Heidelberg, 2011, pp. 97–129. DOI: 10.1007/978-3-642-19157-2_5 (cit. on pp. 130, 147).

[23] Dominique Guinard, Vlad Trifa, Thomas Pham, and Olivier Liechti. "Towards Physical Mashups in the Web of Things". In: *Proceedings of INSS 2009 (IEEE Sixth International Conference on Networked Sensing Systems)*. Pittsburgh, USA, 6/2009 (cit. on pp. 3, 108).

[24] Dominique Guinard, Vlad Trifa, and Erik Wilde. "A Resource Oriented Architecture for the Web of Things". In: *Proceedings of Internet of Things 2010 International Conference (IoT 2010)*. Tokyo, Japan, 2010 (cit. on p. 108).

[25] Dominique Guinard, Vlad Trifa, and Erik Wilde. *Architecting a Mashable Open World Wide Web of Things*. Technical Report 663. Department of Computer Science, ETH Zurich, 2/2010 (cit. on p. 108).

[26] Giancarlo Guizzardi. "Ontological foundations for structural conceptual models". PhD thesis. Enschede: University of Twente, Enschede, The Netherlands, 10/2005 (cit. on p. 92).

[27] Vipul Gupta, Ron Goldman, and Poornaprajna Udupi. "A network architecture for the Web of Things". In: *Proceedings of the Second International Workshop on Web of Things*. WoT '11. San Francisco, California: ACM, 2011, 3:1–3:6. DOI: 10.1145/1993966.1993971 (cit. on pp. 1, 111, 130).

[28] Marc J. Hadley. *Web Application Description Language (WADL)*. Tech. rep. Mountain View, CA, USA, 2006 (cit. on p. 159).

[29] Stephan Haller. *The Things in the Internet of Things*. Tech. rep. SAP, 2010 (cit. on pp. 121, 147, 150 sq.).

[30] Wolfgang Hesse. "More matters on (meta-)modelling: remarks on Thomas Kuhnes matters". In: *Software and Systems Modeling (SoSyM)* 5.4 (12/2006), pp. 387–394. DOI: http://dx.doi.org/10.1007/s10270-006-0033-9 (cit. on p. 94).

[31] Hanh Huu Hoang, Tai Nguyen-Phuoc Cung, Duy Khanh Truong, Dosam Hwang, and Jason J. Jung. "Semantic Information Integration with Linked Data Mashups Approaches". In: *IJDSN* 2014 (2014). DOI: 10.1155/2014/813875 (cit. on p. 138).

[32] C. A. R. Hoare. "An Axiomatic Basis for Computer Programming". In: *Commun. ACM* 12.10 (10/1969), pp. 576–580. DOI: 10.1145/363235.363259 (cit. on p. 81).

[33] V. Hoyer, K. Stanoesvka-Slabeva, T. Janner, and C. Schroth. "Enterprise Mashups: Design Principles towards the Long Tail of User Needs". In: *Services Computing, 2008. SCC '08. IEEE International Conference on*. Vol. 2. 7/2008. DOI: 10.1109/SCC.2008.88 (cit. on pp. 111, 138).

[34] J.W. Hui and D.E. Culler. "Extending IP to Low-Power, Wireless Personal Area Networks". In: *Internet Computing, IEEE* 12.4 (8/2008), pp. 37–45. DOI: 10.1109/MIC.2008.79 (cit. on p. 34).

[35] *Inspiring the Internet of Things*. 10/2013 (cit. on p. 33).

[36] *Information Technology — Open Systems Interconnection — Basic Reference Model: The Basic Model*. ISO/IEC 7498-1:1994. Geneva, Switzerland: ISO, 11/15/1994 (cit. on pp. 22, 46).

[37] Lee KangChan, Jeon JongHong, Lee WonSeok, Jeong Seong-Ho, and Park Sang-Won. *QoS for Web Services: Requirements and Possible Approaches*. `http://www.w3c.or.kr/kr-office/TR/2003/ws-qos/`. W3C Working Group Note 25 November 2003. 2003 (cit. on p. 83).

[38] Klemen Kenda, Carolina Fortuna, Alexandra Moraru, Dunja Mladenić, Blaž Fortuna, and Marko Grobelnik. "Mashups for the Web of Things". In: *Semantic Mashups*. Springer, 2013, pp. 145–169 (cit. on p. 138).

[39] Tim Kindberg, John Barton, Jeff Morgan, Gene Becker, Debbie Caswell, Philippe Debaty, Gita Gopal, Marcos Frid, Venky Krishnan, Howard Morris, John Schettino, Bill Serra, and Mirjana Spasojevic. "People, places, things: web presence for the real world". In: *Mob. Netw. Appl.* 7.5 (10/2002), pp. 365–376. DOI: `10.1023/A:1016591616731` (cit. on p. 1).

[40] Leonard Kleinrock. "Information Flow in Large Communication Nets, Ph.D. Thesis Proposal". PhD thesis. Cambridge, MA, USA: Massachusetts Institute of Technology, 1961 (cit. on pp. 9 sq.).

[41] Jacek Kopecky, Karthik Gomadam, and Tomas Vitvar. "hRESTS: An HTML Microformat for Describing RESTful Web Services". In: *Web Intelligence and Intelligent Agent Technology, IEEE/WIC/ACM International Conference on* 1 (2008), pp. 619–625. DOI: `http://doi.ieeecomputersociety.org/10.1109/WIIAT.2008.379` (cit. on p. 147).

[42] G. Kortuem, F. Kawsar, D. Fitton, and V. Sundramoorthy. "Smart objects as building blocks for the Internet of things". In: *Internet Computing, IEEE* 14.1 (2/2010), pp. 44–51. DOI: `10.1109/MIC.2009.143` (cit. on pp. 1, 108).

[43] S. Kumaran, Rong Liu, P. Dhoolia, T. Heath, P. Nandi, and F. Pinel. "A RESTful Architecture for Service-Oriented Business Process Execution". In: *e-Business Engineering, 2008. ICEBE '08. IEEE International Conference on*. 8/2008, pp. 197–204. DOI: `10.1109/ICEBE.2008.82` (cit. on pp. 110, 113, 122, 125).

[44] Jon Lathem, Karthik Gomadam, and Amit P. Sheth. "SA-REST and (S)mashups: Adding Semantics to RESTful Services". In: *Proceedings of the International Conference on Semantic Computing*. ICSC '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 469–476. DOI: `10.1109/ICSC.2007.79` (cit. on pp. 111, 138).

[45] Carolin Latze, Andreas Ruppen, and Ulrich Ultes-Nitsche. "A Proof of Concept Implementation of a Secure E-Commerce Authentication Scheme". In: *Information Security South Africa Conference 2009, School of Tourism & Hos-*

pitality, University of Johannesburg, Johannesburg, South Africa, July 6-8, 2009. Proceedings ISSA2009. 2009, pp. 379–392 (cit. on p. 233).

[46] Lamar Ledbetter and Brad Cox. "Software-ICs". In: *Byte Magazine* 10.6 (06/1985), pp. 307–316 (cit. on pp. 2, 74, 80).

[47] Carsten Magerkurth, Klaus Sperner, Sonja Meyer, and Martin Strohbach. "Towards Context-Aware Retail Environments: An Infrastructure Perspective". In: *Proceedings of Mobile Interaction in Retail Environments*. MIRE '11. Stockholm, Sweden: DFKI, 2011 (cit. on p. 122).

[48] Carsten Magerkurth, Klaus Sperner, Sonja Meyer, and Martin Strohbach. "Towards context-aware retail environments: An infrastructure perspective". In: *Mobile Interaction in Retail Environments (MIRE 2011)* (2011) (cit. on p. 110).

[49] Jean-Daniel Mathieu. "A methodology to build RESTful Web-Services for the Web of Things — Using an eHealth scenario as an example". Bachelor Thesis. 2013 (cit. on p. 211).

[50] Simon Mayer and David S. Karam. "A computational space for the web of things". In: *Proceedings of the Third International Workshop on the Web of Things*. WOT '12. Newcastle, United Kingdom: ACM, 2012, 8:1–8:6. DOI: 10.1145/2379756.2379764 (cit. on pp. 1, 109, 113, 137, 217).

[51] Sonja Meyer and Andreas Ruppen. "An Approach for a Mutual Integration of the Web of Things with Business Processes". In: *Enterprise and Organizational Modeling and Simulation*. Ed. by Joseph Barjis, Ashish Gupta, and Amir Meshkat. Vol. 153. Lecture Notes in Business Information Processing. Springer Berlin Heidelberg, 2013, pp. 42–56. DOI: 10.1007/978-3-642-41638-5_3 (cit. on pp. 137, 232).

[52] Sonja Meyer, Andreas Ruppen, and Carsten Magerkurth. "Internet of Things-aware Process Modeling: Integrating IoT Devices as Business Process Resources". In: *CAiSE 2013*. 2013 (cit. on pp. 110, 122, 125, 232).

[53] Sonja Meyer, Klaus Sperner, Carsten Magerkurth, and Jacques Pasquier. "Towards modeling real-world aware business processes". In: *Proceedings of the Second International Workshop on Web of Things*. WoT '11. San Francisco, California: ACM, 2011, 8:1–8:6. DOI: http://doi.acm.org/10.1145/1993966.1993978 (cit. on pp. 110, 122).

[54] J. Miller and J. Mukerji. *MDA Guide Version 1.0.1*. Tech. rep. omg/03-06-01. Object Management Group (OMG), 06/2003 (cit. on pp. 91, 153).

[55] G. E. Moore. "Cramming More Components onto Integrated Circuits". In: *Electronics* 38.8 (4/19/1965), pp. 114–117. DOI: 10.1109/JPROC.1998.658762 (cit. on p. 1).

[56] Robert Morris, Robert Morris, Ken Thompson, and Ken Thompson. “Password Security: A Case History”. In: *Communications of the ACM* 22 (1979), pp. 594–597 (cit. on p. 57).

[57] Mathias Mueller. “Design and Implementation of a Web-enabled Electronic Product”. This is Mathias Master Thesis. 2009 (cit. on pp. 111, 134, 138).

[58] C.K. Ogden and I. A. Richards. “The Meaning of Meaning: A Study of the Influence of Language Upon Thought and of the Science of Symbolism.” In: *8th ed. 1923. Reprint New York: Harcourt Brace Jovanovich* (1923) (cit. on pp. 92, 95).

[59] OMG. *Meta Object Facility Specification*. Specification Version 1.4. Object Management Group, 2002 (cit. on p. 96).

[60] Benedikt Ostermaier, Fabian Schlup, and Matthias Kovatsch. “Leveraging the web of things for rapid prototyping of UbiComp applications.” In: *UbiComp (Adjunct Papers)*. Ed. by Jakob E. Bardram, Marc Langheinrich, Khai N. Truong, and Paddy Nixon. ACM International Conference Proceeding Series. ACM, 2010, pp. 375–376 (cit. on p. 130).

[61] Cesare Pautasso. “BPMN for REST”. In: *3rd International Workshop on BPMN*. Luzern, Switzerland: Springer, 11/2011 (cit. on pp. 110, 113, 122, 125).

[62] Cesare Pautasso. “RESTful Web service composition with BPEL for REST”. In: *Data and Knowledge Engineering* 68 (2009), pp. 851–866 (cit. on pp. 110, 113, 122, 125).

[63] Cesare Pautasso and Erik Wilde. “Why is the web loosely coupled?: a multi-faceted metric for service design”. In: *Proceedings of the 18th international conference on World wide web*. WWW ’09. New York, NY, USA: ACM, 2009. DOI: http://doi.acm.org/10.1145/1526709.1526832 (cit. on p. 28).

[64] Cesare Pautasso, Olaf Zimmermann, and Frank Leymann. “Restful web services vs. "big"’ web services: making the right architectural decision”. In: *Proceeding of the 17th international conference on World Wide Web*. WWW ’08. New York, NY, USA: ACM, 2008. DOI: http://doi.acm.org/10.1145/1367497.1367606 (cit. on pp. 18, 28, 36, 61, 63).

[65] V. Pimentel and B.G. Nickerson. “Communicating and Displaying Real-Time Data with WebSocket”. In: *Internet Computing, IEEE* 16.4 (07/2012), pp. 45–53. DOI: 10.1109/MIC.2012.64 (cit. on p. 129).

[66] Antonio Pintus, Davide Carboni, and Andrea Piras. “Paraimpu: a platform for a social web of things”. In: *WWW (Companion Volume)*. 2012, pp. 401–404 (cit. on pp. 111 sq.).

[67] Shuping Ran. “A Model for Web Services Discovery with QoS”. In: *SIGecom Exch.* 4.1 (03/2003), pp. 1–10. DOI: 10.1145/844357.844360 (cit. on p. 83).

[68] J. Rothenberg. *Prototyping as modeling: what is being modeled?* Tech. rep. DTIC Document, 1990 (cit. on p. 94).

[69] Andreas Ruppen and Sonja Meyer. "An Approach for a Mutual Integration of the Web of Things with Business Processes". In: *Enterprise and Organizational Modeling and Simulation*. Ed. by Joseph Barjis, Ashish Gupta, and Amir Meshkat. Vol. 153. Lecture Notes in Business Information Processing. Springer Berlin Heidelberg, 2013, pp. 42–56. DOI: 10.1007/978-3-642-41638-5_3 (cit. on pp. 122, 232).

[70] Andreas Ruppen, Jacques Pasquier, and T. Hürlimann. "A RESTful architecture for integrating decomposable delayed services within the web of things". In: *Int. J. Internet Protoc. Technol.* 6.4 (6/2011), pp. 247–259. DOI: 10.1504/IJIPT.2011.047224 (cit. on pp. 1, 120, 137, 232).

[71] Andreas Ruppen, Jacques Pasquier, and Tony Hurlimann. "A RESTful Architecture for Integrating Decomposable Delayed Services within the Web of Things". In: *Parallel and Distributed Systems (ICPADS), 2011 IEEE 17th International Conference on*. 12/2011, pp. 860–865. DOI: 10.1109/ICPADS.2011.10 (cit. on pp. 109 sq., 120, 201, 232).

[72] Andreas Ruppen, Jacques Pasquier, Jean-Frédéric Wagen, Beat Wolf, and Raphael Guye. "A WoT approach to eHealth: case study of a hospital laboratory alert escalation system". In: *Proceedings of the Third International Workshop on the Web of Things*. WOT '12. Newcastle, United Kingdom: ACM, 2012, 6:1–6:6. DOI: 10.1145/2379756.2379762 (cit. on pp. 123 sq., 207, 232).

[73] Silvia Schreier. "Modeling RESTful applications". In: *Proceedings of the Second International Workshop on RESTful Design*. WS-REST '11. Hyderabad, India: ACM, 2011, pp. 15–21. DOI: 10.1145/1967428.1967434 (cit. on pp. 149, 159, 169).

[74] Charles Severance. "Vint Cerf: A Brief History of Packets". In: *Computer* 45.12 (2012), pp. 10–12. DOI: http://doi.ieeecomputersociety.org/10.1109/MC.2012.422 (cit. on p. 10).

[75] Zach Shelby, Klaus Hartke, Carsten Bormann, and Brian Frank. *Constrained Application Protocol (CoAP)*. Tech. rep. draft-ietf-core-coap-07.txt. Instead of using dedicated radio technology and communication protocols that are optimized for low power consumption, such as IEEE 802.15.4, 6LoW-PAN [9], or CoAP [15]. Fremont, CA, USA: IETF Secretariat, 7/8/2011 (cit. on pp. 34, 221).

[76] Rattapoom Tuchinda, Pedro Szekely, and Craig A. Knoblock. "Building Mashups by example". In: *Proceedings of the 13th international conference on Intelligent user interfaces*. IUI '08. Gran Canaria, Spain: ACM, 2008, pp. 139–148. DOI: 10.1145/1378773.1378792 (cit. on pp. 3, 111, 138).

[77] Tomas Vitvar, Jacek Kopecký, Maciej Zaremba, and Dieter Fensel. "WSMO-Lite: Lightweight Semantic Descriptions for Services on the Web." In: *ECOWS*. IEEE Computer Society, 2007, pp. 77–86 (cit. on p. 149).

[78] Joachim W. Walewski. *Initial Architectural Reference Model for IoT*. Tech. rep. The Internet-of-Things Architecture, 4/2011. DOI: `http://www.iot-a.eu/public/public-documents/documents-1/1/1/d1.2/at_download/file` (cit. on pp. 121, 147).

[79] Erik Wilde and U C Berkeley. *Putting Things to REST*. Tech. rep. November. School of Information, UC Berkeley, 2007 (cit. on pp. 63, 219).

Request for Comments

[rfc1436] F. Anklesaria, M. McCahill, P. Lindner, D. Johnson, D. Torrey, and B. Albert. *The Internet Gopher Protocol (a distributed document search and retrieval protocol)*. RFC 1436 (Informational). Internet Engineering Task Force, 03/1993 (cit. on p. 11).

[rfc1630] T. Berners-Lee. *Universal Resource Identifiers in WWW: A Unifying Syntax for the Expression of Names and Addresses of Objects on the Network as used in the World-Wide Web*. RFC 1630 (Informational). Internet Engineering Task Force, 06/1994 (cit. on p. 64).

[rfc1866] T. Berners-Lee and D. Connolly. *Hypertext Markup Language - 2.0*. RFC 1866 (Historic). Obsoleted by RFC 2854. Internet Engineering Task Force, 11/1995 (cit. on p. 10).

[rfc1945] T. Berners-Lee, R. Fielding, and H. Frystyk. *Hypertext Transfer Protocol – HTTP/1.0*. RFC 1945 (Informational). Internet Engineering Task Force, 05/1996 (cit. on pp. 11, 47, 52, 58).

[rfc3986] T. Berners-Lee, R. Fielding, and L. Masinter. *Uniform Resource Identifier (URI): Generic Syntax*. RFC 3986 (INTERNET STANDARD). Updated by RFCs 6874, 7320. Internet Engineering Task Force, 01/2005 (cit. on pp. 47, 54).

[rfc2396] T. Berners-Lee, R. Fielding, and L. Masinter. *Uniform Resource Identifiers (URI): Generic Syntax*. RFC 2396 (Draft Standard). Obsoleted by RFC 3986, updated by RFC 2732. Internet Engineering Task Force, 08/1998 (cit. on p. 58).

[rfc1738] T. Berners-Lee, L. Masinter, and M. McCahill. *Uniform Resource Locators (URL)*. RFC 1738 (Proposed Standard). Obsoleted by RFCs 4248, 4266, updated by RFCs 1808, 2368, 2396, 3986, 6196, 6270. Internet Engineering Task Force, 12/1994 (cit. on p. 54).

[rfc2460] S. Deering and R. Hinden. *Internet Protocol, Version 6 (IPv6) Specification*. RFC 2460 (Draft Standard). Updated by RFCs 5095, 5722, 5871, 6437, 6564,

6935, 6946, 7045, 7112. Internet Engineering Task Force, 12/1998 (cit. on p. 53).

[rfc6455] I. Fette and A. Melnikov. *The WebSocket Protocol*. RFC 6455 (Proposed Standard). Internet Engineering Task Force, 12/2011 (cit. on p. 128).

[rfc1808] R. Fielding. *Relative Uniform Resource Locators*. RFC 1808 (Proposed Standard). Obsoleted by RFC 3986, updated by RFCs 2368, 2396. Internet Engineering Task Force, 06/1995 (cit. on p. 58).

[rfc2616] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. *Hypertext Transfer Protocol – HTTP/1.1*. RFC 2616 (Draft Standard). Obsoleted by RFCs 7230, 7231, 7232, 7233, 7234, 7235, updated by RFCs 2817, 5785, 6266, 6585. Internet Engineering Task Force, 06/1999 (cit. on pp. 27, 48, 52, 58 sq., 70, 168 sq.).

[rfc2617] J. Franks, P. Hallam-Baker, J. Hostetler, S. Lawrence, P. Leach, A. Luotonen, and L. Stewart. *HTTP Authentication: Basic and Digest Access Authentication*. RFC 2617 (Draft Standard). Updated by RFC 7235. Internet Engineering Task Force, 06/1999 (cit. on pp. 55 sq.).

[rfc2069] J. Franks, P. Hallam-Baker, J. Hostetler, P. Leach, A. Luotonen, E. Sink, and L. Stewart. *An Extension to HTTP : Digest Access Authentication*. RFC 2069 (Proposed Standard). Obsoleted by RFC 2617. Internet Engineering Task Force, 01/1997 (cit. on p. 55).

[rfc2518] Y. Goland, E. Whitehead, A. Faizi, S. Carter, and D. Jensen. *HTTP Extensions for Distributed Authoring – WEBDAV*. RFC 2518 (Proposed Standard). Obsoleted by RFC 4918. Internet Engineering Task Force, 02/1999 (cit. on p. 52).

[rfc5023] J. Gregorio and B. de hOra. *The Atom Publishing Protocol*. RFC 5023 (Proposed Standard). Internet Engineering Task Force, 10/2007 (cit. on p. 127).

[rfc6749] D. Hardt. *The OAuth 2.0 Authorization Framework*. RFC 6749 (Proposed Standard). Internet Engineering Task Force, 10/2012 (cit. on p. 58).

[rfc2817] R. Khare and S. Lawrence. *Upgrading to TLS Within HTTP/1.1*. RFC 2817 (Proposed Standard). Updated by RFCs 7230, 7231. Internet Engineering Task Force, 05/2000 (cit. on p. 52).

[rfc1014] Sun Microsystems. *XDR: External Data Representation standard*. RFC 1014. Internet Engineering Task Force, 06/1987 (cit. on p. 20).

[rfc503] N. Neigus and J. Postel. *Socket number list*. RFC 503. Obsoleted by RFC 739. Internet Engineering Task Force, 04/1973 (cit. on p. 21).

[rfc4287] M. Nottingham and R. Sayre. *The Atom Syndication Format*. RFC 4287 (Proposed Standard). Updated by RFC 5988. Internet Engineering Task Force, 12/2005 (cit. on p. 127).

[rfc791] J. Postel. *Internet Protocol*. RFC 791 (INTERNET STANDARD). Updated by RFCs 1349, 2474, 6864. Internet Engineering Task Force, 09/1981 (cit. on p. 53).

[rfc349] J. Postel. *Proposed Standard Socket Numbers*. RFC 349. Obsoleted by RFC 433. Internet Engineering Task Force, 05/1972 (cit. on p. 21).

[rfc433] J. Postel. *Socket number list*. RFC 433. Obsoleted by RFC 503. Internet Engineering Task Force, 12/1972 (cit. on p. 21).

[rfc204] J. Postel. *Sockets in use*. RFC 204. Updated by RFC 234. Internet Engineering Task Force, 08/1971 (cit. on p. 21).

[rfc7252] Z. Shelby, K. Hartke, and C. Bormann. *The Constrained Application Protocol (CoAP)*. RFC 7252 (Proposed Standard). Internet Engineering Task Force, 06/2014 (cit. on pp. 34, 71, 221).

[rfc617] E.A. Taft. *Note on socket number assignment*. RFC 617. Internet Engineering Task Force, 02/1974 (cit. on p. 21).

[rfc147] J.M. Winett. *Definition of a socket*. RFC 147. Internet Engineering Task Force, 05/1971 (cit. on p. 21).

Referenced Web Resources

[WEB1] *A History of Computer Programming Languages*. `http://cs.brown.edu/~adf/programming_languages.html` (accessed Nov 11, 2013) (cit. on p. 23).

[WEB2] *A history of HTML*. `http://www.w3.org/People/Raggett/book4/ch02.html` (accessed Oct 18, 2013) (cit. on p. 13).

[WEB3] *Apache HTTP Server*. `http://httpd.apache.org/` (accessed March 18, 2014) (cit. on p. 85).

[WEB4] *Apple Health Kit*. `https://www.apple.com/ios/whats-new/health/` (accessed 01 26, 2015) (cit. on p. 199).

[WEB5] *Arduino*. `http://www.arduino.cc/` (accessed Oct 15, 2013) (cit. on pp. 1, 39, 107).

[WEB6] *Avoid hackable URLs*. `http://blog.ploeh.dk/2013/05/01/rest-lesson-learned-avoid-hackable-urls/` (accessed July 23, 2014) (cit. on p. 119).

[WEB7] *Celsius*. `http://en.wikipedia.org/wiki/Celsius` (accessed 08 25, 2014) (cit. on p. 134).

[WEB8] *Communication Technologies for the Web of Things*. `http://www.w3.org/community/wot/wiki/Communications_technologies_for_the_Web_of_Things` (accessed July 27, 2014) (cit. on p. 128).

[WEB9] *Damn Vulnerable Web Application*. http://www.dvwa.co.uk/ (accessed Nov 29, 2013) (cit. on p. 54).

[WEB10] *Defacing and Cookie Stealing with Cross-site scripting*. https://ifconfig.dk/cross-site-scripting/ (accessed Nov 29, 2013) (cit. on p. 54).

[WEB11] *Dictionary.com - Entity*. http://dictionary.reference.com/browse/entity (accessed 10 09, 2014) (cit. on p. 150).

[WEB12] *Dictionary.com Unabridged*. http://dictionary.reference.com/browse/methodology (accessed March 25, 2014) (cit. on p. 90).

[WEB13] *Distributing Python Modules*. https://docs.python.org/2/distutils/setupscript.html (accessed 01 20, 2015) (cit. on p. 177).

[WEB14] *Eric Schmidt's Quite Right The Internet Will Disappear*. http://www.forbes.com/sites/timworstall/2015/01/24/eric-schmidts-quite-right-the-internet-will-disappear-all-technologies-do-as-they-mature/ (accessed 02 02, 2015) (cit. on p. 17).

[WEB15] *Fahrenheit*. http://en.wikipedia.org/wiki/Fahrenheit (accessed 08 25, 2014) (cit. on p. 134).

[WEB16] *Free Documentation Licence (GNU FDL)*. http://www.gnu.org/licenses/fdl.txt (accessed October 8, 2013) (cit. on p. 229).

[WEB17] *firmata protocol*. https://github.com/firmata/protocol (accessed 01 21, 2015) (cit. on p. 183).

[WEB18] *fitbit*. https://www.fitbit.com/ (accessed Mai 5, 2014) (cit. on pp. 107, 123).

[WEB19] *Google Fit*. https://fit.google.com/ (accessed 01 26, 2015) (cit. on p. 199).

[WEB20] *HomeKit*. https://developer.apple.com/homekit/ (accessed 01 9, 2015) (cit. on p. 39).

[WEB21] *HyperText Desing Issues: Topology*. http://www.w3.org/DesignIssues/Topology.html (accessed Oct 18, 2013) (cit. on p. 11).

[WEB22] *If This Then That*. https://ifttt.com/ (accessed 03 17, 2015) (cit. on p. 218).

[WEB23] *Internet of Things Architecture*. http://iot-a.eu (accessed Sept 06, 2012) (cit. on p. 33).

[WEB24] *io*. http://iolanguage.org/ (accessed Sept 09, 2014) (cit. on p. 78).

[WEB25] *Japan Geigermap*. http://japan.failedrobot.com/ (accessed June 3, 2014) (cit. on pp. 110, 134).

[WEB26] *Java API for RESTful Services (JAX-RS).* https://jax-rs-spec.java.net/ (accessed 01 7, 2015) (cit. on p. 37).

[WEB27] *Jawbone up.* https://jawbone.com/up (accessed Mai 5, 2014) (cit. on p. 107).

[WEB28] *Kelvin.* http://en.wikipedia.org/wiki/Kelvin (accessed 08 25, 2014) (cit. on p. 134).

[WEB29] *Koubachi - Interactive Plant Care.* http://www.koubachi.com/ (accessed Mai 5, 2014) (cit. on pp. 37, 39, 107 sq.).

[WEB30] *Linked Data.* http://www.w3.org/DesignIssues/LinkedData.html (accessed 02 05, 2015) (cit. on p. 217).

[WEB31] *Mashups — The evolution of the SOA, Part 2.* http://www.ibm.com/developerworks/webservices/library/ws-soa-mashups2/ (accessed Mai 26, 2014) (cit. on p. 111).

[WEB32] *Merriam Webster - Entity.* http://www.merriam-webster.com/dictionary/entity (accessed 10 09, 2014) (cit. on p. 150).

[WEB33] *"Methodology." Encyclopedia Britanica.* http://www.britannica.com/bps/dictionary?query=methodology (accessed March 25, 2014) (cit. on p. 90).

[WEB34] *Mixu's Node Book.* http://book.mixu.net/node/ch6.html (accessed Feb 18, 2014) (cit. on p. 78).

[WEB35] *Mobile internet devices 'will outnumber humans this year'.* http://www.theguardian.com/technology/2013/feb/07/mobile-internet-outnumber-people (accessed 01 05, 2015) (cit. on p. 31).

[WEB36] *Nike+ fuelband.* http://www.nike.com/us/en_us/c/nikeplus-fuelband (accessed Mai 5, 2014) (cit. on pp. 107, 123).

[WEB37] *Observer Effect.* http://en.wikipedia.org/wiki/Observer_effect_(physics) (accessed 10 27, 2014) (cit. on p. 151).

[WEB38] *openPicus.* http://www.openpicus.com/ (accessed Mai 5, 2014) (cit. on p. 107).

[WEB39] *Open.Sen.se.* http://open.sen.se/ (accessed June 3, 2014) (cit. on p. 112).

[WEB40] *Oxford Dictionaries - Entity.* http://www.oxforddictionaries.com/definition/english/entity (accessed 10 09, 2014) (cit. on p. 150).

[WEB41] *Pacif-i Smart Pacifier.* http://bluemaestro.com/product/pacifi-smart-pacifier/ (accessed 01 9, 2015) (cit. on p. 40).

[WEB42] *PageRank Algorithm Patent - US6285999*. http://worldwide.espacenet.com/publicationDetails/biblio?locale=de_EP&CC=US&NR=6285999 (accessed Dec 19, 2013) (cit. on p. 68).

[WEB43] *Paraimpu*. https://www.paraimpu.com/ (accessed June 3, 2014) (cit. on pp. 110 sqq.).

[WEB44] *Philips Hue*. http://www2.meethue.com/ (accessed 01 9, 2015) (cit. on p. 39).

[WEB45] *Philips Hue lights with XBMC and Ambify*. https://www.youtube.com/watch?v=4KxYKsP9MK0 (accessed 01 7, 2015) (cit. on p. 37).

[WEB46] *Philips HUE wakeuplight (speeded up 20 times)*. https://www.youtube.com/watch?v=84IR-gbd8W8 (accessed 01 7, 2015) (cit. on p. 37).

[WEB47] *Princeton University "About WordNet." WordNet. Princeton University*. http://wordnetweb.princeton.edu/perl/webwn?s=methodology (accessed March 25, 2014) (cit. on p. 90).

[WEB48] *programmableweb*. http://www.programmableweb.com/ (accessed Nov 12, 2013) (cit. on p. 25).

[WEB49] *Raspberry Pi*. http://www.raspberrypi.org/ (accessed 01 9, 2015) (cit. on p. 39).

[WEB50] *Remote Method Invocation*. http://de.wikipedia.org/wiki/Remote_Method_Invocation (accessed 02 04, 2015) (cit. on p. 23).

[WEB51] *RSS 2.0 Specification*. http://cyber.law.harvard.edu/rss/rss.html (accessed July 29, 2014) (cit. on p. 127).

[WEB52] *Samsung prototypes brainwave-reading wearable stroke detector*. http://www.cnet.com/news/samsung-prototypes-brainwave-reading-wearable-stroke-detector/ (accessed 01 26, 2015) (cit. on p. 199).

[WEB53] *Schroedinger's cat*. http://en.wikipedia.org/wiki/Schr%C3%B6dinger's_cat (accessed 08 25, 2014) (cit. on p. 134).

[WEB54] *Schroedinger's cat on YouTube*. https://www.youtube.com/watch?v=IOYyCHGWJq4 (accessed 08 25, 2014) (cit. on p. 134).

[WEB55] *Security-Funktion HSTS als Supercookie*. http://www.heise.de/security/artikel/Security-Funktion-HSTS-als-Supercookie-2511258.html?wt_mc=rss.security.beitrag.atom (accessed 01 12, 2015) (cit. on p. 54).

[WEB56] *Self/Welcome*. http://selflanguage.org/ (accessed Sept 09, 2014) (cit. on p. 78).

[WEB57] *Skoda Octavia Prospekte/Preislisten.* http://www.skoda.ch/beratung/prospekte/prospekte-preisliste-online (accessed April 14, 2014) (cit. on p. 100).

[WEB58] *Sun SPOT World.* http://www.sunspotworld.com/ (accessed Oct 15, 2013) (cit. on pp. 1, 107).

[WEB59] *That 'Internet of Things' Thing.* http://www.rfidjournal.com/articles/view?4986 (accessed 01 5, 2015) (cit. on p. 31).

[WEB60] *The 7 Hottest Trends We'll See at This Year's CES.* http://time.com/3651035/ces-international-2015/ (accessed 01 6, 2015) (cit. on pp. 2, 18, 33).

[WEB61] *The Apache ANT Project.* http://ant.apache.org/ (accessed April 29, 2014) (cit. on p. 145).

[WEB62] *There are now more gadgets on Earth than people.* http://www.cnet.com/news/there-are-now-more-gadgets-on-earth-than-people/ (accessed 01 5, 2015) (cit. on p. 31).

[WEB63] *There are officially more mobile devices than people in the world.* http://www.independent.co.uk/life-style/gadgets-and-tech/news/there-are-officially-more-mobile-devices-than-people-in-the-world-9780518.html (accessed 01 5, 2015) (cit. on pp. 1, 31, 217).

[WEB64] *ThingSpeak.* https://thingspeak.com/ (accessed June 3, 2014) (cit. on pp. 110, 112).

[WEB65] *Twisted Matrix Labs.* https://twistedmatrix.com/trac/ (accessed 03 09, 2015) (cit. on p. 178).

[WEB66] *UDOO.* http://www.udoo.org/ (accessed Oct 15, 2013) (cit. on p. 1).

[WEB67] *Understanding quality of service for Web services.* https://www.ibm.com/developerworks/library/ws-quality/ (accessed March 11, 2014) (cit. on p. 83).

[WEB68] *Vera — Smarter Home Control.* http://getvera.com/ (accessed 01 14, 2015) (cit. on p. 34).

[WEB69] *W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures.* http://www.w3.org/TR/xmlschema11-1/ (accessed 12 04, 2014) (cit. on p. 164).

[WEB70] *Warum Kuoni keine Reisen mehr veranstaltet.* http://www.tagesanzeiger.ch/wirtschaft/unternehmen-und-konjunktur/Warum-Kuoni-keine-Reisen-mehr-veranstaltet/story/13204375 (accessed 01 14, 2015) (cit. on p. 29).

[WEB71] *Web Services Architecture*. http://www.w3.org/TR/ws-arch/ (accessed Oct 28, 2013) (cit. on p. 24).

[WEB72] *WebHooks*. http://www.webhooks.org/ (accessed July 31, 2014) (cit. on p. 130).

[WEB73] *Which Social Networks Are Growing Fastest Worldwide?* http://www.emarketer.com/Article/Which-Social-Networks-Growing-Fastest-Worldwide/1009884 (accessed Oct 21, 2013) (cit. on p. 16).

[WEB74] *WolframAlpha*. http://www.wolframalpha.com/ (accessed 10 30, 2014) (cit. on p. 149).

[WEB75] *Xively*. https://xively.com/ (accessed June 3, 2014) (cit. on pp. 110 sqq.).