Serendipitous Web Applications through Semantic Hypermedia

Ruben Verborgh

Doctoral dissertation under the supervision of
prof. dr. Rik Van de Walle
dr. Erik Mannens

Ghent University, Belgium

Any item may be caused at will to select immediately and automatically another. This is the essential feature of the memex. The process of tying two items together is the important thing.

— *Vannevar Bush, As We May Think (1945)*

Preface

A moneymoon won't find me
My head's my only home
With nothing to remind me
But my vinyl wrapped-up soul
— *Ozark Henry, Icon DJ (2002)*

Like all things in life, research is not about the papers or processes, but about the people behind them. They often say it takes a village to write a book; I found out it takes several to successfully finish a PhD. Research is standing on the shoulders of giants, and a few of those giants deserve a special mention for their contribution to this work.

My supervisor Rik has given me the opportunity to work on a PhD among some of the finest researchers in Belgium. I'm thankful for the enlightening discussions we had, and I look forward to learning more from him. My co-supervisor Erik has always been there for me with understanding and support. His empathic and engaged leadership style continues to be an inspiration. Also thanks to Davy, who taught me the art of research and led me on the path of the Semantic Web.

On that path, I met many people, several of whom became friends. Two encounters in particular have profoundly influenced me and my research. When attending Tom's talk in Hyderabad, I couldn't have suspected how many nice projects we would collaborate on. This book would have been a different one without him. Only a month later, we would both meet Seth on a beach in Crete. The ideas we envisioned there that week will one day surely reshape the Web ;-)

At Multimedia Lab, I'm surrounded by helpful colleagues who have become too numerous to name, but here's a tip of the hat to Sam, Miel, Pieter, Anastasia, Tom, Laurens, Dörthe, Pieterjan, Gerald, Hajar, Joachim, Sebastiaan, Glenn, Jan, Steven, Fréderic, Jonas, Peter, and Wesley. A special thanks to Ellen and Laura for their tremendous efforts in keeping everything running smoothly—we all appreciate it!

To all members of the jury, which includes prof. Patrick De Baets, prof. Filip De Turck, dr. Herbert Van de Sompel, prof. Erik Duval, prof. Geert-Jan Houben, prof. Peter Lambert, and Jos De Roo, I wish to express my sincerest gratitude for validating the work in this thesis. I especially thank Jos for all he taught me about logic and the Web, and for his fantastic work on the EYE reasoner—the best is yet to come.

My profound gratitude goes out to Jean-luc Doumont, whose quest for more efficient communication has forever changed the way I write papers and deliver talks. This book's language, structure, and typography wouldn't nearly have been what they are now without "*Trees, maps, and theorems*". Clearer writing makes research usable.

Thanks to the Agency for Innovation by Science and Technology for providing me with a research grant for four wonderful years. Sadly, high-quality education is not a given right for everybody in this world, yet I do believe the Web will play an important role in changing this.

I am very grateful to Vincent Wade for inviting me as a visiting researcher to Trinity College Dublin, and to Alex O'Connor and Owen Conlan who worked with me there. This unique experience allowed me to broaden my professional view in an international context.

Also thank you to many people I had the pleasure to meet over the past few years for shaping my research in one way or another: Rosa Alarcón, Mike Amundsen, Tim Berners-Lee, Peter Brusilovsky, Max De Wilde, Marin Dimitrov, John Domingue, Michael Hausenblas, Jim Hendler, Eelco Herder, Patrick Hochstenbach, Kjetil Kjernsmo, Craig Knoblock, Jacek Kopecký, Markus Lanthaler, Maria Maleshkova, David Martin, Simon Mayer, Sheila McIlraith, Peter Mechant, Barry Norton, Natasha Noy, Pieter Pauwels, Carlos Pedrinaci, Elena Simperl, Nora Srzentić, and Erik Wilde. The discussions I had with you—few or many, short or long—definitely turned me into a better researcher.

Then of course, a warm-hearted thanks to my family and friends. Thanks mom and dad for your lasting encouragement regardless of the choices I make. Niels and Muriel, I'm proud to be your brother. Thanks grandmother, I believe I inherited your sense of perspective.

A special thank you to my great friend Eddy, who told me I could do anything I wanted, as long as I strived to be creative. It worked out!

Finally, I can't say enough how I admire Anneleen for walking the whole way with me. Thank you, my dear, for being everything I'm not.

Ruben
January 2014

Contents

Glossary

API	Application Programming Interface
BPEL	Business Process Execution Language
CSS	Cascading Style Sheets
FOAF	Friend of a Friend
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
ISBN	International Standard Book Number
JPEG	Joint Photographic Experts Group
JSON	JavaScript Object Notation
N3	Notation3
OWL	Web Ontology Language
OWL-S	OWL for Services
PNG	Portable Network Graphics
QR code	Quick Response code
RDF	Resource Description Framework
RDFa	Resource Description Framework in Attributes
RDFS	Resource Description Framework Schema
REST	Representational State Transfer
RPC	Remote Procedure Call
SOAP	Simple Object Access Protocol
SPARQL	SPARQL Protocol and RDF Query Language
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
URN	Uniform Resource Name
WADL	Web Application Description Language
WSDL	Web Service Description Language
WSMO	Web Service Modeling Ontology
W3C	World Wide Web Consortium
XML	Extensible Markup Language

Summary

Ever since its creation at the end of the 20^{th} century, the Web has profoundly shaped the world's information flow. Nowadays, the Web's consumers no longer consist of solely people, but increasingly of machine clients that have been instructed to perform tasks for people. Lacking the ability to interpret natural language, machine clients need a more explicit means to decide what steps they should take. This thesis investigates the obstacles for machines on the current Web, and provides solutions that aim to improve the autonomy of machine clients. In addition, we will enhance the Web's linking mechanism for people, to enable serendipitous reuse of data between Web applications that were not connected previously.

The Web was not the first hypermedia system, and many earlier alternatives had more complex features, especially with regard to content interlinking. However, the Web was the first system to scale globally. Achieving this required sacrificing more complex features: the Web only offers *publisher-driven, one-directional hyperlinks*, a crucial design choice that stimulated its growth into the world's leading information platform. It did not take long before application development using the Web began, first in a way that resembled traditional remote programming, and later in ways that embraced the Web's nature as a distributed hypermedia system.

In order to understand the Web's properties for software development, the Representational State Transfer (REST) architectural style was created, capturing the constraints that govern the Web and other distributed hypermedia systems. A subset of these constraints describe the *uniform interface*, which enable architectural properties such as the independent evolution of clients and servers. The Web's Hypertext Transfer Protocol (HTTP) implements the uniform interface by providing a limited, standardized set of methods to access and manipulate any resource in any Web application. Furthermore, the *hypermedia constraint* demands that hypermedia drives the interaction: clients should follow links and forms rather than engage in a preprogrammed interaction pattern.

Combining the hypermedia constraint with the limitation that the Web's links can only be created by the information publisher, we arrive at what we've called the *affordance paradox*: the client depends on links supplied by the information publisher, which does not precisely know the intentions of the client. Consequently, hypermedia can only serve as the engine of application state to the extent that the hypermedia document affords the actions the client wants to perform. If a certain Web application does not link to a desired action in another application, that action cannot be executed through hypermedia. This currently necessitates hard-coded knowledge about both applications, which endangers the independent evolution of clients and servers.

In order to solve this issue, we first need a way for machines to interpret the effect of actions. The *Semantic Web* is a layer on top of the existing Web that provides machine-interpretable markup. It allows content publishers to annotate their existing data in a way that enables intelligent machine processing. While several efforts have also looked at describing *dynamic* aspects, there is currently no method to rigorously capture the semantics of Web Application Programming Interfaces (APIS) that conform to the REST constraints. This prompted us to create RESTdesc, a description format that explains the functionality of an API by capturing it into first-order logic rules. A RESTdesc description indicates which HTTP request allows the transition from certain preconditions to related postconditions. In contrast to classical Web API descriptions, RESTdesc is designed to support hypermedia-driven interactions at runtime instead of imposing a hard-wired plan at compile-time.

As hypermedia documents allow clients to look ahead only a single step at a time, it is necessary to provide a planning strategy that enables reaching complex goals. RESTdesc rules are expressed in the Notation3 (N3) language, so regular N3 reasoners can compose RESTdesc descriptions into a plan. This is enabled by their built-in *proof* mechanism, which explains how a certain goal can be reached by applying rules as inferences. This proof also guarantees that, if the execution of the Web APIS happens as described, the composition satisfies the given goal. The performance of current N3 reasoners is sufficiently high to find the necessary Web APIS and compose them in realtime. Furthermore, the proposed mechanism allows the automated consumption of Web APIS, guided by a proof but still driven by hypermedia, which enables dynamic interactions.

The automated understanding of actions and the ability to find actions that match a certain context allow us to solve the Web's affordance paradox. Instead of the current linking model, in which the affordance on a certain resource is provided by the party that created this resource, we can collect affordance from distributed sources. With our proposed solution, called *distributed affordance*, a platform dynamically adds hypermedia controls by automatically matching a list of preferred actions to semantic annotations of the content.

For instance, users can have a set of actions they would like to perform on movies, such as finding reviews or downloading them to their digital television. A distributed affordance platform in their browser can automatically make those actions available every time a movie title appears on a page they visit. This removes the limitation of having to rely on links supplied by the information publisher. Especially on mobile devices, which have a more limited set of input controls, such direct links can greatly enhance people's browsing experience. Furthermore, machine clients need not be preprogrammed to use resources from one applications in another, as they can rely on the generated links.

This leads to a more serendipitous use of data and applications on the Web, in which data can flow freely between applications. Similar to how people discover information on the Web by following links, automated agents should be able to perform tasks they have not been explicitly preprogrammed for. Thereby, they gradually become *serendipitous applications*. In addition to *autonomous agents* that act as personal assistants, we envision two other opportunities. *Semantics-driven applications* are able to perform a specific service on any given Linked Data stream, regardless of how it is structured. *Client-side querying* improves scalability and fosters serendipity by moving the intelligence from the server to the client.

The conclusion is that semantic technologies combined with hypermedia allow a new generation of applications that are more reusable across different contexts. Although it remains a challenge to convince information and API publishers of their benefits, semantic annotations significantly improve the opportunities for autonomous applications on the Web.

Chapter **1**

Introduction

Like the fool I am and I'll always be
I've got a dream
They can change their minds
But they can't change me

— Jim Croce, I've Got a Name (1973)

If you ask me, the World Wide Web has been the most important invention of the past decades. Never before in human history have we seen a faster spread of information throughout the entire world. At the dawn of the 21st century, humans are no longer the only information consumers: increasingly, automated software clients try to make sense of what's on the Web. This thesis investigates the current obstacles and catalysts on the road toward a unified Web for humans and machines. It then explores how such a symbiosis can impact the role of the Web for people.

During three and a half years of research, I have been investigating how one day, autonomous pieces of software might use the Web similar to the way people can. This was inspired by Tim Berners-Lee's vision of the Semantic Web [1], a layer on top of the existing Web that makes it interpretable for so-called *intelligent agents*. At one of the first conferences I attended, a keynote talk by Jim Hendler, co-author of the Semantic Web vision article, left me rather puzzled. Near the end of his talk—after convincing us all that the necessary technology is already out there—he posed the question: *"so where are the agents?"*

Is the search for intelligent agents the ultimate goal of the Semantic Web, or is it just a story to explain its potential? In any case, the idea of autonomous personal digital assistants exerts a strong attraction.

More than a decade of Semantic Web research unquestionably resulted in great progress, but nothing that resembles the envisioned intelligent agents is available. The Web has rapidly evolved, and many automated clients were created—yet all of them are preprogrammed for specific tasks. The holy grail of semantic technologies remains undiscovered, and researchers are sceptical as to whether it exists. The unbounded enthusiasm gradually makes place for pragmatism, as with any technology that advances on the hype cycle [3].

I had to maintain a realistic viewpoint during my search for solutions: trying to solve every possible challenge for autonomous agents would result in disappointment. The Semantic Web remains just a technology—albeit one that is assumed to make intelligent applications on the Web easier than its predecessors [2]. However, I believe the techniques discussed in this thesis advances the state of the art by making certain autonomous interactions possible that were significantly more difficult to achieve before. It cannot be a definitive answer to the quest for intelligent agents, but it might offer one of the stepping stones toward more autonomy for such agents.

Along the way, I will question some of the established principles and common practices on the Web. In particular, I will examine how we currently approach software building for the Web and plea for several changes that can make it more accessible for machines. As semantic technologies were never meant to be disruptive, the presented methods allow a gradual transition, backward-compatible with the existing Web infrastructure.

This thesis is structured in 8 chapters. After this introductory chapter, I will discuss the following topics:

- **Chapter 2 – Hypermedia** introduces the evolution of hypertext and hypermedia into the current Web. We detail how the REST architectural style has influenced the Web and examine why the Web's current hypertext design is insufficient to support autonomous agents. This leads to the three research questions that drive this thesis.

- **Chapter 3 – Semantics** sketches the main components of Semantic Web technology and zooms in on the vision of intelligent agents. We discuss Linked Data as a pragmatic view on semantics. Finally, we elaborate on the relation between hypermedia and semantics on the Web.

- **Chapter 4 – Functionality** argues that machine clients need a way to predict the effect of actions on the Web. It introduces my work on RESTdesc, a lightweight approach to capture the functionality of Web APIs. A hypermedia-driven process for agents can offer an alternative to predetermined and rigid interaction patterns.

- **Chapter 5 – Proof** discusses my work on goal-driven Web API composition and the importance of proof and trust in the context of autonomous agents. We reconcile the error-prone act of API execution with the strictness of first-order logic and proofs.

- **Chapter 6 – Affordance** addresses an issue with the Web's linking model: information publishers are responsible for link creation, yet they have insufficient knowledge to provide exactly those links a specific client needs. I introduce the concept of distributed affordance to generate the needed links in a personalized way.

- **Chapter 7 – Serendipity** questions the current way of Web application development. It proposes the use of semantic hypermedia as an enabling mechanism for applications that adapt to a specific client and problem context.

- **Chapter 8 – Conclusion** reviews the content of the preceding chapters, recapitulating the answers to the research questions that guide this thesis.

This thesis has been conceived as a book with a narrative, preferring natural language over mathematical rigorousness to the extent possible and appropriate. The underlying motivation is to make this work more accessible, while references to my publications guide the reader towards in-depth explanations.

Since no act of research ever happens in isolation, I will use the authorial "we" throughout the text, except in places where I want to emphasize my own viewpoint.

I hope this introduction may be the start of a fascinating journey through hypermedia and semantics. I learned a lot while conducting this research; may the topics in this book inspire you in turn.

References

[1] Tim Berners-Lee, James Hendler, and Ora Lassila. The Semantic Web. *Scientific American*, 284(5):34–43, May 2001.

[2] Christian Bizer, Tom Heath, and Tim Berners-Lee. Linked Data – the story so far. *International Journal on Semantic Web and Information Systems*, 5(3):1–22, March 2009.

[3] Jackie Fenn and Mark Raskino. Understanding Gartner's hype cycles. Technical report. 2 July 2013. *http://www.gartner.com/id=2538815*

Chapter **2**

Hypermedia

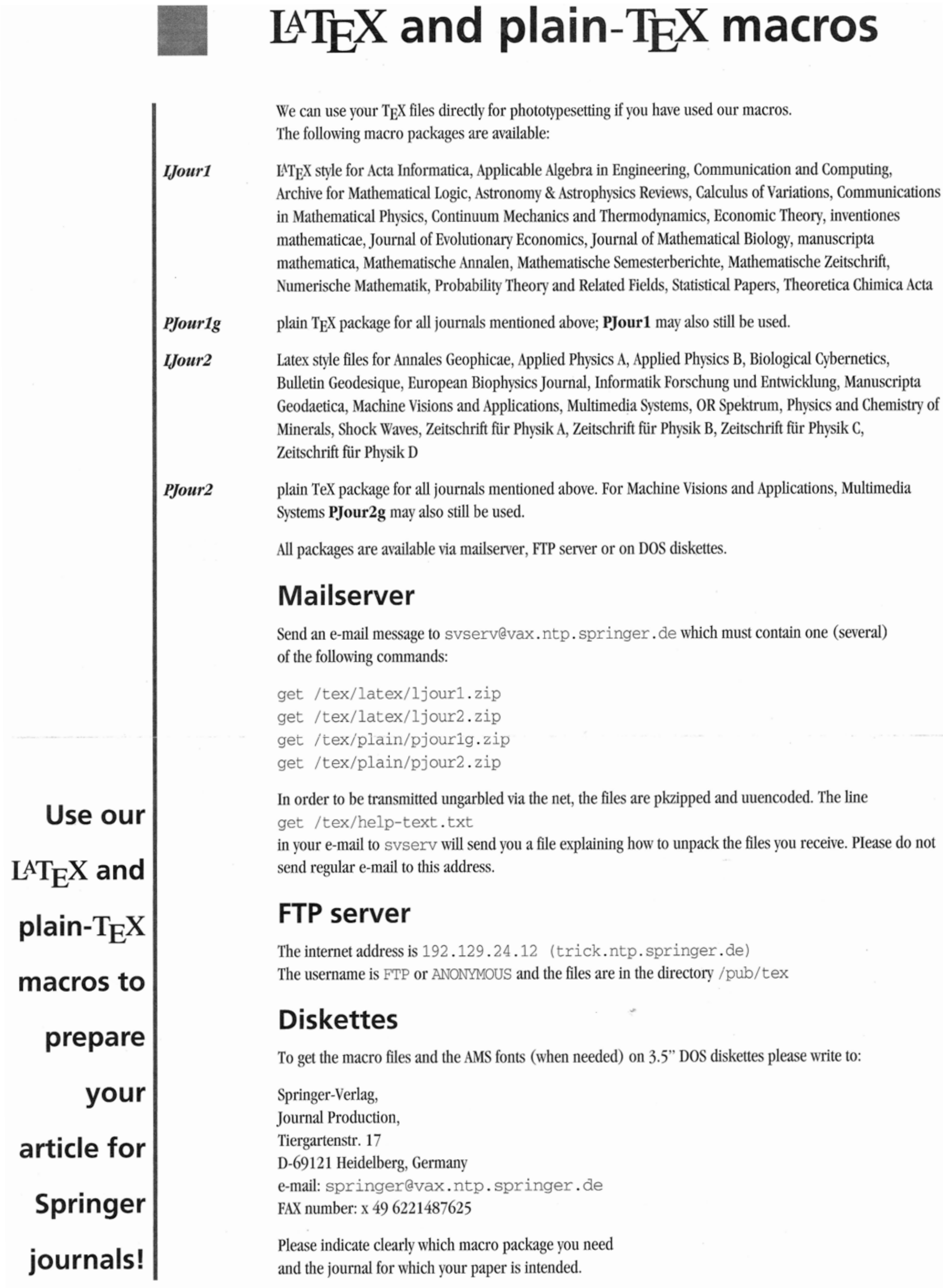
J'irai où ton souffle nous mène
Dans les pays d'ivoire et d'ébène

— *Khaled, Aïcha (1996)*

Hypermedia plays a fundamental role on the Web. Unfortunately, in order to turn the Web into the global hypertext system it is today, several compromises had to be made. This chapter starts with a short history of hypermedia, moving on to its implementation on the Web and its subsequent conceptualization through REST. If we want machines to automatically perform tasks on the Web, enhancements are necessary. This observation will lead to the research questions guiding this doctoral dissertation.

In colloquial language, "the Internet" and "the Web" are treated synonymously. In reality, the *Internet* [9] refers to the international computer network, whereas the *World Wide Web* [5] is an information system, running on top of the Internet, that provides interconnected documents and services. As many people have been introduced to both at the same time, it might indeed be unintuitive to distinguish between the two. An interesting way to understand the immense revolution the Web has brought upon the Internet is to look at the small time window in the early 1990s when the Internet had started spreading but the Web hadn't yet. The flyer in Figure 1 dates from this period, and was targeted at people with a technical background who likely had Internet access. It instructs them to either send an e-mail with commands, or connect to a server and manually traverse a remote file system in order to obtain a set of documents. In contrast,

While the information system is actually the *Web*, in practice, people usually refer to any action they perform online simply as "using the *Internet*".



LaTeX and plain-TeX macros

We can use your TeX files directly for phototypesetting if you have used our macros.
The following macro packages are available:

LJour1 LaTeX style for Acta Informatica, Applicable Algebra in Engineering, Communication and Computing, Archive for Mathematical Logic, Astronomy & Astrophysics Reviews, Calculus of Variations, Communications in Mathematical Physics, Continuum Mechanics and Thermodynamics, Economic Theory, inventiones mathematicae, Journal of Evolutionary Economics, Journal of Mathematical Biology, manuscripta mathematica, Mathematische Annalen, Mathematische Semesterberichte, Mathematische Zeitschrift, Numerische Mathematik, Probability Theory and Related Fields, Statistical Papers, Theoretica Chimica Acta

PJour1g plain TeX package for all journals mentioned above; **PJour1** may also still be used.

LJour2 Latex style files for Annales Geophicae, Applied Physics A, Applied Physics B, Biological Cybernetics, Bulletin Geodesique, European Biophysics Journal, Informatik Forschung und Entwicklung, Manuscripta Geodaetica, Machine Visions and Applications, Multimedia Systems, OR Spektrum, Physics and Chemistry of Minerals, Shock Waves, Zeitschrift für Physik A, Zeitschrift für Physik B, Zeitschrift für Physik C, Zeitschrift für Physik D

PJour2 plain TeX package for all journals mentioned above. For Machine Visions and Applications, Multimedia Systems **PJour2g** may also still be used.

All packages are available via mailserver, FTP server or on DOS diskettes.

Use our LaTeX and plain-TeX macros to prepare your article for Springer journals!

Mailserver

Send an e-mail message to svserv@vax.ntp.springer.de which must contain one (several) of the following commands:

```
get /tex/latex/ljour1.zip
get /tex/latex/ljour2.zip
get /tex/plain/pjour1g.zip
get /tex/plain/pjour2.zip
```

In order to be transmitted ungarbled via the net, the files are pkzipped and uuencoded. The line
`get /tex/help-text.txt`
in your e-mail to svserv will send you a file explaining how to unpack the files you receive. Please do not send regular e-mail to this address.

FTP server

The internet address is 192.129.24.12 (trick.ntp.springer.de)
The username is FTP or ANONYMOUS and the files are in the directory /pub/tex

Diskettes

To get the macro files and the AMS fonts (when needed) on 3.5" DOS diskettes please write to:

Springer-Verlag,
Journal Production,
Tiergartenstr. 17
D-69121 Heidelberg, Germany
e-mail: springer@vax.ntp.springer.de
FAX number: x 49 6221487625

Please indicate clearly which macro package you need
and the journal for which your paper is intended.

Figure 1: When usage of the Web was not widespread, Internet documents had to be retrieved by following a list of steps instead of simply going to an address. This flyer from around 1993 instructs Internet users how to retrieve files by sending commands via e-mail or by manually traversing a server.

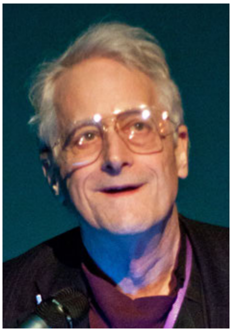
the Web allows publishers to simply print an address that can be typed into a Web browser. This address could point to an information page that contains links to the needed documents. Nowadays, it is even common to have a machine-readable QR code on such a flyer, so devices like smartphones can follow the "paper link" autonomously.

This QR code leads you to the page at *springer.com* where the documents of Figure 1 can be retrieved. Compare the simplicity of scanning the code or typing that link to following the figure's instructions.

Clearly, we've come a long way. This chapter will tell the history of hypermedia and the Web through today's eyes, with a focus on what is still missing for current and future applications. These missing pieces form the basis for this dissertation's research questions, which are formulated at the end of the chapter. The next section takes us back in time for a journey that surprisingly already starts in 1965—when the personal computer revolution was yet to begin.

A history of hypermedia

The first written mention of the word **hypertext** was by Ted Nelson in a 1965 article [15], where he had introduced it as *"a body of written or pictorial material interconnected in such a complex way that it could not conveniently be presented or represented on paper."* In that same article, he mentioned **hypermedia** as a generalization of the concept to other media such as movies (consequently called **hyperfilm**). While this initial article was not very elaborate on the precise meaning of these terms, his infamous cult double-book "Computer Lib / Dream Machines" [16] provided context and examples to make them more clear. Later, in "Literary Machines" [17], he defined hypertext as *"[…] non-sequential writing—text that branches and allows choices to the reader, best read at an interactive screen. As popularly conceived, this is a series of text chunks connected by links which offer the reader different pathways."*



Theodor Holm Nelson (*1937) is a technology pioneer most known for Project Xanadu [23]. Even though a fully functional version has not been released to date, it inspired generations of hypertext research. When coining the term, he wrote *"we've been speaking hypertext all our lives and never known it."* [16] ©Daniel Gies

It is important to realize that Nelson's vision differs from the Web's implementation of hypertext in various ways. He envisioned *chunk-style hypertext* with footnotes or labels offering choices that came to the screen as you clicked them, *collateral hypertext* to provide annotations to a text, *stretchtext*, where a continuously updating document could contain parts of other documents with a selectable level of detail, and *grand hypertext*, which would consist of everything written about a subject [16]. In particular, Nelson thought of much more flexible ways of interlinking documents, where links could be multi-directional and created by any party, as opposed to the uni-directional, publisher-driven links of the current Web. Information could also be intertwined with other pieces of content, which Nelson called *transclusion*.

The idea of interlinking documents even predates Nelson. Vannevar Bush wrote his famous article "As We May Think" in 1945, detailing a hypothetical device the *memex* [8], that enabled researchers to follow a complex trail of documents… *on microfilm*. That idea can in turn be traced back to Paul Otlet, who imagined a mesh of many *electric telescopes* already in 1934 [18]. While unquestionably brilliant, both works now read like anachronisms. They were onto something crucial, but the missing piece would only be invented a few decades later: the *personal computer*.

Although Nelson's own hypertext platform *Xanadu* was never realized [23], other computer pioneers such as Doug Engelbart started to implement various kinds of hypertext software. By 1987, the field had sufficiently matured for an extensive survey, summarizing the then-existing hypertext systems [10]. Almost none of the discussed systems are still around today, but the concepts presented in the article sound familiar. The main difference with the Web is that all these early hypermedia systems were **closed**. They implemented hypermedia in the sense that they presented information on a screen that offered the user choices of where to go next. These choices, however, were limited to a local set of documents. In the article, Conklin defines the concept of hypertext as *"windows on the screen […] associated with objects in a database"* [10], indicating his presumption that there is indeed a single database containing all objects. Those systems are thus closed in the sense that they cannot cross the border of a single environment, and, as a consequence, also in the sense that they cannot access information from systems running different software.

As a result, hypertext systems were rather small: documentation, manuals, books, topical encyclopedias, personal knowledge bases, … In contrast, Nelson's vision hinted at a *global* system, even though he did not have a working implementation by the end of the 1980s, when more than a dozen other hypertext systems were already in use. The focus of hypertext research at the time was on adding new features to existing systems. In hindsight, it seems ironic that researchers back then literally didn't succeed in thinking "outside the box".

The invention of the Web

His World Wide Web was only accepted as a demo in 1991 [4]. Yet at the 1993 Hypertext conference, all projects were somehow connected to the Web, as Tim Berners-Lee recalls.

Looking through the eyes of that time, it comes as no surprise that Tim Berners-Lee's invention was not overly enthusiastically received by the 1991 *Hypertext* conference organizers [4]. The **World Wide Web** [5] looked very basic on screen (only text with links), whereas other systems showed interactive images and maps. But in the end, the **global scalability** of the Web turned out to be more important than the bells and whistles of its competitors. It quickly turned the Web into the most popular application of the Internet. Nearly all other hypertext research was halted, with several researchers switching to Web-related topics such as *Web engineering* or *Semantic Web (Chapter 3)*. Remaining core hypermedia research is now almost exclusively carried out within the field of *adaptive hypermedia (Chapter 6)*.

The Web owes its success to its architecture, which was designed to scale globally. Therefore, it is crucial to have a closer look at the components that make up the invention.

The Web's components

The Web is not a single monolithic block, but rather a combination of three core components, each of which is discussed below.

Uniform Resource Locator (URL) A URL [7] has double functionality. On the one hand, it uniquely *identifies* a resource, just like a national identification number identifies a person. On the other hand, it also *locates* a resource, like a street address allows to locate a person. However, note that both functions are clearly distinct: a national identification number doesn't tell you where a person lives, and a street address doesn't always uniquely point to a single person. URLS provide both identification and location at the same time, because they are structured in a special way. The *domain name* part of a URL allows the browser to locate the server on the Internet, and the *path* part gives the server-specific name of the resource. Together, these parts uniquely identify—and locate—each resource on the Internet.

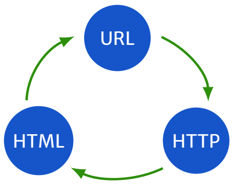
URLS start with a *protocol* (`http` or `https`), followed by a *domain name* to identify the server and a *path* to identify the resource on the server.

Hypertext Transfer Protocol (HTTP) Web clients and servers communicate through the standardized protocol HTTP [13]. This protocol has a simple request/response message paradigm. Each HTTP request consists of the method name of the requested action and the URL to the resource that is the subject of this action. The set of possible methods is limited, and each method has highly specific semantics. A client asks for a representation of a resource (or a manipulation thereof), and the server sends an HTTP response back in which the representation is enclosed.

All HTTP requests and responses contain metadata in standardized headers. For instance, a client can indicate its version, and a server can specify the resource's creation date.

Hypertext Markup Language (HTML) Finally, the Web also needs a language to mark up hypertext, which is HTML [3]. Although HTML is only one possible representation format—as *any* document type can be transported by HTTP—its native support for several hypermedia controls [1] makes it an excellent choice for Web documents. HTML documents can contain links to other resources, which are identified by their URL. Upon activation of the link, the browser *dereferences* the URL by locating, downloading, and displaying the document.

These three components are strongly interconnected. URLS are the identification and location mechanism used by HTTP to manipulate resources and to retrieve their representations. Many resources have an HTML representation (or another format with hypermedia support) that in turn contains references to other resources through their URL.



A URL identifies a resource in an HTTP request, which returns an HTML representation that links to other resources through URLS.

The Web's architectural principles

Like many good inventions, the Web somehow happened by accident. That's not to say that Berners-Lee did not deliberately design URLs, HTTP, and HTML as they are—it's that the formalization and analysis of the Web's architectural principles had not been performed back then. To this end, Roy Thomas Fielding introduced a conceptual framework capable of analyzing large-scale distributed hypermedia systems like the Web, which he called the **Representational State Transfer (REST)** architectural style [11, 14]. REST is a tool to understand the architectural properties of the Web and a guide to maintain these properties when developing future changes or additions.

HTTP, the protocol of the Web, is not the only implementation of REST. And unfortunately, not every HTTP application necessarily conforms to all REST constraints. Yet, full adherence to these constraints is necessary in order to inherit all desirable properties of the REST architectural style.

Fielding devises REST by starting from a system without defined boundaries, iteratively adding *constraints* to induce desired properties. In particular, there's a focus on the properties *scalability*, allowing the Web to grow without negative impact on any of the involved actors, and *independent evolution of client and server*, ensuring interactions between components continue to work even when changes occur on either side. Some constraints implement widely understood concepts, such as the **client-server** constraints and the **cache** constraints, which won't be discussed further here. Two constraints are especially unique to REST (and thus the Web), and will play an important role in the remainder of this thesis: the **statelessness** constraint and the **uniform interface** constraints.

The statelessness constraint

When a client is sending "give me the next page", the interaction is *stateful*, because the server needs the previous message to understand what page it should serve. In contrast, "give me the third page of search results for '`apple`'" is *stateless* because it is fully self-explanatory—at the cost of a substantially longer message length.

REST adds the constraint that the client-server interaction must be **stateless**, thereby inducing the properties of *visibility*, *reliability*, and *scalability* [11]. This means that every request to the server must contain all necessary information to process it, so its understanding does not depend on previously sent messages. This constraint is often loosely paraphrased as "the server doesn't keep state," seemingly implying that the client can only perform read-only operations. Yet, we all know that the Web does supports many different kinds of write operations: servers do remember our username and profile, and let us add content such as text, images, and video. Somehow, there exists indeed a kind of state that is stored by the server, even though this constraint seems to suggest the contrary. This incongruity is resolved by differentiating between two kinds of state: **resource state** and **application state** [19]. Only the former is kept on the server, while the latter resides inside the message body (and partly at the client).

Before we explain the difference, we must first obtain an understanding of what exactly constitutes a **resource**. Resources are the fundamental unit for information in REST. Broadly speaking, *"any information that can be named can be a resource"* [11]. In practice, the resources of a particular Web application are the conceptual pieces of information exposed by its server. Note the word "conceptual" here; resources identify constant *concepts* instead of a concrete value that represents a concept at a particular point in time. For instance, the resource *"today's weather"* corresponds to a different value every day, but the way of mapping the concept to the value remains constant. A resource is thus never equal to its value; *"today's weather"* is different from an HTML page that details this weather. For the same reason, *"The weather on February 28th, 2014"* and *"today's weather"* are distinct concepts and thus different resources—even if 28/02/2014 were today.

The idea behind REST is to define resources at the application's domain level. This means that technological artefacts such as "a service" or "a message" are *not* resources of a book store application. Instead, likely resource candidates are "book", "user profile", and "shopping basket".

Resource state, by consequence, is thus the combined state of all different resources of an application. This state *is* stored by the server and thus *not* the subject of the statelessness constraint. Given sufficient access privileges, the client can view and/or manipulate this state by sending the appropriate messages. In fact, the reason the client interacts with the server is precisely to view or modify resource state, as these resources are likely not available on the client side. This is why the client/server paradigm was introduced: to give a client access to resources it does not provide itself.

In a book store, resource state would be the current contents of the shopping basket, name and address of the user, and the items she has bought.

Application state, in contrast, describes where the client is in the interaction: what resource it is currently viewing, what software it is using, what links it has at its disposition, ... It is *not* the server's responsibility to store this. As soon as a request has been answered, the server should not remember it has been made. This is what makes the interaction scalable: no matter how many clients are interacting with the server, each of them is responsible for maintaining its own application state. When making a request, the client sends the relevant application state along. Part of this is encoded as metadata of each request (for example, HTTP headers with the browser version); another part is implicitly present through the resource being requested. For instance, if the client requests the fourth page of a listing, the client must have been in a state where this fourth page was accessible, such as the third page. By making the request for the fourth page, the server is briefly reminded of the relevant application state, constructs a response that it sends to the client, and then forgets the state again. The client receives the new state and can now continue from there. The uniform interface, which is the next constraint we'll discuss, provides the means of achieving statelessness in REST architectures.

The book the user is consulting and the credentials with which she's signed in are two typical examples of application state.

A *back* button that doesn't allow you to go to your previous steps is an indication the *server* maintains the application state, in violation of the statelessness constraint.

The uniform interface constraints

The central distinguishing feature of the REST architectural style is its emphasis on the **uniform interface**, consisting of four constraints, which are discussed below. Together, they provide *simplification*, *visibility*, and *independent evolution* [11].

In some Web applications, we can see *actions* such as `addComment` as the target of a hyperlink. However, these are not resources according to the definition: an "`addComment`" is not a concept. As an unfortunate consequence, their presence thus breaks compatibility with REST.

Identification of resources Since a *resource* is the fundamental unit of information, each resource should be uniquely identifiable so it can become the target of a hyperlink. We can also turn this around: any indivisible piece of information that can (or should) be identified in a unique way is one of the application's resources. Since resources are conceptual, things that cannot be digitized (such as persons or real-world objects) can also be part of the application domain—even though they cannot be transmitted electronically.

Each resource can be identified by several *identifiers*, but each identifier must not point to more than one resource. On the Web, the role of unique identifiers is fulfilled by URLS, which identify resources and allow HTTP to locate and interact with them.

The common human-readable representation formats on the Web are HTML and plaintext. For machines, JSON, XML, and RDF can be found, as well as many binary formats such as JPEG and PNG.

Manipulation of resources through representations Clients never access resources directly in REST systems; all interactions happen through *representations*. A representation represents the *state* of a resource—which is conceptual in nature—as a byte sequence in a format that can be chosen by the client or server (hence the acronym REST or "*REpresentational State* Transfer"). Such a format is called a *media type*, and resources can be represented in several media types. A representation consists of the actual data, and metadata describing this data. On the Web, this metadata is served as HTTP headers [13].

Out-of-band information is often found in software applications or libraries, an example being human-readable documentation. It increases the difficulty for clients to interoperate with those applications. Media types are only part of the solution.

Self-descriptive messages Messages exchanged between clients and servers in REST systems should not require previously sent or out-of-band information for interpretation. One of the aspects of this is *statelessness*, which we discussed before. Indeed, messages can only be self-descriptive if they do not rely on other messages. In addition, HTTP also features *standard methods* with well-defined semantics (`GET`, `POST`, `PUT`, `DELETE`, . . .) that have properties such as safeness or idempotence [13]. However, in Chapter 4, we'll discuss when and how to attach more specific semantics to the methods in those cases where the HTTP specification deliberately leaves options open.

Hypermedia as the engine of application state The fourth and final constraint of the uniform interface is that hypermedia must be the engine of application state. It is sometimes referred to by its HATEOAS acronym; we will use the term "hypermedia constraint". From the statelessness constraint, we recall that *application state* describes the position of the client in the interaction. The present constraint demands that the interaction be driven by information *inside* server-sent hypermedia representations [12] rather than *out-of-band* information, such as documentation or a list of steps, which would be the case for Remote Procedure Call (RPC) interactions [20]. Concretely, REST systems must offer hypermedia representations that contain the controls that allow the client to proceed to next steps. In HTML representations, these controls include links, buttons, and forms; other media types offer different controls [1].

In REST Web applications, clients advance the application state by activating hypermedia controls. Page 3 of a search result is retrieved by following a link, not by constructing a new navigation request from scratch.

With the history of hypermedia in mind, this constraint seems very natural, but it is crucial to realize its importance and necessity, since the Web only has publisher-driven, one-directional links. When we visit a webpage, we indeed expect the links to next steps to be there: an online store leads to product pages, a product page leads to product details, and this page in turn allows to order the product. However, we've all been in the situation where the link we needed wasn't present. For instance, somebody mentions a product on her homepage, but there is no link to buy it. Since Web linking is unidirectional, there is no way for the store to offer a link from the homepage to the product, and hence, no way for the user to complete the interaction in a hypermedia-driven way. Therefore, the presence of hypermedia controls is important.

Many—if not most—Web APIS that label themselves as "REST" or "RESTful" fail to implement the hypermedia constraint and are thus merely HTTP APIS, not REST APIS. Part of the ignorance might be due to Fielding's only brief explanation of this constraint in his thesis [11]. As he later explained on his blog [12], where he criticized the incorrect usage of the "REST" label, this briefness was because of a lack of time; it does not imply the constraint would be less important than others. To make the distinction clear, Web APIS that conform to all REST constraints are sometimes referred to as *hypermedia APIS* [2].

While humans excel in finding alternative ways to reach a goal (for instance, entering the product name in a search engine and then clicking through), machine clients do not. These machine clients are generally pieces of software that aim to bring additional functionality to an application by interacting with a third-party Web application, often called a Web API (Application Programming Interface) in that context. According to the REST constraints, separate resources for machines shouldn't exist, only *different representations*. Machines thus access the same resources through the same URLS as humans. In practice, many representations for machine clients unfortunately do not contain hypermedia controls. As machines have no flexible coping strategies, they have to be rigidly preprogrammed to interact with Web APIS in which hypermedia is *not* the engine of application state. If we want machines to be flexible, the presence of hypermedia controls is a necessity, surprisingly even *more* than for human-only hypertext.

Hypermedia on the Web

Fielding coined his definition of hypertext only in April 2008, several years after the derivation of REST, in a talk titled *"A little REST and relaxation"*. Yet, its significance is important.

Fielding's definition of hypertext [11] (and by extension, hypermedia) guides us to an understanding of the role of hypermedia on the Web:

> *When I say hypertext, I mean the simultaneous presentation of information and controls such that the information becomes the affordance through which the user (or automaton) obtains choices and selects actions.* — *Roy Thomas Fielding*

As this definition is very information-dense, we will interpret the different parts in more detail.

First, the definition mentions the **simultaneous presentation of information and controls**. This hints at the use of formats that intrinsically support hypermedia controls, such as HTML, where the presentation of the information is necessarily simultaneous with the controls because they are intertwined with each other. However, intertwining is not a strict requirement; what matters is that the client has access to the information and to the controls that drive the application state at the same time.

Second, by their presence, these controls **transform the information into an affordance**. As the precise meaning and significance of the term *affordance* will be clarified in Chapter 6, it suffices here to say that the controls make the information *actionable*: what previously was only text now provides its own interaction possibilities.

Third, these interaction possibilities allow humans and machine clients to **choose and select actions**. This conveys the notion of Nelson's definition that the text should allow choices to the reader on an interactive screen [17]. Additionally, it refers to the hypermedia constraint, which demands the information contains the controls that allow the choice and selection of next steps.

Nowadays, most machine clients have been preprogrammed for interaction with a limited subset of Web APIS. However, I do expect this to change in the future—and I aim to contribute to that change with the work described in this thesis.

Note how the definition explicitly includes machine clients. As we said before, the REST architecture offers similar controls (or affordances) to humans and machines, both of which use hypermedia. We can distinguish three kinds of machine clients. A *Web browser* is operated by a human to display hypermedia, and it can enhance the browsing experience based on a representation's content. An *API client* is a preprogrammed part of a software application, designed to interact with a specific Web API. An *autonomous agent* [6] is capable of interacting with several Web APIS in order to perform complex tasks, without explicitly being programmed to do so.

Research questions

If we bring together Fielding's definition of hypertext, the hypermedia constraint, and the publisher-driven, unidirectional linking model of the Web, an important issue arises. Any hypermedia representation must contain the links to next steps, yet how can the *publisher* of information, responsible for creating this representation, know or predict what the next steps of the *client* will be? It's not because the publisher is the client's preferred party to provide the information, that it is also the best party to provide the controls to interact with this information [21, 22]. Even if it were, the next steps differ from client to client, so a degree of personalization is involved—but based on what parameters? And is it appropriate to pass those to the publisher?

The same decisions that lead to the scalability of the Web are those that make it very hard to realize the hypermedia constraint. The fact that the publishers offer links to a client makes it easier for this client to continue the interaction, but at the same time puts a severe constraint on those publishers, who won't be able to give *every* client exactly what it needs.

Given the current properties of the Web, hypermedia can only be the engine of application state in as far as the publisher is able to provide all necessary links. While this might be the case for links that lead toward the publisher's own website, this is certainly not possible on the open Web with an ever growing number of resources. The central research question in this thesis is therefore:

How can we automatically offer human and machine clients the hypermedia controls they require to complete tasks of their choice?

An answer to this research question will eventually be explained in Chapter 6, but we need to tackle another issue first. After all, while humans generally understand what to do with hypermedia links, merely sending controls to a machine client is not sufficient. This client will need to interpret how to make a choice between different controls and what effect the activation of a certain control will have in order to decide whether this helps to reach a certain goal. The second research question captures this problem:

How can machine clients use Web APIs in a more autonomous way, with a minimum of out-of-band information?

Chapters 4 and 5 will explore a possible answer to this question, which will involve semantic technologies, introduced in Chapter 3. Finally, I want to explore the possibilities that the combination of semantics and hypermedia brings for Web applications:

How can semantic hypermedia improve the serendipitous reuse of data and applications on the Web?

This question will be the topic of Chapter 7, and is meant to inspire future research. As I will explain there, many new possibilities reside at the crossroads of hypermedia and semantics.

In this chapter, we looked at the Web from the hypermedia perspective, starting with the early hypertext systems and how the Web differs from them. Through the REST architectural style, a formalization of distributed hypermedia systems, we identified a fundamental problem of the hypermedia constraint: publishers are responsible for providing controls, without knowing the intent of the client or user who will need those controls. Furthermore, hypermedia controls alone are not sufficient for automated agents; they must be able to interpret what function the controls offer. I will address these problems by combining hypermedia and semantic technologies.

References

[1] Mike Amundsen. Hypermedia types. In: Erik Wilde and Cesare Pautasso, editors, *REST: From Research to Practice*, pages 93–116. Springer, 2011.

[2] Mike Amundsen. Hypermedia APIS with HTML5 and Node. O'Reilly, December 2011.

[3] Robin Berjon, Travis Leithead, Erika Doyle Navara, Edward O'Connor, and Silvia Pfeiffer. HTML5 – a vocabulary and associated APIS for HTML and XHTML. Candidate Recommendation. World Wide Web Consortium, December 2012. *http://www.w3.org/TR/html5/*

[4] Tim Berners-Lee. Weaving the Web: The original design and ultimate destiny of the World Wide Web by its inventor. HarperCollins Publishers, September 1999.

[5] Tim Berners-Lee, Robert Cailliau, and Jean-François Groff. The world-wide web. *Computer Networks and ISDN Systems*, 25(4–5):454–459, 1992.

[6] Tim Berners-Lee, James Hendler, and Ora Lassila. The Semantic Web. *Scientific American*, 284(5):34–43, May 2001.

[7] Tim Berners-Lee, Larry Masinter, and Mark McCahill. Uniform Resource Locators (URL). Request For Comments 1738. Internet Engineering Task Force, December 1994. *http://tools.ietf.org/html/rfc1738*

[8] Vannevar Bush. As we may think. *The Atlantic Monthly*, 176(1):101–108, July 1945.

[9] Vinton Cerf, Yogen Dalal, and Carl Sunshine. Specification of Internet transmission control program. Request For Comments 675. Internet Engineering Task Force, December 1974. *http://tools.ietf.org/html/rfc675*

[10] Jeff Conklin. Hypertext: an introduction and survey. *Computer*, 20(9): 17–41, September 1987.

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

[12] Roy Thomas Fielding. REST APIs must be hypertext-driven. Untangled – Musings of Roy T. Fielding. October 2008. *http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven*

[13] Roy Thomas Fielding, Jim Gettys, Jeffrey Mogul, Henrik Frystyk, Larry Masinter, Paul Leach, and Tim Berners-Lee. Hypertext Transfer Protocol (HTTP). Request For Comments 2616. Internet Engineering Task Force, June 1999. *http://tools.ietf.org/html/rfc2616*

[14] Roy Thomas Fielding and Richard N. Taylor. Principled design of the modern Web architecture. *Transactions on Internet Technology*, 2(2): 115–150, May 2002.

[15] Ted Nelson. Complex information processing: a file structure for the complex, the changing and the indeterminate. *Proceedings of the ACM 20th National Conference*, pages 84–100, 1965.

[16] Ted Nelson. Computer Lib / Dream Machines. self-published, 1974.

[17] Ted Nelson. Literary Machines. Mindful Press, 1980.

[18] Paul Otlet. Traité de documentation : Le livre sur le livre, théorie et pratique. Editiones Mundaneum, Brussels, Belgium, 1934.

[19] Leonard Richardson and Sam Ruby. RESTful Web Services. O'Reilly, May 2007.

[20] Simon St. Laurent, Edd Dumbill, and Joe Johnston. Programming Web Services with XML-RPC. O'Reilly, June 2001.

[21] Ruben Verborgh, Michael Hausenblas, Thomas Steiner, Erik Mannens, and Rik Van de Walle. Distributed affordance: an open-world assumption for hypermedia. *Proceedings of the 4th International Workshop on RESTful Design*. www 2013 Companion, pages 1399–1406, May 2013.

[22] Ruben Verborgh, Mathias Verhoeven, Erik Mannens, and Rik Van de Walle. Semantic technologies as enabler for distributed adaptive hyperlink generation. *Late-Breaking Results, Project Papers and Workshop Proceedings of the 21st Conference on User Modeling, Adaptation and Personalization*, June 2013.

[23] Gary Wolf. The curse of Xanadu. *Wired*, 3(6), June 1995.

Chapter 3

Semantics

Into this house we're born
Into this world we're thrown
Like a dog without a bone
An actor out on loan

— *The Doors, Riders on the Storm (1971)*

Since machines cannot fully interpret natural language (yet), they cannot make sense of textual content on the Web. Still, humans are not the only users of the Web anymore: many software agents consume online information in one way or another. This chapter details the efforts of making information machine-interpretable, the implications this has on how we should publish information, and the possibilities this brings for intelligent agents. We then discuss whether semantics are a necessity for hypermedia.

It didn't take long for machine clients to appear, as the Web's excellent scalability led to such tremendous growth that manually searching for content became impossible. *Search engines* started emerging, indexing the content of millions of webpages and making them accessible through simple keywords. Although various sophisticated algorithms drive today's search engines, they don't "understand" the content they index. Clever heuristics that try to infer meaning can give impressive results, but they are never perfect: Figure 2 shows an interesting case where Google correctly answers a query for paintings by Picasso, but fails when we ask for his books.

In 2008, Google already gave access to more than 1 trillion unique pieces of content through keyword-based search [2]. Lately, the search engine started focusing on giving *direct* answers to a query instead of presenting links to webpages that might provide those answers [21].

If we want machines to do more complex tasks than finding documents related to keywords, we could ask ourselves whether we should make the interpretation of information easier for them.

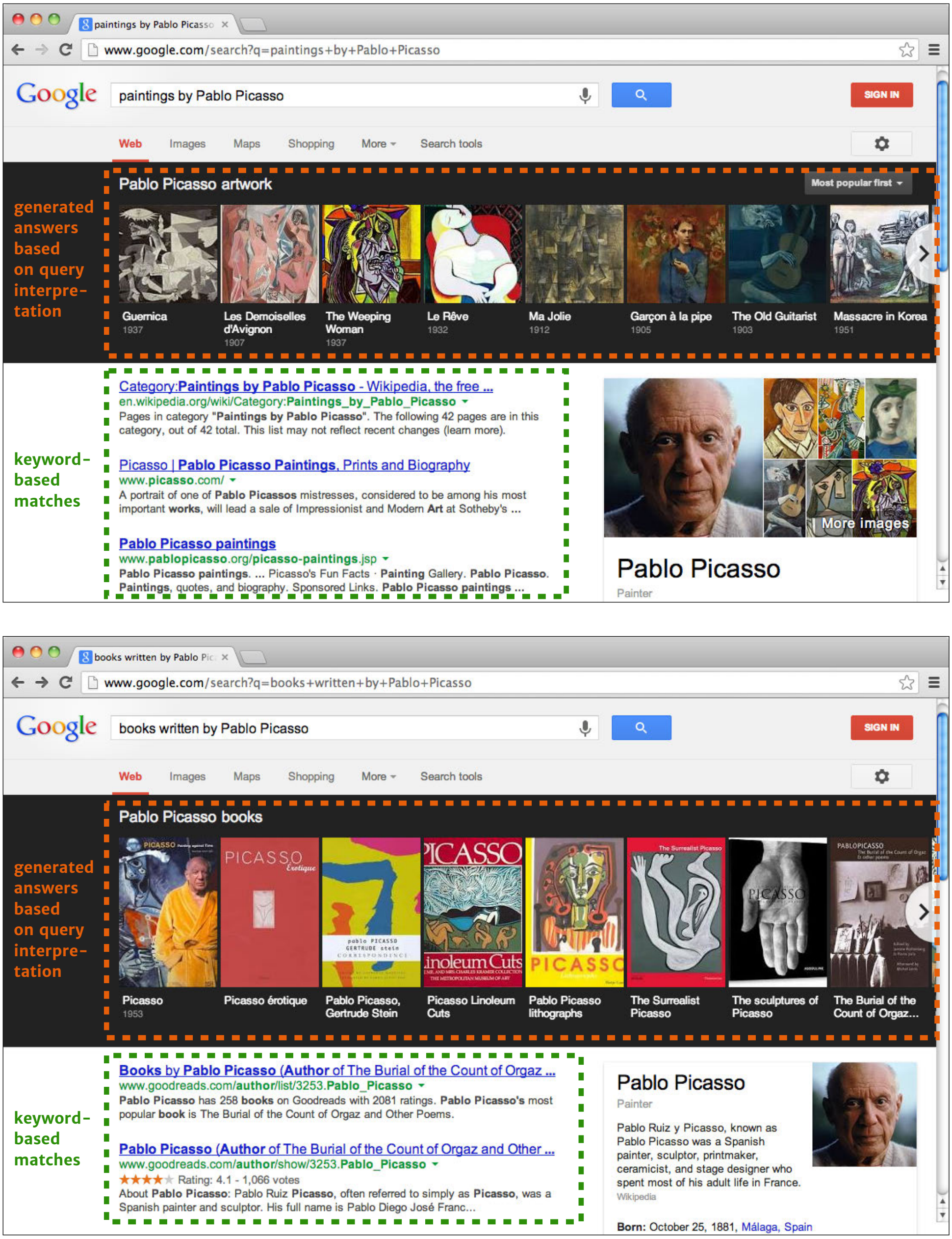


Figure 2: In addition to offering the traditional keyword-based matches, Google tries to interpret the query as a question and aims to provide the answer directly. However, machine-based interpretation remains error-prone. For instance, Google can interpret the query "*paintings by Pablo Picasso*" correctly, as it is able to show a list of paintings indeed. The query "*books written by Pablo Picasso*" seemingly triggers a related heuristic, but the results consist of books *about*—not *written by*—the painter; an important semantic difference. *©Google*

The Semantic Web

The idea of adding *semantics* to Web resources was popularized by the now famous 2001 *Scientific American* article by Tim Berners-Lee, Jim Hendler, and Ora Lassila, wherein they laid out a vision for what they named the *Semantic Web*. Perhaps the most important starting point is this fragment [12]:

> *The Semantic Web is not a separate Web but an extension of the current one, in which information is given well-defined meaning, better enabling computers and people to work in cooperation.* — *Tim Berners-Lee et al.*

The original article starts with a futuristic vision of intelligent agents that act as personal assistants. At the 2012 International Semantic Web Conference, Jim Hendler revealed this angle was suggested by the editors, and then jokingly tested how much of this vision was already being fulfilled by Apple's Siri (which promptly failed to recognize his own name).

The Web already harbors the infrastructure for machines, as explained in the previous chapter when discussing the REST architectural style. However, there's only so much a machine can do with *structural* markup tags such as those found in HTML documents: the data can be parsed and transformed, but all those tasks require precise instruction if there is no deeper understanding of that data. Compare this to processing a set of business documents in a language you don't understand. If someone tells you how to classify them based on structural characteristics, such as the presence of certain words or marks, you might be able to do that. However, this strategy fails for documents that are structured differently, even if they contain the same information.

Knowledge representation

A first task of the Semantic Web is thus knowledge representation: providing a model and syntax to exchange information in a machine-interpretable way. The Resource Description Framework (RDF) [28] is a model that represents knowledge as **triples** consisting of a *subject*, *predicate*, and *object*. Different syntaxes exist; the Turtle syntax [4] expresses triples as simple patterns that are easily readable for humans and machines. Starting from a basic example, the fact that Tim knows Ted can be expressed as follows in Turtle.

```
:Tim :knows :Ted.
```

This is a single triple consisting of the three parts separated by whitespace, `:Tim` (subject), `:knows` (predicate), and `:Ted` (object), with a final period at the end. While a machine equipped with a Turtle parser is able to slice up the above fragment, there is not much semantics to it. To a machine, the three identifiers are opaque and thus a meaningless string of characters like any other.

The XML serialization of RDF used to be the standard, but its hierarchical structure is often considered more complex than Turtle's triple patterns.

Just like on the "regular" Web, the trick is identification: if we use URLS for each part, then each concept is uniquely identified and thus receives a well-defined interpretation.

```
<http://dbpedia.org/resource/Tim_Berners-Lee>↩
  <http://xmlns.com/foaf/0.1/knows>↩
  <http://rdf.freebase.com/ns/en.ted_nelson>.
```

In the above fragment, the identifiers have been replaced by URLS which correspond to, respectively, Tim Berners-Lee, the concept "knowing", and Ted Nelson. This is how **meaning** is constructed: a concept is uniquely identified by one or more URLS, and a machine can interpret statements about the concept by matching its URL. If a machine is aware that the above URL identifies Tim Berners-Lee, then it can determine the triple is a statement about this person. If it is also aware of the "knows" predicate, it can determine that the triple means "Tim Berners-Lee knows somebody". And of course, comprehension of Ted Nelson's URL implies the machine can *"understand"* the triple: Tim has a "knows" relation to Ted—or "Tim knows Ted" in human language. Of course, the notion of *understanding* should be regarded as *interpretation* here. It conveys the fact a machine can now apply the properties of the "knows" relationship to infer other facts; it does not trigger the cognitive, intellectual, or emotional response the same information does when perceived by a human. This is not unlike the Chinese room thought experiment [31]—the ability to manipulate symbols doesn't necessarily imply understanding.

As we've seen in the last chapter, a URL identifies a *conceptual* resource, so it is perfectly possible to point to a person or a real-world relation. But how can we *represent* a person digitally? We can't—but we can represent a document *about* this person. If you open any of the three URLS in a browser, you will see they indeed redirect to a document using HTTP status code `303 See Other` [23]. The differentiation between non-representable and representable resources has been the subject of a long-standing discussion in the W3C Technical Architecture Group [7].

Since URLS appear a lot in RDF fragments, Turtle provides an abbreviated syntax for them:

```
@prefix dbp: <http://dbpedia.org/resource/>.
@prefix foaf: <http://xmlns.com/foaf/0.1/>.
@prefix fb: <http://rdf.freebase.com/ns/>.

dbp:Tim_Berners-Lee foaf:knows fb:ted_nelson.
```

Note how recurring parts of URLS are declared at the top with `prefix` directives, which saves space and improves clarity when there are many triples in a document.

Now what if a machine doesn't have any knowledge about one or more of the URLS it encounters? This is where the power of the "classic" Web comes in again. By **dereferencing** the URL—using HTTP to retrieve a representation of the resource—the machine can discover the meaning of the concept in terms of its relation to other concepts it *does* recognize. Once again, the knowledge resides in the links [19].

Time will tell if a comparison to the human brain, where information is encoded as *connections* between neurons, could be appropriate.

Ontologies

Related knowledge is often grouped together in **ontologies**, which express the relationship between concepts. For instance, the "knows" predicate on the previous page comes from the Friend of a Friend (FOAF) ontology, which offers a vocabulary to describe people and their relationships. If we dereference the URL of this predicate, we will be redirected to an RDF document that expresses the ontology using RDF Schema [18] (RDFS—a set of basic ontological properties) and Web Ontology Language [29] (OWL—a set of more complex constructs). The relevant part of the ontology looks similar to this:

Just like with regular Web documents, concepts can have many URLS, as long as one URL identifies only a single concept. Multiple ontologies can thus define the same concept (but they'll likely do it in a slightly different way).

```
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix owl: <http://www.w3.org/2002/07/owl#>.

foaf:knows a owl:ObjectProperty;
           rdfs:domain foaf:Person;
           rdfs:label "knows";
           rdfs:range foaf:Person.
```

For brevity, prefixes used before won't be repeated; Turtle parsers still need them, though.

This expresses that "knows" is a property that can occur from a person resource to another person resource. Also note the use of semicolons for continued statements about a same subject, and the predicate "a", which is short for `rdf:type`. This ontology can help machines to build an understanding of concepts—under the fair assumption that they have built-in knowledge about RDFS and OWL. For instance, if a software agent wouldn't recognize any of the URLS in the earlier "Tim knows Ted" example, it could look up the "knows" predicate and derive that both Tim and Ted must be a `foaf:Person`.

The `rdf` namespace is *http://www.w3.org/1999/02/22-rdf-syntax-ns#*.

The more ontological knowledge is available, the more deductions can be made. For instance, the human-readable documentation of FOAF says that the "knows" property indicates some level of reciprocity. With OWL, we can capture this as:

```
foaf:knows a owl:SymmetricProperty.
```

This would allow a machine to conclude that, if "Tim knows Ted", the triple "Ted knows Tim" must also be a fact—even if it is not explicitly mentioned in the initial Turtle fragment. It can even deduce that without having to understand anything about the entities "Ted", "knows", or "Tim", because the knowledge that "knows" is a symmetric predicate is sufficient to deduce the reverse triple.

Dan Brickley, the author of FOAF, noticed later that `foaf:knows`, despite its definition, became widely used for uni-directional "knows" relations; for instance, the Twitter followers of a certain person. This indicates that meaning can evolve through usage, not unlike *semantic drift* in natural languages.

Reasoning

To make such deductions, we need Semantic Web **reasoners** that are able to make semantically valid inferences. Various types of reasoners exist: some possess implicit, built-in knowledge about RDF and OWL; others are designed for explicit knowledge addition. An example of the former category is Pellet [32]; examples of the latter category are cwm [5] and EYE [20]. In the context of my work, reasoners with explicit knowledge are more helpful, as they allow a higher degree of customization. In particular, cwm and EYE are rule-based reasoners for the Notation3 (N3) language [9], which is a superset of Turtle that includes support for *formulas*, *variables*, and *quantification*, allowing the creation of **rules**. For instance, the following rule indicates that if person A knows person B, then person B also knows person A:

Formulas enable the use of a set of triples (between braces) as the subject or object of another triple.

A rule is actually a regular triple "`:x => :y.`", where the arrow => is shorthand for `log:implies`, and the `log` prefix expands to *http://www.w3.org/2000/10/swap/log#*.

```
{
  ?a foaf:knows ?b.
}
=>
{
  ?b foaf:knows ?a.
}.
```

If we supply the above rule to an N3 reasoner together with the triple "`:Tim foaf:knows :Ted`", then this reasoner will use N3Logic semantics [10] to deduce the triple "`:Ted foaf:knows :Tim`" from that.

As in any branch of software engineering, maximizing reuse is important for efficient development. Therefore, it is more interesting to encode the symmetry of `foaf:knows` on a higher level of abstraction. We can encode this meaning directly on the ontological level:

Rules for common RDFS and OWL predicates can be loaded from the EYE website [20]. They provide explicit reasoning on triples that use those constructs.

```
{
  ?p a owl:SymmetricProperty.
  ?a ?p ?b.
}
=>
{ ?b ?p ?a. }.
```

Indeed, for any symmetric property P that is true for A with respect to B holds that it's also true for B with respect to A. Therefore, the statement that `foaf:knows` is symmetric, together with the above rule for symmetric properties, will allow to make the same conclusion about Tim and Ted. However, this rule can be reused on other symmetric properties and is thus preferred above the first one.

An important difference with offline reasoning is that Semantic Web reasoning makes the **open-world assumption**. Since different sources of knowledge are spread across the Web, the fact that a triple does not appear in a certain document does *not* entail the conclusion that this triple is false or does not exist. Similar to how the Web treats hypermedia, the Semantic Web gives up *completeness* in favor of *decentralization* and *openness*. This gives an interesting flexibility to knowledge representation, but also has limitations on what we can do easily. For instance, *negations* are particularly hard to express. Another consequence is that resources with different URLs are not necessarily different—this has to be explicitly indicated or deduced.

The far-reaching consequence of an open world is that no single resource can contain the full truth: *"anyone can say anything about anything"* [8, 28].

Agents

One of the concepts that seems inseparably connected to the Semantic Web is the notion of intelligent **software agents** that perform complex tasks based on the knowledge they extract from the Web. The original idea was that you could instruct your personal agent somehow to perform tasks for you online [12]. Typical examples would be scenarios that normally require a set of manual steps to be completed. For instance, booking a holiday, which requires interacting with your agenda and arranging flights, hotels, and ground transport, among other things. It's not hard to imagine the many steps this takes, and every one of them involves interaction with a different provider. If a piece of software can understand the task "booking a holiday" and if it can interact with all of the involved providers, it should be able to perform the entire task for us.

While the initial optimism was high by the end of the 1990s—and certainly in the initial Semantic Web article [12]—the expectations have not yet been met. Jim Hendler, co-author of that famous article, rightly wondered where the intelligent agents are [24], given that all necessary pieces of technology have been already developed. However, this is also a question of *usage*, leading to the Semantic Web's classical *chicken-and-egg* problem: there aren't enough semantic data and services because there are no agents, and there are no agents because there aren't enough data and services. The possible benefits semantic technologies might bring currently don't provide the necessary incentive for publishers to "semanticize" their data and services [34]. Furthermore, one could doubt whether the technology is sufficiently advanced to provide the degree of intelligence we desire. Nonetheless, the current Semantic Web infrastructure provides the foundations for agents to independently consume information on the open Web.

We might wonder to what extent Apple's digital assistant Siri already fulfills the Semantic Web vision of intelligent agents [3]. Even though responding to voice commands with various online services is impressive for today's standards, Siri operates on a *closed world*: it can only offer those services it has been preprogrammed for. Semantic Web agents would need to operate on an open world. *©Apple*

Linked Data

On more than one occasion, Tim Berners-Lee has called Linked Data *"the Semantic Web done right"*.

In the early years of the Semantic Web, the focus on the agent vision was very strong and this attracted several people from the artificial intelligence community [24]. However, this also made the Semantic Web a niche topic, difficult to understand without a strong background in logics. And at the same time, the chicken-and-egg deadlock situation still remained—no agents without data and vice-versa. Tim Berners-Lee realized this, and proposed four rules to make data available in the spirit of the (Semantic) Web. They became known as the **Linked Data principles** [6]:

Confusingly, Berners-Lee also coined the *five stars of Linked (Open) Data* that correspond roughly to the four principles [6].

1. Use URIS as names for things.
2. Use HTTP URIS so that people can look up those names.
3. When someone looks up a URI, provide useful information, using the standards (RDF, SPARQL).
4. Include links to other URIS so that they can discover more things.

A common example of URIS that are not URLS are ISBN URIS. For instance, `urn:isbn:9780061122590` identifies a book, but does not locate it.

The first principle is a matter of unique identification. Up until now, we have only talked about Uniform Resource *Locators* (URLS), but Uniform Resource *Identifiers* (URIS) [11] are a superset thereof, providing identification but not necessarily location. The second principle specifically asks for HTTP URIS (thus URLS). This might seem evident, but actually, many datasets and ontologies used non-HTTP URIS in the beginning days of the Semantic Web. If we want software agents to discover meaning automatically by dereferencing, URLS as identifiers are a prerequisite. Third, dereferencing these URLS should result in representations that are machine-interpretable. And fourth, such representations should contain links to other resources, so humans and machines can build a context.

More triples do not necessarily bring more knowledge though, as humorously proven by *Linked Open Numbers*, a dataset with useless facts about natural numbers [35].

Since their conception in 2007, these principles have inspired many new datasets [15] and continue to be an inspiration. We are now at a stage where a considerable amount of data with an open license is available for automated consumption. Large data sources are DBpedia [16], which contains data extracted automatically from Wikipedia, and Freebase [17], a crowd-sourced knowledge base.

Linked Data is decentralized knowledge representation on a Web scale. True to the Semantic Web principles, the meaning of the data resides in its links. If a machine doesn't recognize a URL, it can dereference this URL to find an explanation of the resource in terms of the resources that it links to. By design, no knowledge source will ever be complete, but the open-world assumption allows for this. After all, no Web page contains *all* information about a single topic.

The hypermedia connection

The REST principles

How does hypermedia fit into the semantics story? After all, the Semantic Web happens on the Web, the basis of which is hypermedia. If we take a closer look at the Linked Data principles, we notice that they align well with the constraints of REST's uniform interface. To make this more obvious, let's try to reformulate these constraints as four rules that correspond to those of Linked Data:

The semantic and REST communities tend to be quite disparate, yet their basic principles are very similar.

1. Any concept that might be the target of a hypertext reference must have a resource identifier.
2. Use a generic interface (like HTTP) for access and manipulation.
3. Resources are accessed through various representations, consisting of data and metadata.
4. Any hypermedia representation must contain controls that lead to next steps.

The parallels are striking, but not surprising—what is important for the Web must be important for the Semantic Web. In particular, the same links that are the essence of Linked Data are crucial to satisfying the *hypermedia constraint*. In that sense, this constraint is the operational version of the fourth Linked Data principle: Linked Data requires links in order to interpret a concept without prior knowledge; REST requires links in order to navigate an application without prior knowledge.

In REST systems, hypermedia should be the engine of application state. Similarly, on the Semantic Web, hypermedia should be the engine of knowledge discovery.

A little semantics

In REST architectures, *media types* are used to capture the structure and semantics of a specific kind of resource [22, 36]. After all, the uniform interface is so generic that application-specific semantics must be described inside the representation. Yet, there's a trade-off between *specificity* and *reusability* [30]. Media types that precisely capture an application's semantics are likely too specific for any other application, and media types that are generic enough to serve in different applications are likely not specific enough to automatically interpret the full implications of an action. Therefore, more media types do not necessarily bring us closer to an independent evolution of clients and servers.

HTML is a generic media type, as it can accommodate any piece of content, albeit with only limited machine-interpretability. The vCard format is highly specific, as it can contain only contact information, but machines interpret it without difficulty.

If *semantic annotations* are added to a generic media type, they can provide a more specific meaning to a resource, enabling complex interactions on its content. And, as we'll see in the next chapter, semantics can help a software agent understand what actions are possible on that resource, and what happens if an action is executed.

If a machine can extract an address from semantic annotations in an HTML page, it gets the same options as with vCard.

The phrase *"a little semantics goes a long way"* must be one of the most widely known within the community. (Some people like to add *"…but no semantics gets you even further."*)

However, the explanation of the REST principles in the last chapter can make us wonder why we would *enhance* media types with semantics. Content negotiation can indeed make the same resource available in separate human- and machine-targeted representations. In practice, content-negotiation is not widespread. Part of this is because people are unfamiliar with the principle, as we almost exclusively deal with single-representation files when using a local computer. Additionally, many Web developers are only vaguely familiar with representation formats other than HTML. Finally, for many applications, human- and machine-readable aspects are needed at the same time. For instance, search engines process HTML content aided by annotations, and a browser can read annotations to enhance the display of a webpage. Several annotation mechanisms for HTML exist:

The 2012 version of the *Common Crawl Corpus* shows that Microformats are currently most popular on the Web, followed at a considerable distance by RDFa and finally HTML5 Microdata [14]. Perhaps in the future, the Microformats advantage will decrease, as new formats no longer emerge. The question then becomes whether RDFa and Microdata will survive, and which of them will take the lead.

Microformats [27] are a collection of conventions to structure information based on specific HTML elements and attributes. Examples are hCard to mark up address data and hCalendar for events. The drawback of Microformats is that they are collected centrally and only specific domains are covered. Furthermore, the syntax of each Microformat is slightly different.

RDFa or Resource Description Framework in Attributes [1] is a format to embed RDF in HTML representations. Its benefit is that any vocabulary can be used, and with RDFa Lite [33], a less complex syntax is possible. Usage of Facebook's OpenGraph vocabulary [26] is most common [14], thanks to the incentive for adopters to have better interactions on the Facebook social networking site.

Microdata is a built-in annotation format in HTML5 [25]. An incentive to adopt this format is Schema.org [13], a vocabulary created and endorsed by Google and other search engines. The expectation is that they will index publishers' content more accurately and enhance its display if relevant markup is present [34].

While an increasing amount of semantic data on the Web is welcomed, the current diversity makes it in a sense more difficult for publishers to provide the right annotations. After all, the benefit of semantic technologies should be that you are free to use *any* annotation, since a machine is able to infer its meaning. However, the current annotation landscape forces publishers to provide annotations in different formats if they want different consumers to interpret them. On the positive side, the fact that there are several incentives to publish semantic annotations gives agents many opportunities to perform intelligent actions based on the interpretation of a resource.

The Semantic Web provides tools that help machines make sense of content on the Web. The Linked Data initiative aims to get as many datasets as possible online in a machine-interpretable way. Semantic technologies can help agents consume hypermedia without the need for a specific document type, improving the autonomy of such agents. There are several incentives for publishers to embed semantic markup in hypermedia documents, which aids automated interpretation. However, fragmentation issues still remain.

References

[1] Ben Adida, Mark Birbeck, Shane McCarron, and Ivan Herman. RDFa core 1.1. Recommendation. World Wide Web Consortium, 7 June 2012. *http://www.w3.org/TR/2012/REC-rdfa-core-20120607/*

[2] Jesse Alpert and Nissan Hajaj. We knew the Web was big… Google Official Blog, July 2008. *http://googleblog.blogspot.com/2008/07/we-knew-web-was-big.html*

[3] Jacob Aron. How innovative is Apple's new voice assistant, Siri? *New Scientist*, 212(2836):24, 3 November 2011.

[4] David Beckett, Tim Berners-Lee, Eric Prud'hommeaux, and Gavin Carothers. Turtle – Terse RDF Triple Language. Candidate Recommendation. World Wide Web Consortium, 19 February 2013. *http://www.w3.org/TR/turtle/*

[5] Tim Berners-Lee. cwm, 2000–2009. *http://www.w3.org/2000/10/swap/doc/cwm.html*

[6] Tim Berners-Lee. Linked Data, July 2006. *http://www.w3.org/DesignIssues/LinkedData.html*

[7] Tim Berners-Lee. What is the range of the HTTP dereference function? Issue 14. W3C Technical Architecture Group, 25 March 2002. *http://www.w3.org/2001/tag/group/track/issues/14*

[8] Tim Berners-Lee. What the Semantic Web can represent, December 1998. *http://www.w3.org/DesignIssues/RDFnot.html*

[9] Tim Berners-Lee and Dan Connolly. Notation3 (N3): a readable RDF syntax. Team Submission. World Wide Web Consortium, 28 March 2011. *http://www.w3.org/TeamSubmission/n3/*

[10] Tim Berners-Lee, Dan Connolly, Lalana Kagal, Yosi Scharf, and Jim Hendler. N3Logic: A logical framework for the World Wide Web. *Theory and Practice of Logic Programming*, 8(3):249–269, May 2008.

[11] Tim Berners-Lee, Roy Thomas Fielding, and Larry Masinter. Uniform Resource Identifier (URI): generic syntax. Request For Comments 3986. Internet Engineering Task Force, January 2005. *http://tools.ietf.org/html/rfc3986*

[12] Tim Berners-Lee, James Hendler, and Ora Lassila. The Semantic Web. *Scientific American*, 284(5):34–43, May 2001.

[13] Bing, Google, Yahoo!, and Yandex. Schema.org, *http://schema.org/*

[14] Christian Bizer, Kai Eckert, Robert Meusel, Hannes Mühleisen, Michael Schuhmacher, and Johanna Völker. Deployment of RDFa, Microdata, and Microformats on the Web – a quantitative analysis. *Proceedings of the 12th International Semantic Web Conference*, October 2013.

[15] Christian Bizer, Tom Heath, and Tim Berners-Lee. Linked Data – the story so far. *International Journal on Semantic Web and Information Systems*, 5(3):1–22, March 2009.

[16] Christian Bizer, Jens Lehmann, Georgi Kobilarov, Sören Auer, Christian Becker, Richard Cyganiak, and Sebastian Hellmann. DBpedia – a crystallization point for the Web of Data. *Web Semantics: Science, Services and Agents on the World Wide Web*, 7(3):154–165, 2009.

[17] Kurt Bollacker, Colin Evans, Praveen Paritosh, Tim Sturge, and Jamie Taylor. Freebase: a collaboratively created graph database for structuring human knowledge. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1247–1250, 2008.

[18] Dan Brickley and Ramanathan V. Guha. RDF vocabulary description language 1.0: RDF Schema. Recommendation. World Wide Web Consortium, 10 February 2004. *http://www.w3.org/TR/rdf-schema/*

[19] Vannevar Bush. As we may think. *The Atlantic Monthly*, 176(1):101–108, July 1945.

[20] Jos De Roo. Euler Yet another proof Engine, 1999–2013. *http://eulersharp.sourceforge.net/*

[21] Amir Efrati. Google gives search a refresh. *The Wall Street Journal*, 15 March 2012.

[22] Roy Thomas Fielding. REST APIs must be hypertext-driven. Untangled – Musings of Roy T. Fielding. October 2008. *http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven*

[23] Roy Thomas Fielding, Jim Gettys, Jeffrey Mogul, Henrik Frystyk, Larry Masinter, Paul Leach, and Tim Berners-Lee. Hypertext Transfer Protocol (HTTP). Request For Comments 2616. Internet Engineering Task Force, June 1999. *http://tools.ietf.org/html/rfc2616*

[24] James Hendler. Where are all the intelligent agents? *IEEE Intelligent Systems*, 22(3):2–3, May 2007.

[25] Ian Hickson. HTML microdata. Working Draft. World Wide Web Consortium, 25 October 2012. *http://www.w3.org/TR/2012/WD-microdata-20121025/*

[26] Facebook Inc. The Open Graph protocol, *http://ogp.me/*

[27] Rohit Khare and Tantek Çelik. Microformats: a pragmatic path to the Semantic Web. *Proceedings of the 15th International Conference on World Wide Web*, pages 865–866, 2006.

[28] Graham Klyne and Jeremy J. Carrol. Resource Description Framework (RDF): Concepts and Abstract Syntax. Recommendation. World Wide Web Consortium, 10 February 2004. *http://www.w3.org/TR/rdf-concepts/*

[29] Deborah Louise McGuinness and Frank van Harmelen. OWL Web Ontology Language – overview. Recommendation. World Wide Web Consortium, 10 February 2004. *http://www.w3.org/TR/owl-features/*

[30] Leonard Richardson, Mike Amundsen, and Sam Ruby. RESTful Web APIs. O'Reilly, September 2013.

[31] John Rogers Searle. Minds, brains, and programs. *Behavioral and Brain Sciences*, 3(3):417–427, September 1980.

[32] Evren Sirin, Bijan Parsia, Bernardo Cuenca Grau, Aditya Kalyanpur, and Yarden Katz. Pellet: a practical OWL-DL reasoner. *Web Semantics: Science, Services and Agents on the World Wide Web*, 5(2):51–53, 2007.

[33] Manu Sporny. RDFa Lite 1.1. Recommendation. World Wide Web Consortium, 7 June 2012. *http://www.w3.org/TR/rdfa-lite/*

[34] Ruben Verborgh, Erik Mannens, and Rik Van de Walle. The rise of the Web for Agents. *Proceedings of the First International Conference on Building and Exploring Web Based Environments*, pages 69–74, 2013.

[35] Denny Vrandečić, Markus Krötzsch, Sebastian Rudolph, and Uta Lösch. Leveraging non-lexical knowledge for the Linked Open Data Web. *5^{th} Review of April Fool's day Transactions*, 2010.

[36] Jim Webber, Savas Parastatidis, and Ian Robinson. REST in Practice. O'Reilly, September 2010.

Chapter 4

Functionality

How do you do the things that you do?
No one I know could ever keep up with you
How do you do?
Did it ever make sense to you?

— Roxette, How Do You Do! (1992)

For people, navigating hypermedia feels entirely natural. We read texts, view images and video, and *through* them, we can not only reach the next piece of information we need; we can also perform actions that modify things in a predictable way. Machines face far greater difficulties. Even if they can interpret the information on a page, it's currently difficult for them to understand the impact of change. Therefore, we've developed RESTdesc, a method to describe the functionality of hypermedia controls in applications. Unlike other approaches, it focuses on enabling autonomous agents to use Web applications in a hypermedia-driven way.

The uniform interface constraints of the REST architectural style mandate that messages be self-descriptive [5]. This is why HTTP adopts *standard methods* that act on resources, as opposed to many other remote protocols that allow the definition of any method. The uniform interface brings simplicity through universal understanding: a software agent knows that `GET` retrieves a document and that `DELETE` removes it. However, there is only so much that can be expressed in a uniform interface. Every application offers specific functionality that cannot be accounted for unambiguously in a specification that applies to *any* Web application.

Unfortunately, on today's Web, many APIs circumvent the uniform interface by adding methods to the URL or the message body. These APIs then lose desirable properties of REST, moving the interpretation from the message to out-of-band documentation.

Sometimes, POST is even used in those cases where another standard HTTP method is perfectly applicable, stretching its semantics beyond what was accounted for. Part of the explanation for this non-standard usage is that HTML only supports GET and POST on forms, even though it offers the other methods through JavaScript.

The POST method essentially encompasses everything no other standard method provides. In the HTTP specification, it is defined to cover annotation of resources, message posting, data processing, and database appending, but the actual function performed by the POST method is *"determined by the server and is usually dependent on the request URI"* [6]. This means that, for any POST request, the message is self-descriptive in the sense that it asks for an action that follows the definition, but that definition is so uniform that we don't know what exactly is going on. On the protocol level, we understand the message. On the application level, we don't—unless we look for clues in the message body, but they are usually only interpretable by humans. And once the message has been constructed, it might already be too late: an operation with possible side effects could have been requested. Summarizing, in addition to the semantics of the *information* itself, agents require the semantics of the *actions* this information affords.

Describing functionality

Design goals

RDF could capture functionality indirectly, but this would necessitate an interpretation that is not native to RDF processors.

RDF, as its expansion to "Resource Description Framework" indicates, has been created for the description of *resources*, the unit of information in the REST architectural style. RDF captures the state of a resource at a given moment in time, but it cannot directly capture state *changes*. We need a method for *dynamic* information that conforms to the following characteristics.

The last four goals refer to REST's uniform interface constraints [5]: resource-orientation, manipulation through representations, self-describing messages, and hypermedia as the engine of application state.

- The goal of descriptions is to capture **functionality**: expressing the relation between *preconditions* and *postconditions*.
- Consumers should require **no additional interpretation** beyond knowledge of HTTP.
- Descriptions should be **resource-oriented**: they should describe on the level of *application-specific* resources, not in terms of generic concepts such as "services" or "parameters".
- Resources should be described in a **representation-independent** way—the media type is determined at runtime.
- Each description can be **interpreted independently** of others.
- The runtime interaction should remain **driven by hypermedia**: descriptions *support* the interaction but do not *determine* it.

With these goals in mind, we can derive the foundations of a method to describe the functionality of those Web APIs that conform to all REST constraints, indicated by the term *hypermedia APIs* [1].

Description anatomy

In essence, there are three things we need to capture: *preconditions*, *postconditions*, and the *HTTP request* that allows the state transition from the former to the latter. The other properties depend on the design choices we make. Those three components are related to each other as follows. Given a set of preconditions $pre_{\mathbf{A}}$ on a resource x, a description of an action **A** should express what request is necessary to obtain a set of postconditions $post_{\mathbf{A}}$. We could represent this relationship schematically as follows:

$$\mathbf{A}(x) \equiv pre_{\mathbf{A}}(x) \xrightarrow{req_{\mathbf{A}}(x)} post_{\mathbf{A}}(x)$$

The fact that the implication is fulfilled by executing the request $req_{\mathbf{A}}$ is symbolized by indicating it on top of the implication arrow, but we still need to formalize this meaning. A naive conversion to first-order logic treats the request as a part of the precondition:

The preconditions only imply the postconditions *through* the execution of the request.

$$\mathbf{A}(x) \equiv pre_{\mathbf{A}}(x) \wedge req_{\mathbf{A}}(x) \Longrightarrow post_{\mathbf{A}}(x)$$

The above equation expresses that, if the preconditions are fulfilled, an execution of the request will always lead to the postconditions. However, this cannot be guaranteed in practice. On large distributed systems such as the Web, requests can fail for a variety of reasons. Therefore, we can only state that a hypothetical request r exists that makes the postconditions true:

Having preconditions in the antecedent does not account for errors that are likely to happen in distributed systems and is thus insufficient.

$$\mathbf{A}(x) \equiv pre_{\mathbf{A}}(x) \Longrightarrow \exists r \left(req_{\mathbf{A}}(x,r) \wedge post_{\mathbf{A}}(x,r)\right)$$

So given preconditions $pre_{\mathbf{A}}(x)$, there always exists a request r for the action **A** on the resource x for which postconditions $post_{\mathbf{A}}(x,r)$ hold. We cannot be sure whether all requests that look like r will succeed, since several components can fail, but we can *attempt* to construct r. Consequently, this is the model we will use for descriptions.

Preconditions in the consequent align best with reality: some successful request exists, but that doesn't guarantee success for each similar request.

This design choice is also necessary to avoid introducing a logical contradiction in the process. Indeed, if we had modeled the request in the antecedent of the rule, and its execution would fail for any reason, then we couldn't combine the prior knowledge about the action *("the preconditions and request always lead to the postconditions")* and the posterior knowledge *("the request has been executed but the postconditions do not hold")* in a monotonic logic environment. Additionally, the existential formulation allows "triggering" the rule before the request has been issued—exactly what we need in order to use descriptions for planning. In the next section, we will translate this abstract syntax into a concrete description format.

While monotonicity is not strictly required, it makes reasoning simpler and is a prerequisite to generate proofs, wherein the use of retractable facts would be problematic.

Expressing descriptions

The presence of variables and quantification suggest that regular RDF won't possess the expressivity needed to convey these descriptions. As indicated in the previous chapter, Notation3, an RDF superset, does provide this support. Additionally, we need a vocabulary to detail HTTP requests: the existing "HTTP vocabulary in RDF" [9] provides all necessary constructs. The combination of N3 and this vocabulary form the functional Web API description method I named RESTdesc [19, 20]. The skeleton of a RESTdesc description looks like this:

The specified preconditions on a resource imply the existence of a certain request that effectuates postconditions on this resource. *Variables* like `?resource` are universally quantified; *blank nodes* such as `_:request` are existentially quantified [2].

```
{
  ?resource  ...  ... .
}
=>
{
  _:request http:methodName [...];
            http:requestURI [...];
            http:resp [...].
  ?resource  ...  ... .
}.
```

For instance, the following description explains that you can receive an 80px-high thumbnail of an image by performing a `GET` request on the link labeled `ex:smallThumbnail`:

The `ex:` prefix is local to the application; agents aren't assumed to already understand it.

Actually, this RESTdesc description explains what the `ex:smallThumbnail` relation means in terms of non-local predicates, for instance those from DBpedia. In contrast to "traditional" ontologies, the meaning expressed here is *operational*: it details what happens when a certain request is executed upon the object.

```
@prefix ex: <http://example.org/image#>.
@prefix http: <http://www.w3.org/2011/http#>.
@prefix dbpedia: <http://dbpedia.org/resource/>.
@prefix dbpedia-owl: <http://dbpedia.org/ontology/>.

{
  ?image ex:smallThumbnail ?thumbnail.
}
=>
{
  _:request http:methodName "GET";
            http:requestURI ?thumbnail;
            http:resp [ http:body ?thumbnail ].

  ?image dbpedia-owl:thumbnail ?thumbnail.
  ?thumbnail a dbpedia:Image;
             dbpedia-owl:height 80.0.
}.
```

Let's examine this example description, considering the design goals.

- The description captures functionality, in the sense that it expresses the `ex:smallThumbnail` relation not in a static, ontological way, but from the viewpoint of an HTTP request it affords, and the properties a result of this request will have. This allows an agent to decide whether to issue the request, based on the desirability of its effects.
- To generate the request, agents only need to understand the `http` ontology, which is universal to all RESTdesc descriptions. This particular description also uses an application-specific `ex` ontology and DBpedia resources. An agent doesn't need knowledge about the `ex` ontology, as the description explains the used predicate. The usage of DBpedia concepts seems to cause more difficulties for agents, however, any agent consuming this API will have to deal with images in *some* way, even if only implicitly through goals set by the user. Since DBpedia is published as Linked Data, we assume the agent is able to map `dbpedia:Image` to its own understanding of *"image"* if required.
- The description is resource-oriented: it focuses on the actual application domain (images and thumbnails) instead of a meta-level (such as services or parameters).
- The representation is not fixed and can be determined at runtime: the description only explains that the image will be a thumbnail of the original image and that it will have a height of 80 pixels.
- No other descriptions are needed for interpretation. For instance, it does not matter how the original image is created.
- The description supports hypermedia-driven interactions, as it doesn't contain fixed URLs or templates. Rather, it expresses the fact that *if* we follow an image's `smallThumbnail` link through hypermedia, *then* we can perform a `GET` request on this link's target, which will result in a resource with these properties.

Basic knowledge of HTTP is the only constraint we put on agents, and this is reasonable since they need HTTP in any case. All other knowledge is domain-specific; we should strive to use Linked Data so the agent can look up any unknown terms autonomously.

RESTdesc is *not* limited to APIs that communicate in RDF, but those APIs that do have several benefits, as we'll see in Chapter 7.

One can question the utility of describing a `GET` request, since the HTTP specification specifies this method already in detail [6]. The answer is twofold. On the one hand, *dereferencing* is assumed within Linked Data, so we can indeed omit the implied `GET` request in descriptions. On the other hand, the description above conveys an *expectation*: it tells that the representation will be a thumbnail of the original image. This can save us the `GET` operation if we don't need that. More importantly, it can guide an agent when *planning* a sequence of steps: even if an image has not been created yet, the agent knows it will be able to get its thumbnail.

`GET` requests are *safe* and *idempotent*, so they may not change resource state. In contrast to the *unsafe* `POST` requests, they are defined strictly and narrowly. That doesn't mean there's no use for descriptions—agents need to know what things they can retrieve.

Functional descriptions are certainly necessary for methods that are intentionally underspecified, such as HTTP POST. Depending on the resource URI, a wide variety of actions might happen. RESTdesc narrows this down to what is described. For instance, the following N3 rule states that an image posted to an album receives comments and thumbnail links.

No restrictions are placed on ?image; it can be an image on the local file system, or any image on the Web. These details are agreed on at runtime. During planning, it suffices to know that POSTing the image will lead to the described effects.

```
{
  ?profile ex:hasAlbum ?album.
  ?image a dbpedia:Image.
}
=>
{
  _:request http:methodName "POST";
            http:requestURI ?album;
            http:body ?image;
            http:resp [ http:body ?image ].
  ?image ex:comments _:comments;
         ex:smallThumbnail _:thumb;
         ex:mediumThumbnail _:mediumThumb;
         ex:belongsTo ?album.
}.
```

It seems as if the response (in http:resp) to POSTing the image is the image itself. However, REST accesses resources through representations: we send a representation of the image and receive another, augmented with links.

This description captures an action that requires a link to an album and, independently thereof, an image. The image can then be used in the body of a POST request on the album, and this will establish a belongsTo relation between the image and the book. Furthermore, the image will provide access to links for comments and thumbnails. Note that the previous description explained the smallThumbnail relation, so the two descriptions together inform an agent that, after uploading an image, it can retrieve the corresponding 80px-high thumbnail.

The fact that the consequent of every POST rule can be deduced if its preconditions are satisfied follows from our design choices, since the rules were created to contain the "hypothetical request" that makes the postconditions become true.

Upon critical inspection of the rule, an apparent contradiction should be clarified. The consequent of an N3 rule is a conjunction of triples, but $P \Rightarrow Q \wedge R$ implies $P \Rightarrow Q$. Thus, omitting the HTTP request from the rule is a semantically valid operation. Unfortunately, this doesn't correspond to reality: the mere existence of an album and an image does not necessarily mean they are connected in any way. Yet, we must accept the limitations of first-order logic: it doesn't have a time aspect; everything that *can* be true *is* instantaneously true. Therefore, any POST request that *can* be executed *is* assumed to be "executed", or at least, its effects can serve as input for other rules. When used to our advantage, this is a useful property for planning.

Hypermedia-driven execution

The role of descriptions

Describing Web APIs is only one part of the solution. Software agents have to consume these descriptions as part of their process to meet a certain goal set out by the user. This might seem contradictory, since the hypermedia constraint demands that the interaction be driven by hypermedia controls in order to guarantee the independent evolution of client and server; out-of-band information should not be necessary to engage in the interaction. However, there are two remarks on this.

First, for humans, it is straightforward to use hypermedia, as we often have *implicit* out-of-band knowledge of what we want to achieve. For instance, suppose we want to buy a certain book online. Before we even start, we know we will end up on a site that offers several books for sale. One way to start would be to type the book's title into a search engine, which (as we expected) will give us links to book sites. We can click one of them, and we assume there will be a link "add to shopping basket" or similar. Before we click that link, we know this will eventually let us pay for the items in the basket and choose a delivery method. So while the actual interaction is driven by hypermedia, the driver behind the process is *planning*, which is based on expectation and intuition, implicit forms of out-of-band information. Machines don't have this kind of intuition, and the descriptions provide the expectations they need for planning the interaction—without changing the fact that each individual step is driven by hypermedia. Descriptions merely help machines look beyond the direct next steps each hypermedia document offers.

The act of navigating in a hypermedia-driven way is sometimes expressed colloquially as "following your nose". Indeed, at each step, you can look around and choose where to go next, as if by chance. However, following one's nose is easy for people, as we ultimately know the overall direction in which we're heading. In fact, this is a form of out-of-band information—and clearly not a bad thing.

Second, the problem with out-of-band knowledge in REST architectures is that this information is in practice interpreted by a human and then hard-coded into the system. For instance, in an RPC API, where the interaction is *not* driven by hypermedia, the steps have to be preprogrammed in the client software. This limits the client's capabilities to a certain API—even to a specific version of this API. In contrast, RESTdesc descriptions are discovered by the *agent* and at *runtime*. Thereby, the agent remains uncoupled from any specific API and thus allows for an independent evolution of client and server. In that sense, machine-interpretable descriptions discovered at runtime are *not* out-of-band. They form the information that helps to use the application, like natural language text on a novel Web application guides the user who cannot rely on prior expectations.

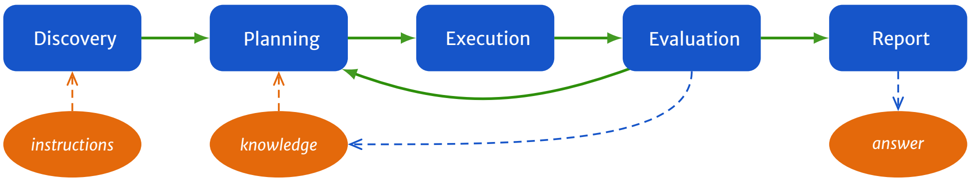
Most descriptions have traditionally been used at *design time*, where they indeed served an-out-of-band role. The application was compiled against a certain description, and then unable to work with other APIs or versions. RESTdesc descriptions are designed for runtime use.

In conclusion, runtime descriptions are no more *out-of-band* than intuition and expectation, which makes them harmless.

The interaction from begin to end

The agent should be as generic as possible, so it doesn't need any domain-specific knowledge; only an understanding of HTTP.

Having discussed the role of descriptions for autonomous agents, we now investigate how the actual hypermedia interaction will happen. As summarized in the schema below, the agent starts from a set of instructions and discovers descriptions that it uses to create a plan, based on the currently available knowledge. The first step of the plan is executed, and its results are evaluated and added to the knowledge base. If the goal is not reached yet, a new plan is created from the current starting point and the loop continues. Otherwise, the agent reports an answer (or error) back to the user.



Receiving instructions First, the user sends instructions to the agent. One way is to set a certain goal that must be met, given some background knowledge. For instance, the background knowledge here includes the fact that the user has an online photo album:

```
<http://example.org/profiles/lisa>↩
  ex:hasAlbum <http://example.org/albums/453>.
```

The user's goal is to obtain a thumbnail of a chosen image:

```
<http://www.w3.org/images/logo>↩
  dbpedia-owl:thumbnail ?logoThumbnail.
```

Humans can deal with incompleteness: we often have an idea of the initial and last steps, and assure ourselves we'll find a way to get through the middle. Machines need concrete plans: only if all steps are there, they can determine if the goal can be reached. Therefore, they absolutely require a description for *each* step.

Discovering descriptions Next, the agent needs to find descriptions to understand the possible actions. Broadly speaking, there are three ways to make this happen. First, the hypermedia-driven way would be to start from the background knowledge. For instance, starting from the photo album, the agent looks for links toward descriptions. These might be organized similarly to current human-readable API documentation pages, with a homepage leading to deeper topics. However, it might be difficult to find the right starting point in a large knowledge base. Therefore, a second approach is to consult an index or repository of descriptions (similar to search engines). In this case, the query would consist of the dbpedia-owl:thumbnail predicate. The third option is to dereference this predicate, since it might link to relevant descriptions.

Planning Once the descriptions have been retrieved, the agent creates a plan that, given the background knowledge, finds steps that lead toward the goal. These steps use resources and their links as high-level concepts. For instance, the plan in our example will instruct to post the image to an album, and then to follow the `smallThumbnail` link—without detailing the specific URLS (as those are yet unknown).

The technique to generate a plan from RESTdesc descriptions is explained in detail in the next chapter. A plan's core consists of *instantiated descriptions*.

Executing The first step of the plan is read and executed through hypermedia. Even though simply phrased, this is the crucial step that makes this method different from others. "Through hypermedia" means that the step, an instantiated description, will guide the action, although hypermedia is used to execute it. Concretely, in our example, the description's antecedent

Execution is similar to how we browse webpages: *guided* by some plan, but *driven* by hypermedia.

```
?profile ex:hasAlbum ?album.
?image a dbpedia:Image.
```

will have been instantiated in the plan with background knowledge as

```
<http://example.org/profiles/lisa>↩
  ex:hasAlbum <http://example.org/albums/453>.
<http://www.w3.org/images/logo> a dbpedia:Image.
```

The instantiation can be performed by a regular N3 reasoner, as will be detailed in the next chapter. The agent does thus not need N3 parsing or manipulation; regular RDF knowledge is sufficient.

and, as a result of this binding, the request in the consequent will be

```
_:request http:methodName "POST";
  http:requestURI <http://example.org/albums/453>;
  http:body <http://www.w3.org/images/logo>;
```

Executing this request directly is difficult, because we don't know in what format we should send the image. Instead, we consult the original resource `/albums/453`, asking for a machine-readable representation, and look for the form that allows uploading an image. This will tell us whether we have to send the URL directly or, for instance, a JPEG representation. Therefore, through the hypermedia controls inside the representation of `/albums/453`, the agent uploads the image. Even before it performs the upload, the rest of the instantiated description conveys expectations of what will happen:

```
<http://www.w3.org/images/logo>↩
   ex:comments _:comments;
   ex:smallThumbnail _:thumb;
   ex:mediumThumbnail _:mediumThumb;
   ex:belongsTo <http://example.org/albums/453>.
```

Only image and album have been instantiated, because they were bound variables in the description; the others were not.

In other words, the uploaded image will have comments, thumbnails, and will belong to the album. Note how the actual URLs of the comments and thumbnails are still unknown; they will only be filled out once the request has been executed. Nonetheless, the fact that there will be *some* thumbnail link has been sufficient for the planner to schedule the next step, which is to follow the concrete `smallThumbnail` link once the server sends it.

Replanning is important. In the simplest case, if everything went as expected, then the new plan is simply the current plan without its already completed first step. This stage is where the expectation will be checked against reality (in contrast to RPC-style interactions, where the entire control flow is fixed). It enables the agent to accurately respond to any situation at hand.

Evaluating and replanning The challenge of interacting with distributed systems is that things don't always go according to plan. Even though the agent has several assumptions about what the response will look like, there can never be a guarantee. Instead of steadily continuing with the plan, the agent inspects the hypermedia response and extracts all machine-interpretable knowledge. In the worst case, the response crucially differs from the expectation, and the goal will have to be reached in another way. But in the best case, the response brings us actually closer to the goal, maybe even more than anticipated. We're all familiar with this aspect on the Web: navigation happens serendipitously. The agent then verifies whether the goal state has been reached. If not, the background knowledge is augmented with the extracted knowledge, and the agent goes back to the planning step. The same goal still has to be reached, but it should now be closer than before (even if this means we'll have to find a different way).

Here, the new plan will contain the concrete link to the thumbnail, which was found inside the image representation returned by the server after the upload. In the subsequent execution step, the agent will simply have to `GET` the link's target to obtain the thumbnail.

In addition, real-world effects of the action might also have occurred, if they were part of the goal.

Reporting If the evaluation phase reveals the goal has been reached, then the answer is reported back to the user. Should, for any reason, the goal turn out to be unreachable, then the current status and the reason for failure are displayed. The same happens with any irrecoverable errors that cannot be solved by replanning.

This process indicates how descriptions and hypermedia can work together to support dynamic complex interactions between a client and a server in an evolvable way. Even though the descriptions and the corresponding HTTP request have been instantiated in the initial plan, the hypermedia response is inspected at every step and the current plan is adjusted according to the obtained result. Furthermore, the application state will only be advanced through the hypermedia controls supplied by each representation.

Limitations

RESTdesc does not strive to be a solution for all semantic agent requirements. Rather, it focuses on performing well in a broad range of cases. Below, we discuss limitations and possible coping strategies.

RESTdesc is expressed in N3 rules, which are implications in a **monotonic first-order logic** system. This limits what we can express. In particular, monotonicity means that we cannot retract statements after they have been asserted, which can give rise to inconsistencies. For instance, suppose a photograph is either private or public, and that this can be changed through a `PUT` request. Then we have a description that expresses *"if the visibility is 'public', then it can become 'private'"*. However, since a first-order world has no notion of time (everything that *can* be true *is* true), this implies that the photograph is private, contradicting the fact that it was public. However, this is seldom a problem in practice. First, reasoning under constraints is always difficult; therefore, over-restrictive knowledge should be avoided (as is common with ontologies as well). Second, methods such as `PUT` and `DELETE` are already precisely specified by HTTP and do not need application-specific clarification, unlike `POST`. Third, the interaction process demands replanning in every step. Therefore, facts that conflict with acquired knowledge can be omitted from later stages, as they are no longer relevant.

`DELETE` is an especially tricky method: in monotonic logic, a given thing that exists cannot *unexist*. However, we could mark it as "deleted" with a flag.

Next, RESTdesc descriptions rely on the hypermedia constraint, as they assume a **link in the precondition**. However, such a link is not always present: what if we want to reuse a resource from one Web API in another, but the APIs do not link to each other? The answer is to use descriptions that make the link on the semantic level instead:

Without hypermedia, the glue between two related resources isn't a control, but rather some common piece of information.

```
{ ?book dbpedia-owl:isbn ?isbn. }
=>
{
  _:request http:methodName "GET";
            http:requestURI
              ("http://books.org/" ?isbn "/cover");
            http:resp [ http:body _:cover ].
  ?book dbpedia-owl:thumbnail _:cover.
}.
```

For brevity, we use a list to express URI construction, but more sophisticated mechanisms such as URI templates offer a flexible alternative.

The above description expresses that if you have the ISBN number of a book, you can construct a URL that will lead to an image of its cover. This allows an agent to go from any page to the book API, even if that page doesn't provide the link. While this is hardly hypermedia-driven, when the page doesn't afford the action we need, it's our only option.

Cases of missing links will be discussed in detail in Chapter 6, where descriptions without a link in the antecedent are crucial.

Alternative description methods

Web service descriptions

The idea of describing dynamic interactions on the Web has been around for a long time, with substantially varying approaches depending on the underlying technology. The first generation of dynamic content was brought by *Web services*, the idea of which is to exchange messages (mostly in XML) over HTTP in a way that enables remote procedure calling. The most widely-known Web service protocol is the **Simple Object Access Protocol (SOAP)** [7]. A client sends a SOAP XML message to a server, typically containing a specific action name and its parameters, to which the server replies with another SOAP XML message. Note that SOAP neither uses hypermedia nor conforms to other REST uniform interface principles, as it works with an action/message-centric rather than a resource-oriented model.

We haven't discussed Web services so far, because I don't consider them first-class Web citizens. They exist and have a use, but they operate by a *separate* protocol on top of HTTP (or even something else) and thus don't integrate with hypermedia at all. REST APIS instead function on the level of the Web.

Interacting with SOAP services requires out-of-band knowledge. The **Web Service Description Language (WSDL)** [4] describes the interactions that are possible and what each message should look like. According to its specification, WSDL also allows to describe the abstract functionality provided by Web services. However, the definition of "functionality" is different from what we've assumed in this chapter. In a WSDL description, the `interface` element explains the supported operations of the service and the input and output parameters each operation requires. The notion of functionality is thus limited to the knowledge of a set of supported method names and their parameters, similar to a *method signature* in statically typed programming languages, albeit in a generic and platform-independent way. As a result, WSDL descriptions do not provide sufficient information for machines to decide whether the functionality offered by the service matches their current goals.

A WSDL document can be compared to a header file of a shared library on a local system. Such a file details in a machine-processable way how to interact with the library, but does not explain its provided actions.

In contrast, "functionality" in the context of RESTdesc means that machines are able to match the description of a Web API to their own knowledge base and/or goals in order to determine whether the API performs an action that is meaningful to what they want to achieve. It implies "understanding" in the Semantic Web sense of the word—interpreting information by relating it to known concepts in a way that enables acting upon that information.

WSDL descriptions can serve as a contract during development. Tools can automatically generate code for the communication with WSDL-described services, expressing them in a programming language's usual abstractions. Unfortunately, this closely ties applications to the services offered by one server at a specific moment in time.

SOAP tries to fit the Web into the classical programming paradigm, which is not built to withstand constant evolution.

Web services played an important part in the initial Semantic Web vision [3]. Therefore, much of the early work focused on making services accessible for machines through *semantic service descriptions*. One of the results of those efforts is **OWL for Services (OWL-S)** [15]. OWL-S descriptions are expressed in RDF and consist of three parts: a *profile*, a *process model*, and a *service grounding*. The profile advertises what the service offers, using *input and output parameters*, *preconditions* and *results*. These last two are expressed as literals in specialized languages, embedded in the main RDF document. They capture functionality in the Semantic Web sense, explaining the result of a service invocation in terms of parameter relations. The process model is meant to detail the interaction more precisely and comes into play when the client actually wants to use the service. Finally, the grounding explains how the parameters are captured in an actual exchange between a client and a server.

The Web Service Modeling Ontology (WSMO) is the other well-known semantic Web service description method [14].

Besides the focus on Web services instead of REST Web APIs, the difference between OWL-S and RESTdesc lies in the way they express functionality. First of all, OWL-S describes services on the meta-level, whereas RESTdesc conveys functionality on an API's application domain level. Thus, in an OWL-S document, the interaction is described in terms of parameters, whereas RESTdesc uses the application's concepts directly, thereby not enforcing a particular vocabulary or terminology. This is possible because RESTdesc assumes the underlying API conforms to the REST constraints, which demand resource-orientation.

OWL-S comes with WSDL support built in. However, it allows for extension with other groundings, for instance, SPARQL [21].

Second, the interpretation of conditions and results in OWL-S is not integrated, as these are expressed in external languages. Therefore, the interpretation of the RDF semantics of an OWL-S document does not imply an understanding of its functionality. With RESTdesc, the interpretation of the description and its functionality are integrated: the ability to parse N3 implies a correct interpretation of RESTdesc.

OWL-S supports three expression languages by default, while enabling the addition of others such as N3 [22].

Third, RESTdesc describes the request together with the functionality, while OWL-S separates the description of what the service does from how this is achieved (although it only supports SOAP natively). While this gives WSDL more flexibility, it comes with a considerable overhead. Support for other groundings in RESTdesc was not considered, as we especially target hypermedia APIs.

RESTdesc's integration of a function and the HTTP request that affords it helps composition, as the next chapter will explain.

Finally, RESTdesc descriptions are substantially shorter than their OWL-S counterparts. A RESTdesc description contains typically between 10 and 20 lines of RDF, while OWL-S usually takes a hundred lines or a multiple thereof. Brevity was an important RESTdesc design consideration, which is partly enabled by the assumption of the REST constraints.

The information density of RESTdesc is rather high, allowing to understand descriptions at a glance.

Web API descriptions

Together with several fellow API researchers, I've written a more comprehensive survey on Web API description [18]. This section provides a brief overview of our findings.

As Web APIs surpassed the popularity of Web services, methods for describing Web APIs emerged. This is still an ongoing research topic, so no method is widely adopted yet. Many methods don't require full compliance with all REST constraints, the most neglected one being the hypermedia constraint. "Web APIs" are therefore temporarily treated synonymously to "HTTP APIs" here, as opposed to the term "hypermedia APIs", which signals actual hypermedia-driven REST APIs.

The **Web Application Description Language (WADL)** [8] can be considered the native HTTP equivalent of the messaging-driven WSDL. It can describe resources and the links between them, albeit in a way that tends to be more RPC-oriented. Its serialization format is an XML document that details resource types and their methods syntactically, without offering any form of functionality.

Several methods extend existing documents with annotations in order to capture extra semantics. **MicrowsMO** [12], built on top of the HTML-based format **hRESTS** [10], uses microformats to add machine-interpretable information to human-readable documentation. Yet, these annotations don't describe a functional relation. MicrowsMO also offers *lifting* and *lowering*, the transformation of representation formats, whereas RESTdesc assumes this is handled by the server or another intermediary. **Semantic Annotations for WSDL (SAWSDL)** [11] offer comparable functionality, but extend WSDL instead. It offers functionality in the form of preconditions and effects. The **Minimal Service Model** [16] approaches Web service and Web API description in an operation-based manner, aiming to capture functional aspects in addition to parameters and methods.

The realization that the expressivity of RDF is too limited to directly express dynamic processes has inspired several methods based on other expression languages. The common denominator is the necessity for variables and quantification, which are easily represented in SPARQL-like graph patterns. **Linked Open Services** [13] and **Linked Data Services** [17] are two such approaches. As in OWL-S, graph patterns are expressed as literals, making the interpretation of descriptions non-integrated.

RESTdesc differs from the above approaches in its focus on hypermedia APIs and a direct level of description without the use of service elements such as parameters. It offers succinct descriptions that allow the discovery of Web APIs based on desired functionality. Furthermore, it has a built-in composition mechanism, which is the subject of the next chapter.

People browsing hypermedia applications know how to proceed after every page by relying on expectations and natural language, which is impossible for machine clients. RESTdesc descriptions therefore help machines look beyond the hypermedia controls offered in a single step by providing expectations of what can happen. Descriptions are instantiated into a plan to achieve a specific goal, given certain background knowledge. The interaction itself is guided by the plan, but remains driven by hypermedia: after each step, an agent reacts to a hypermedia response by interpreting it and replanning accordingly.

References

[1] Mike Amundsen. Hypermedia APIS with HTML5 and Node. O'Reilly, December 2011.

[2] Tim Berners-Lee and Dan Connolly. Notation3 (N3): a readable RDF syntax. Team Submission. World Wide Web Consortium, 28 March 2011. *http://www.w3.org/TeamSubmission/n3/*

[3] Tim Berners-Lee, James Hendler, and Ora Lassila. The Semantic Web. *Scientific American*, 284(5):34–43, May 2001.

[4] David Booth and Canyang Kevin Liu. Web Services Description Language (WSDL) version 2.0 part 0: primer. Recommendation. World Wide Web Consortium, June 2007. *http://www.w3.org/TR/wsdl20-primer/*

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

[6] Roy Thomas Fielding, Jim Gettys, Jeffrey Mogul, Henrik Frystyk, Larry Masinter, Paul Leach, and Tim Berners-Lee. Hypertext Transfer Protocol (HTTP). Request For Comments 2616. Internet Engineering Task Force, June 1999. *http://tools.ietf.org/html/rfc2616*

[7] 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. Recommendation. World Wide Web Consortium, 27 April 2007. *http://www.w3.org/TR/soap12-part1/*

[8] Marc Hadley. Web Application Description Language. Member Submission. World Wide Web Consortium, 31 August 2009. *http://www.w3.org/Submission/wadl/*

[9] Johannes Koch, Carlos A. Velasco, and Philip Ackermann. HTTP vocabulary in RDF 1.0. Working Draft. World Wide Web Consortium, 10 May 2011. *http://www.w3.org/TR/HTTP-in-RDF10/*

[10] Jacek Kopecký, Karthik Gomadam, and Tomas Vitvar. hRESTS: an HTML microformat for describing RESTful Web services. *Proceedings of the 2008 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology*, pages 619–625, IEEE Computer Society, 2008.

[11] Jacek Kopecký, Tomas Vitvar, Carine Bournez, and Joel Farrell. SAWSDL: semantic annotations for WSDL and XML schema. *Internet Computing, IEEE*, 11(6):60–67, 2007.

[12] Jacek Kopecký, Tomas Vitvar, and Dieter Fensel. MicroWSMO and hRESTS. Technical report D3.4.3. SOA4All, March 2009. *http://sweet.kmi.open.ac.uk/pub/microWSMO.pdf*

[13] Reto Krummenacher, Barry Norton, and Adrian Marte. Towards Linked Open Services and processes. In: *Future Internet Symposium*. Volume 6369 of Lecture Notes in Computer Science, pages 68–77. Springer, 2010.

[14] Rubén Lara, Dumitru Roman, Axel Polleres, and Dieter Fensel. A conceptual comparison of WSMO and OWL-S. In: Liang-Jie Zhang and Mario Jeckle, editors, *Web Services*. Volume 3250 of Lecture Notes in Computer Science, pages 254–269. Springer, 2004.

[15] David Martin, Mark Burstein, Drew McDermott, Sheila McIlraith, Massimo Paolucci, Katia Sycara, Deborah Louise McGuinness, Evren Sirin, and Naveen Srinivasan. Bringing semantics to Web services with OWL-S. *World Wide Web*, 10(3):243–277, September 2007.

[16] Carlos Pedrinaci, Dong Liu, Maria Maleshkova, David Lambert, Jacek Kopecký, and John Domingue. iServe: a linked services publishing platform. *Proceedings of the Ontology Repositories and Editors for the Semantic Web Workshop*, June 2010.

[17] Sebastian Speiser and Andreas Harth. Integrating Linked Data and services with Linked Data Services. In: *The Semantic Web: Research and Applications*. Volume 6643 of Lecture Notes in Computer Science, pages 170–184. Springer, 2011.

[18] Ruben Verborgh, Andreas Harth, Maria Maleshkova, Steffen Stadtmüller, Thomas Steiner, Mohsen Taheriyan, and Rik Van de Walle. Survey of semantic description of REST APIs. In: Cesare Pautasso, Erik Wilde, and Rosa Alarcón, editors, *REST: Advanced Research Topics and Practical Applications*, pages 69–89, 2014.

[19] Ruben Verborgh, Thomas Steiner, Davy Van Deursen, Sam Coppens, Joaquim Gabarró Vallés, and Rik Van de Walle. Functional descriptions as the bridge between hypermedia APIs and the Semantic Web. *Proceedings of the Third International Workshop on RESTful Design*, pages 33–40, ACM, April 2012.

[20] Ruben Verborgh, Thomas Steiner, Davy Van Deursen, Jos De Roo, Rik Van de Walle, and Joaquim Gabarró Vallés. Capturing the functionality of Web services with functional descriptions. *Multimedia Tools and Applications*, 64(2):365–387, May 2013.

[21] Ruben Verborgh, Davy Van Deursen, Jos De Roo, Erik Mannens, and Rik Van de Walle. SPARQL endpoints as front-end for multimedia processing algorithms. *Proceedings of the 4th Workshop on Service Matchmaking and Resource Retrieval in the Semantic Web*, November 2010.

[22] Ruben Verborgh, Davy Van Deursen, Erik Mannens, Chris Poppe, and Rik Van de Walle. Enabling context-aware multimedia annotation by a novel generic semantic problem-solving platform. *Multimedia Tools and Applications*, 61(1):105–129, November 2012.

Chapter 5

Proof

I don't need to fight
To prove I'm right
I don't need to be forgiven

— *The Who, Baba O'Riley (1971)*

Proofs justify how we arrived at a conclusion given a set of facts and rules that we're allowed to apply. On the Semantic Web, a proof lets a machine explain precisely how it obtained a certain result. Although usually reserved for static data, in this chapter, proofs will play a crucial role in the dynamic world of Web APIs. With some creativity, they can guarantee the correctness of a composition of several Web APIs before its execution, and even serve as an efficient method to automatically create such compositions.

The ability to *prove* that a conclusion is correct has become one of the foundations of science. A proof justifies a statement by decomposing it into more elementary pieces, which may only be combined using a strictly constrained methodology. Those pieces can in turn be proven, until we arrive at fundamental elements that we cannot decompose and have chosen to accept as truth. The mechanisms of proof allow us to discover knowledge derived from the truth—and to distinguish that from what is false.

Each individual argument needs to be verifiable, as contradicting facts lead to the *principle of explosion*: an inconsistency allows to conclude anything.

In a world where actions will be undertaken autonomously by machines, an important question is whether these actions and their results represent the intentions of the person who instructed those machines. Therefore, it shouldn't come as a surprise that the notion of "proof" was already present in the initial Semantic Web vision [9]. Results obtained by machines can be trusted when accompanied by an independently verifiable, machine-readable proof document. Such proofs can then be exchanged by different parties [12] and verified by dedicated *proof checkers*.

Automated agents require an even higher trust level than most Web applications, because people are no longer *directly* in control of decisions.

Additionally, **trust** is an important aspect. If a proof uses a certain piece of data as justification, then you need to be able to decide whether you can rely on its source. Together with **digital signatures** that allow to verify the authenticity of the information, trust records ensure that the foundations of the proof are valid [9]. Certainly in an environment where *"anyone can say anything about anything"* [7], we need to be selective as to what information we let our conclusions be built upon.

The apparent shortage of "Linked Rules" is rather unfortunate, since a main N3Logic goal was precisely to *share* information that requires more expressivity than regular RDF [8].

Unfortunately, proof on today's Web is still at an early stage. In Chapter 3, we discussed Linked Data as a pragmatic view on the Semantic Web—and so far, proofs didn't attract a lot of attention in the Linked Data ecosphere. The focus on raw data might explain this, as things then become more a question of trust than proof. However, many of the large datasets, such as DBpedia and Freebase, provide data that originates from other sources. So at the very least, full trust would require the **provenance** information of the original data, but a proof that the resulting data has been correctly derived would be necessary for total certainty. This illustrates a balance between trust and proof: ultimately, we must accept some axioms, similar to the choices mathematics and natural sciences have to make. Yet the more we emphasize a verifiable proof, the less we need to (blindly) trust.

As we would expect, N3 proofs generated by one reasoner can be parsed and interpreted by another. Interoperability is as crucial as with data.

On the bright side, many current N3 reasoners provide support to prove the conclusions drawn from triples and rules. As part of the Semantic Web Application Platform [6], the cwm reasoner and an ontology with elementary components like `Proof` and `Inference` were created [5]. The ontology is usually referred to by the `r` prefix and the corresponding URL *http://www.w3.org/2000/10/swap/reason#*. It provides the means to explain formally and in meticulous detail how and why a reasoner was able to derive a certain set of facts. Conveniently, an automated proof checker is available [5].

In this chapter, we will first discuss the essential components of N3 proofs with static data and rules. Next, we will incorporate dynamic information in those proofs by using RESTdesc descriptions of Web APIs. This then leads to the question of how we can be sure that a given *composition* of Web APIs realizes desired functionality, and how such compositions can be created automatically. Finally, we'll explain the role of proofs in the hypermedia-driven execution process of autonomous agents.

Understanding proofs

To understand how N3 reasoners generate proofs and how these proofs are structured, we will study the N3 proof of a classical syllogism. It expresses perhaps the most famous example of inference:

Ancient syllogisms have had a profound influence on deductive reasoning.

> *Socrates is a man, and all men are mortal.*
> *Therefore, Socrates is mortal.*

This translates into predicate logic as follows:

$$\frac{\begin{array}{r} \mathit{man}(\text{Socrates}) \\ \forall x\ (\mathit{man}(x) \implies \mathit{mortal}(x)) \end{array}}{\mathit{mortal}(\text{Socrates})}$$

Our goal is to obtain this conclusion from an N3 reasoner. To that end, we first have to represent the initial knowledge as RDF. The triple below has been found at *http://dbpedia.org/resource/Socrates*:

Storing data and rules as separate resources allows independent reuse.

```
dbpedia:Socrates a dbpedia-owl:Person.
```

The implication can be represented as follows in an N3 document:

```
{ ?person a dbpedia-owl:Person. }
=>
{ ?person a dbpedia-owl:Mortal. }.
```

We could present both resources to an N3 reasoner and demand to derive all possible statements that can be entailed. However, this might not always be practical: the number of entailed triples can potentially be enormous, even with a moderate number of rules, and not all of those triples are relevant to solve the given question. In the worst case, rules can trigger recursively and lead to an infinite stream of triples, which can of course never be generated.

Instead, we'll ask a specific query: all triples that have Socrates as the subject. This is the graph pattern "`dbpedia:Socrates ?p ?o`." Most N3 reasoners have a specific query mechanism called *filter rules*, which are N3 rules used in a similar way to SPARQL `CONSTRUCT` queries. A filter rule's antecedent instructs the reasoner to find all matching patterns, which are then shaped according to the rule's consequent. Since we are interested in "Socrates" triples, and we want to retrieve them exactly as they are, our filter rule becomes:

Any rule in which the antecedent and consequent are the same might seem a strange tautology. After all, $P \Rightarrow P$ always holds, so why include it then? The answer is that filter rules are not knowledge rules; they instruct a reasoner to find the graph P and to derive a graph P', which could (but doesn't need to) be identical to P.

```
{ dbpedia:Socrates ?p ?o. }
=>
{ dbpedia:Socrates ?p ?o. }.
```

The reasoner is asked to execute this query on the given N3 documents.

Figure 3 lists an example proof generated by the EYE reasoner [10] in response to our input and query, using the following command:

```
eye socrates.ttl mortal.n3 --query query.n3
```

Query results can contain both pre-existing and entailed triples.

The query execution results in the following triples:

```
dbpedia:Socrates a dbpedia-owl:Person.
dbpedia:Socrates a dbpedia-owl:Mortal.
```

We will now go through the proof to understand how the reasoner arrived at this conclusion, thus starting from the result and heading toward the initial facts. At the highest level, a proof consists of a `Proof` entity, which is a `Conjunction` of different `components`. In this case, those components are `#lemma1` and `#lemma2`. The proof `gives` the two Socrates triples, the derivation of which is detailed by these Lemmata 1 and 2, which in turn have their own justification. There are two possible lemma types: `Inference` and `Extraction`.

For consistency reasons, the filter rule will always be instantiated explicitly, even if it is a *pass-through* rule that simply returns the same, as in this case. The reasoner is obliged to instantiate the filter rule before arriving at the final conclusion; its usage can in fact be considered the reasoner's goal.

Lemma 1 is an inference that results in the fact that Socrates is a man. It might seem surprising to see this fact is the result of an inference rather than a simple extraction, as it was also present in the input files. However, the `rule` used for this lemma is the special filter rule (Lemma 4), which was instantiated with the triple itself (Lemma 3), leading indeed to this conclusion. In other words, Lemmata 3 and 4 detail the origin of the knowledge *("Socrates is a man")* and the filter rule *("find a triple about Socrates and return it")*, whereas Lemma 1 details the instantiation of this rule with the knowledge using `rule` and `evidence` predicates. Extractions such as `#lemma3` and `#lemma4` don't require further proving, as they directly point to the source that must be parsed to obtain the triple or rule.

Lemma 2, the other component of the main proof, also shows an application of the filter rule, as indicated by its `rule` property. This lemma results in the fact that Socrates is a mortal. However, the `evidence` isn't an extraction this time, but another inference detailed in Lemma 5. This lemma then derives Socrates' mortality by applying the *"if human, then mortal"* rule (Lemma 6) on the fact that Socrates is human (Lemma 3). As the `binding` details, the rule is instantiated by replacing the variable `?person` by `dbpedia:Socrates`:

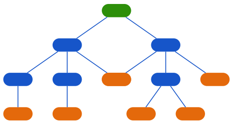
Interestingly, proving the "main" inference is only a small part of the whole.

```
{ dbpedia:Socrates a dbpedia-owl:Person. }
=>
{ dbpedia:Socrates a dbpedia-owl:Mortal. }.
```

This leads to the conclusion of Lemma 5, which was picked up by the filter rule in Lemma 2 and finally propagated to the main proof.

```
@prefix dbpedia: <http://dbpedia.org/resource/>.
@prefix dbpedia-owl: <http://dbpedia.org/ontology/>.
@prefix var: <var#>.
@prefix r: <http://www.w3.org/2000/10/swap/reason#>.
@prefix n3: <http://www.w3.org/2004/06/rei#>.

<#proof> a r:Proof, r:Conjunction;
  r:component <#lemma1>, <#lemma2>;
  r:gives {
    dbpedia:Socrates a dbpedia-owl:Person.
    dbpedia:Socrates a dbpedia-owl:Mortal.
  }.

<#lemma1> a r:Inference; r:gives { dbpedia:Socrates a dbpedia-owl:Person };
  r:evidence (<#lemma3>);
  r:binding [ r:variable [ n3:uri "var#p" ];
              r:boundTo  [ n3:uri "http://www.w3.org/1999/02/22-rdf-syntax-ns#type" ]];
  r:binding [ r:variable [ n3:uri "var#o" ];
              r:boundTo  [ n3:uri "http://dbpedia.org/ontology/Person" ]];
  r:rule <#lemma4>.

<#lemma2> a r:Inference; r:gives { dbpedia:Socrates a dbpedia-owl:Mortal };
  r:evidence (<#lemma5>);
  r:binding [ r:variable [ n3:uri "var#p" ];
              r:boundTo  [ n3:uri "http://www.w3.org/1999/02/22-rdf-syntax-ns#type" ]];
  r:binding [ r:variable [ n3:uri "var#o" ];
              r:boundTo  [ n3:uri "http://dbpedia.org/ontology/Mortal" ]];
  r:rule <#lemma4>.

<#lemma3> a r:Extraction; r:gives { dbpedia:Socrates a dbpedia-owl:Person };
  r:because [ a r:Parsing; r:source <socrates.ttl> ].

<#lemma4> a r:Extraction; r:gives { @forAll var:p, var:o.
            { dbpedia:Socrates var:p var:o } => { dbpedia:Socrates var:p var:o }};
  r:because [ a r:Parsing; r:source <query.n3> ].

<#lemma5> a r:Inference; r:gives { dbpedia:Socrates a dbpedia-owl:Mortal };
  r:evidence (<#lemma3>);
  r:binding [ r:variable [ n3:uri "var#person" ];
              r:boundTo  [ n3:uri "http://dbpedia.org/resource/Socrates" ]];
  r:rule <#lemma6>.

<#lemma6> a r:Extraction; r:gives { @forAll var:person.
            { var:person a dbpedia-owl:Person } => { var:person a dbpedia-owl:Mortal }};
  r:because [ a r:Parsing; r:source <mortal.n3> ].
```

Figure 3: A proof is a conjunction of components, recursively constructed out of inferences and extractions. This proof indicates that, if all men are mortal and Socrates is a man, then Socrates is mortal.



Each proof is a graph that starts from the conclusion, going through inferences to arrive at trusted facts.

From this example, it is apparent that N3Logic proofs, just like mathematical proofs, have a *recursive* structure. Each derived triple must be justified by a lemma, the assumptions of which have to be justified in turn, until we arrive at the parsing of input files, which are the axioms. At that level, acceptance becomes a matter of trust, unless the input files themselves are accompanied by a proof. Another observation is that proofs have a backward flow: they start from the conclusion, which is gradually decomposed into elementary parts. This might lead to confusion at first, since we will encounter results before we learn how they are derived. Yet, it fits the central philosophy of tracing back each fact to a derivation from other verifiable facts, until we arrive at the source.

Jim Hendler provides an example wherein a server states an agent is in debt. Reluctant to transfer the money without a cause, the agent asks for proof. The server then justifies its demand with evidence of some unpaid purchase. Now convinced, the agent sends the sum due [12].

In this context, the proof can be interpreted as a *dialog*. If one agent claims that Socrates is a mortal, another agent can ask why. In response, the first agent will say this is the result of applying a specific rule on the fact that Socrates is a human. Still unsatisfied, the other agent demands more details, upon which the first explains this fact was obtained from `socrates.ttl` with the rule extracted from `mortal.n3`. The other can then choose to ask the same questions to the data sources of these files, where it would perhaps learn that the contents of `socrates.ttl` were taken from DBpedia. The interrogation continues until the agent finds sources it trusts (or not, in which case it rejects the proof result because of unsure premises).

The proofs we have discussed so far employ pre-existing facts to justify a certain assertion. In contrast, we want to verify whether the execution of a series of Web API calls—which contain *dynamic* information that is not known beforehand—will deliver an intended result (without undesired side-effects). This guarantee is necessary for an agent before it can engage in complex interactions because, as we recall from the previous chapter, hypermedia allows agents to look only one step ahead. A proof that a particular step goes in the right direction provides the confidence needed to take that step.

We're again safeguarded by RESTdesc's choice to place the HTTP request as an existential in the conclusion: a proof will state a chain of requests matching the goal exists, without claiming success for all executions.

Since RESTdesc descriptions are expressed as N3 rules, they can also serve as inferences in proofs. What distinguishes RESTdesc rules from others is that they describe the expected results of Web API calls, which are not necessarily executed. So under the assumption that the employed RESTdesc descriptions accurately capture the result that will occur in reality, proofs with RESTdesc rules can be interpreted as Web API *compositions* that reach a certain goal. In addition to indicating *if* a goal can be fulfilled, the proof will explain *how*, by detailing a possible chain of HTTP requests.

Automatically generating compositions

Instead of applying proofs to verify the derivation of facts from static knowledge, we will now discuss a proof that shows how a goal can be achieved using Web API calls. Suppose an agent has access to the local image `lena.jpg` as part of its background knowledge:

```
<lena.jpg> a dbpedia:Image.
```

The goal set by the user is to obtain a thumbnail of that specific image:

```
{ <lena.jpg> dbpedia-owl:thumbnail ?result. }
=>
{ <lena.jpg> dbpedia-owl:thumbnail ?result. }.
```

To solve this problem, it has several Web API descriptions at its disposition, including the ones we have discussed in the previous chapter. The description below explains that `smallThumbnail` links lead to an 80px-high thumbnail:

Descriptions can be given explicitly or discovered automatically, as we will discuss in Chapter 7.

```
{ ?image ex:smallThumbnail ?thumbnail. }
=>
{
  _:request http:methodName "GET";
            http:requestURI ?thumbnail;
            http:resp [ http:body ?thumbnail ].
  ?image dbpedia-owl:thumbnail ?thumbnail.
  ?thumbnail a dbpedia:Image;
             dbpedia-owl:height 80.0.
}.
```

The following description captures the fact that images uploaded to the `/images/` resource receive `comments` and `smallThumbnail` links:

```
{ ?image a dbpedia:Image.  }
=>
{
  _:request http:methodName "POST";
            http:requestURI "/images/";
            http:body ?image;
            http:resp [ http:body ?image ].
  ?image ex:comments _:comments;
         ex:smallThumbnail _:thumb.
}.
```

Some details have been omitted from the upload description to simplify the proof slightly. Yet, using the exact description as in Chapter 4 would yield analogous results.

In practice, many more descriptions will be available; it is precisely the task of the reasoner generating the proof to select the relevant ones.

Parsing details were not listed; Lemmata 4 to 7 of the proof correspond to the respective snippets on the previous page.

Figure 4 shows the proof that was obtained by sending the background knowledge and several descriptions to a reasoner with the given query. As expected, its structure is similar to that of Figure 3: one main proof entity, recursively decomposed into inferences and extractions. The difference lies in the usage of RESTdesc descriptions to perform inferences. Since RESTdesc expresses functionality as regular N3 rules, reasoners do not need additional knowledge to incorporate them in proofs. At the same time, all the instantiated descriptions retain their additional operational meaning.

While proofs are constructed from the conclusion toward the initial assumptions, Web API executions start from an initial state to finally arrive at a goal. Therefore, in this discussion, we will follow the proof in the reverse direction, in order to highlight the connection to Web APIs. Since Lemmata 4 to 7 are merely extractions obtained through parsing, we will skip to the first inference that uses them.

At first sight, more existentials seem to appear in the proof than in the description. This is not the case: even though the `resp` triple is written in `[]` notation, it remains a blank node, which thus needs a new identifier.

Lemma 3 details the instantiation of the RESTdesc description for image uploading (Lemma 7). The knowledge that `lena.jpg` is an image (Lemma 4) satisfies that description's precondition, so it is triggered. Note how the `image` variable is bound to `lena.jpg`, whereas all existentially quantified variables are instantiated with *newly created* blank nodes, as can be seen by the `Existential` type. Here, the parametrized request in the RESTdesc rule's consequent has been instantiated to a `POST` request to `/images/` with a request body of `lena.jpg`. All necessary information to construct this request is thus in place. Even though the outcome of the request is unknown at this stage, the description stated there would be `comments` and `smallThumbnail` links. As their exact targets cannot be determined yet, they're represented by the new blank nodes `_:sk2` and `_:sk3`.

The usage of blank nodes in RESTdesc rules serves as a way to track values throughout a proof.

Lemma 2 continues from the obtained result that `lena.jpg` has a `smallThumbnail` link to `_:sk3`, instantiating this in the corresponding RESTdesc API description (Lemma 6). The actual thumbnail URL is undetermined, which is reflected in the incompleteness of the resulting API call—a `GET` request to `_:sk3` that leads to the thumbnail. While its actual value is still *undetermined*, `_:sk3` is not *unspecified*: it refers to the target of the `smallThumbnail` link that will be obtained through the `POST` request in Lemma 3. The hypermedia links and their semantics *propagate through the proof* as blank nodes, substitutes for concrete values that will be determined during the execution.

Finally, Lemma 1 is the obligatory instantiation of the filter rule that represents the agent's goal (Lemma 5). It takes the conclusion of Lemma 2—the existence of a `thumbnail`—and finds `_:sk3` as the needed result. Lemma 1 is thereby sufficient to conclude the proof.

```
<#proof> a r:Proof, r:Conjunction;
  r:component <#lemma1>;
  r:gives { <lena.jpg> dbpedia-owl:thumbnail _:sk3. }.

<#lemma1> a r:Inference;
  r:gives { <lena.jpg> dbpedia-owl:thumbnail _:sk3. };
  r:evidence (<#lemma2>);
  r:binding [ r:variable [ n3:uri "var#result"];
              r:boundTo  [ a r:Existential; n3:nodeId "_:sk3"]];
  r:rule <#lemma5>. # extracted by parsing agent_goal.n3

<#lemma2> a r:Inference;
  r:gives { _:sk4 http:methodName "GET".  _:sk4 http:requestURI _:sk3.
           _:sk4 http:resp _:sk5.        _:sk5 http:body _:sk3.
           <lena.jpg> dbpedia-owl:thumbnail _:sk3.
           _:sk3 a dbpedia:Image.        _:sk3 dbpedia-owl:height 80.0. };
  r:evidence (<#lemma3>);
  r:binding [ r:variable [ n3:uri "var#image"];
              r:boundTo  [ n3:uri "lena.jpg"]];
  r:binding [ r:variable [ n3:uri "var#thumbnail"];
              r:boundTo  [ a r:Existential; n3:nodeId "_:sk3"]];
  r:binding [ r:variable [ n3:uri "var#x2"];
              r:boundTo  [ a r:Existential; n3:nodeId "_:sk4"]];
  r:binding [ r:variable [ n3:uri "var#x3"];
              r:boundTo  [ a r:Existential; n3:nodeId "_:sk5"]];
  r:rule <#lemma6>. # extracted by parsing description_smallThumbnail.n3

<#lemma3> a r:Inference;
  r:gives { _:sk0 http:methodName "POST".  _:sk0 http:requestURI "/images/".
           _:sk0 http:body <lena.jpg>.
           _:sk0 http:resp _:sk1.         _:sk1 http:body <lena.jpg>.
           <lena.jpg> ex:comments _:sk2.  <lena.jpg> ex:smallThumbnail _:sk3. };
  r:evidence (<#lemma4>); # extracted by parsing agent_background_knowledge.ttl
  r:binding [ r:variable [ n3:uri "var#image"];
              r:boundTo  [ n3:uri "lena.jpg"]];
  r:binding [ r:variable [ n3:uri "var#x1"];
              r:boundTo  [ a r:Existential; n3:nodeId "_:sk0"]];
  r:binding [ r:variable [ n3:uri "var#x2"];
              r:boundTo  [ a r:Existential; n3:nodeId "_:sk1"]];
  r:binding [ r:variable [ n3:uri "var#x3"];
              r:boundTo  [ a r:Existential; n3:nodeId "_:sk2"]];
  r:binding [ r:variable [ n3:uri "var#x4"];
              r:boundTo  [ a r:Existential; n3:nodeId "_:sk3"]];
  r:rule <#lemma7>. # extracted by parsing description_upload.n3

<#lemma4> a r:Extraction. <#lemma5> a r:Extraction. # Parsing details omitted for brevity
<#lemma6> a r:Extraction. <#lemma7> a r:Extraction. # (must be present in an actual proof).
```

Figure 4: This proof shows that a thumbnail of `lena.jpg` can be obtained by a composition of a `POST` and a `GET` request. The properties of these requests are detailed in the proof through instantiated descriptions.

We can make three core observations about a composition proof, assuming that all of the used descriptions accurately reflect reality. First, the existence of the proof indicates the desired goal is *reachable* given the current knowledge and Web APIs. Independent of whether we'll execute the composition (and whether each of its steps will be successful), it is at least theoretically possible to reach the goal. This might not really seem a significant achievement, but remember that hypermedia-driven clients can only see the direct next steps; the steps thereafter are unclear for automated agents.

Proofs provide lookahead functionality that serves as a machine substitute for people's intuition on possible further steps.

Second, one way of *achieving the goal* is the Web API composition suggested in the proof. Even though not all of the parameters are fully determined yet—as they depend on the execution of earlier API calls—a possible plan to arrive at the goal is directly available. In fact, the proof *is* the composition and vice-versa. This doesn't mean *all* steps of the plan have to be followed; on the contrary, the interaction will happen dynamically through hypermedia.

Third, each composition has at least *one fully determined API call*. This is indeed a logical consequence of the structure of proofs: one of the present RESTdesc descriptions must have been instantiated solely with data from extractions or regular inferences. This request is usually the first that is executed through hypermedia. Furthermore, in those cases where relevant hypermedia controls are unavailable, the agent can execute the HTTP request as described in the proof. This happens for instance when two resources from different applications are not connected. For example, the RESTdesc description of the image upload relies on a hard-coded URL instead of hypermedia. This means we could take an image from any application and upload it into the current one, even if there is no control that affords this.

The trade-off between following hypermedia controls and executing HTTP requests directly is featured in the next chapter.

Although the example proof presented here was rather simple, more complex proofs can be realized. In particular, this proof's composition is a linear concatenation of Web APIs, whereas in the general case, complex interdependency patterns between API calls are possible. As proofs prohibit circular dependencies, an executable order of requests will always exist. Branches in the proof graph that are independent of others can be executed in parallel; the proof makes the dependencies visible through the various propagating placeholders represented by blank nodes.

Just like in any other proof, regular N3 rules and RDF or OWL ontological constructs can also serve as inferences. Mixing them with RESTdesc rules can lead to the derivation of new facts even before API calls have been executed—and these facts can then propagate to other API calls.

Ontologies compensate for possible vocabulary mismatches between API descriptions from various providers or applications.

Pragmatic proofs for planning

Pre-proofs and post-proofs

The method we've introduced for Web API composition should be considered a *pragmatic* proof [19]. We call it "pragmatic" because it realizes composition generation, at the cost of accepting that things can (and will) go wrong every once in a while—a characteristic that is usually unacceptable for proofs. Yet pragmatism is required for consumers of Web APIs: applications evolve constantly, so we trade the safety of fixed message exchanges for the flexibility of dynamic and serendipitous interactions.

Proofs are typically not regarded pragmatic, as they quite rigorously verify facts. Our pragmatic angle captures the usage of such a strict method to control a process that is inherently error-prone.

As a result, composition proofs rely on stronger assumptions than regular proofs. Whereas proofs are normally rooted in knowledge and rules that have to be trusted (or proven in turn), proofs with RESTdesc rules inherit the assumptions of these Web API descriptions. Concretely, any RESTdesc rule assumes a request exists that derives the postconditions from the preconditions. This allows the reasoner to propagate these postconditions throughout the proof. Should such request not exist (for instance, because the server has crashed), the proof is invalid because of unjustified premises. In order to make these assumptions explicit, we distinguish between two kinds of proof during the process of the execution:

- A **pre-execution proof *(pre-proof)*** assumes the execution of all described Web API calls will behave as expected.
- A **post-execution proof *(post-proof)*** contains static data as evidence, obtained through executing Web API calls.

As each proof details the sources that were used to generate it, we can automatically verify what the assumptions were and hence the degree of trust we need. If the extractions contain RESTdesc rules, we need to be aware of the extra degree of trust needed. Therefore, a recommended practice is to indicate this explicitly in resources that contain RESTdesc descriptions.

Any client could prohibit RESTdesc descriptions as fact sources to guarantee that requests are already executed.

Note that the above distinction between a pre- and post-proof is relative rather than absolute. A *complete* pre-proof is generated before any execution has taken place, such as as the proof in Figure 4. Analogously, the generation of a *complete* post-proof happens after all executions were performed. However, each execution can have its own pre- and post-proof. After one of the API calls from the complete pre-proof has been executed, a post-proof at this stage replaces that single API call by its results, which were determined by the execution.

The information gathered during proof creation can serve as provenance [11].

As an example, we will apply the concept of pre- and post-proofs on the image upload scenario, as part of the execution process detailed in the previous chapter. The proof in Figure 4 is a pre-proof and forms the agent's initial plan. It contains two API calls, a `GET` and `POST` request, and only the latter has all necessary parameters in place for execution. The agent therefore starts with the `POST` request. As there are no hypermedia controls in this particular case, the request is executed as listed in the proof.

In the previous chapter, the first step did involve hypermedia controls; it depends on the use case.

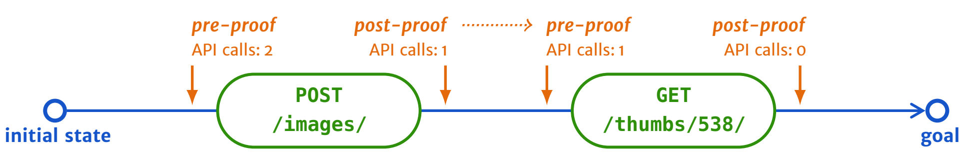
In response, the server returns a representation that, as expected from the API description, contains a `smallThumbnail` link. For instance, assume the link points to `/thumbs/538/`. The classical planning strategy would be to continue from the initial proof, as we now know that the existential `_:sk3` is bound to `/thumbs/538/`. In contrast, the agent first verifies the success of the request by generating a post-proof from the initial state, the response, and the API descriptions. If successful, this post-proof should contain one less API call, as the needed `smallThumbnail` triple is now present as a fact and needs not be derived. In case of failure, the post-proof will again suggest the same `POST` request or result in a contradiction. In that case, the corresponding description may not be used again, and a new pre-proof should be generated.

Reasoners will strive to obtain the shortest proof, so we're sure the number of APIs doesn't increase.

A successful *post*-proof generated after this first request can directly serve as a *pre*-proof to continue with the next request. Indeed, it will contain all remaining API requests. However, to make the request, the agent can simply follow the `smallThumbnail` link in the representation, as the necessary hypermedia control is present. The interaction thus continues in a hypermedia-driven way, with pre-proofs as a guideline toward the next step, and post-proofs to verify the correctness of the previous step. After following the link, the agent receives a thumbnail. The reasoner can generate another post-proof from this, but it will be rather short: the goal of having a thumbnail is now directly implied by the facts.

In the final post-proof, the goal follows directly from the combination of background knowledge and obtained API results.

The diagram below summarizes the role of pre- and post-proofs in this example execution. Note in particular how the decreasing number of API calls in the proof indicates the goal is approaching. Also, the correspondence of a successful post-proof to the subsequent pre-proof is highlighted. This illustrates the pragmatic role of proofs in hypermedia-driven execution.



Performance of composition and selection

We still need to tackle the most pragmatic of all questions on the Web: *does it scale?* Is proof-based composition generation a feasible strategy given an ever increasing number of Web APIs? The success of our approach depends on whether state-of-the art N3 reasoners are able to generate proofs within a reasonable amount of time. In practice, the composition time should be negligible compared to the execution time of the API calls.

As of mid-2013, the widely known Web API directory ProgrammableWeb listed more than 10,000 APIs [4].

To verify this, we have created a benchmark suite [18] that generates test descriptions in such a way they can be composed into graphs of a chosen length. Web API calls can depend on any number of others. By varying the length and the number of dependencies between calls, we can investigate the influence on performance. Below are results obtained by the EYE reasoner on a 2.66 GHZ quad-core CPU.

number of APIS	**4**	**8**	**16**	**32**	**64**	**128**	**256**	**512**	**1,024**
1 dependency									
parsing	53 ms	54 ms	55 ms	58 ms	64 ms	78 ms	104 ms	161 ms	266 ms
reasoning	4 ms	5 ms	7 ms	11 ms	20 ms	43 ms	77 ms	157 ms	391 ms
total	57 ms	58 ms	62 ms	70 ms	84 ms	121 ms	181 ms	318 ms	657 ms
2 dependencies									
parsing	53 ms	59 ms	56 ms	60 ms	67 ms	85 ms	117 ms	184 ms	331 ms
reasoning	6 ms	69 ms	41 ms	45 ms	56 ms	84 ms	174 ms	461 ms	1,466 ms
total	59 ms	128 ms	97 ms	104 ms	123 ms	169 ms	292 ms	645 ms	1,797 ms
3 dependencies									
parsing	53 ms	68 ms	56 ms	61 ms	70 ms	90 ms	129 ms	208 ms	371 ms
reasoning	12 ms	45 ms	49 ms	61 ms	99 ms	200 ms	544 ms	1,639 ms	6,493 ms
total	66 ms	114 ms	105 ms	122 ms	169 ms	290 ms	673 ms	1,847 ms	6,864 ms

The total time has been split into the time used for the actual reasoning and proof generation on the one hand, and the startup and parsing time on the other hand, since parsing results can be cached. Note how a chain of 1,024 API calls with a single dependency takes less than a second to compose; the execution time of each of those calls could typically take up to a few hundred milliseconds already. More dependencies take longer, but are still manageable. Furthermore, the number of dependencies will be small in practice.

Parsing time can virtually be eliminated by preloading Web API descriptions.

The other important aspect is selection time: how fast can a reasoner find relevant descriptions? Therefore, we tested compositions of length 32 with a variable number of *dummy* descriptions that looked similar to others, but were not relevant to the composition. Note how the reasoning time remains low, even with a high number of dummies.

number of dummies	**1,024**	**2,048**	**4,096**	**8,192**	**16,384**	**32,768**	**65,536**	**131,072**
32 APIS, 1 dependency								
parsing	276 ms	528 ms	1,001 ms	1,949 ms	3,916 ms	7,827 ms	17,127 ms	34,526 ms
reasoning	12 ms	20 ms	18 ms	68 ms	107 ms	113 ms	122 ms	228 ms
total	289 ms	548 ms	1,019 ms	2,018 ms	4,023 ms	7,940 ms	17,249 ms	34,754 ms

Other composition approaches

The bulk of the work on composition on the Web has been performed in the context of classical Web services [13, 16]. The relatively recent interest in REST APIs, especially in industrial contexts, makes composition of REST APIs a yet underexplored topic—certainly with regard to hypermedia-driven characteristics. Nonetheless, several researchers have contributed to this domain, so we will summarize their published work below.

In the decade following the year 2000, Web applications started evolving rapidly, and so did Web services and later Web APIs. Soon, the idea came to combine those different services in small applications called *mashups*, giving rise to a new generation of demand-driven applications [3]. A key question in this area is how to integrate different services with a minimum of case-specific programming.

Some approaches focus on the integration of REST APIs in existing tools and workflows. Pautasso extends the Business Process Execution Language (BPEL), normally targeted at traditional Web services, to REST APIs [15]. He explains how the composition for REST APIs is different because of the late binding to URI addresses and the use of a uniform rather than a specific interface. The extensions proposed in the paper enable manual BPEL composition methods to work in resource-oriented environments. In other work, he demonstrates the integration with the visual composition language JOpera, and outlines the important features a REST composition language should support: *dynamic late binding*, *the uniform interface*, *dynamic typing*, *content-type negotiation*, and *state inspection* [14]. An alternative model is provided by Bite [17]. Bear in mind that the composition creation still happens manually with these approaches; their contribution resides on the interface and data flow level.

Alarcón, Wilde, and Bellido acknowledge the significant mismatch between action-centric composition methods and REST, and propose a novel method based on Petri nets [2] with APIs described in the Resource Linking Language [1]. The hypermedia constraint forms a fundamental part of the method, as it focuses on hypermedia controls and their semantics. The downside of the approach is that the composition is static.

The difference with our method is that we strongly lean toward the agent vision of the Semantic Web, combined with the hypermedia-driven viewpoint of REST. Our aim is to have an autonomous agent that consumes Web APIs to satisfy a given goal. Pre- and post-proofs enable a flexible way to adaptively respond in an interaction.

When trying to reach a complex goal, agents need to plan beyond the initial next steps offered in hypermedia-driven interactions. Proofs can combine RESTdesc descriptions into a composition, designed to meet the demands of a predefined goal. By distinguishing between pre- and post-proofs, the assumption of successful execution can be made explicit, while still obtaining a correct proof in the classical sense. In contrast to most composition approaches, the composition plan serves only as a guidance—the interaction itself remains fully driven by hypermedia and can be verified at each step. Similar to how the resource-orientation of REST APIs allowed us to derive concise descriptions because of their correspondence to the RDF model, N3 proofs seamlessly accommodate dynamic REST interactions.

References

[1] Rosa Alarcón and Erik Wilde. Linking data from RESTful services. *Proceedings of the 4th Workshop on Linked Data on the Web*, April 2010.

[2] Rosa Alarcón, Erik Wilde, and Jesus Bellido. Hypermedia-driven RESTful service composition. In: *Service-Oriented Computing*. Volume 6568 of Lecture Notes in Computer Science, pages 111–120. Springer, 2011.

[3] Djamal Benslimane, Schahram Dustdar, and Amit Sheth. Services mashups: the new generation of Web applications. *Internet Computing*, 12(5):13–15, 2008.

[4] David Berlind. ProgrammableWeb's directory hits 10,000 APIs. And counting. ProgrammableWeb blog, 23 September 2013. *http://blog.programmableweb.com/2013/09/23/programmablewebs-directory-hits-10000-apis-and-counting/*

[5] Tim Berners-Lee. cwm, 2000–2009. *http://www.w3.org/2000/10/swap/doc/cwm.html*

[6] Tim Berners-Lee. Semantic Web Application Platform, 2005. *http://www.w3.org/2000/10/swap/*

[7] Tim Berners-Lee. What the Semantic Web can represent, December 1998. *http://www.w3.org/DesignIssues/RDFnot.html*

[8] Tim Berners-Lee, Dan Connolly, Lalana Kagal, Yosi Scharf, and Jim Hendler. N3Logic: A logical framework for the World Wide Web. *Theory and Practice of Logic Programming*, 8(3):249–269, May 2008.

[9] Tim Berners-Lee, James Hendler, and Ora Lassila. The Semantic Web. *Scientific American*, 284(5):34–43, May 2001.

[10] Jos De Roo. Euler Yet another proof Engine, 1999–2013. *http://eulersharp.sourceforge.net/*

[11] Paul Groth and Luc Moreau. PROV-overview. Working Group Note. World Wide Web Consortium, 30 April 2013. *http://www.w3.org/TR/prov-overview/*

[12] James Hendler. Agents and the Semantic Web. *Intelligent Systems*, 16(2): 30–37, 2001.

[13] N. Milanovic and M. Malek. Current solutions for Web service composition. *Internet Computing*, 8(6):51–59, 2004.

[14] Cesare Pautasso. Composing RESTful services with JOpera. In: Alexandre Bergel and Johan Fabry, editors, *Software Composition*. Volume 5634 of Lecture Notes in Computer Science, pages 142–159. Springer, 2009.

[15] Cesare Pautasso. RESTful Web service composition with BPEL for REST. *Data & Knowledge Engineering*, 68(9):851–866, 2009.

[16] Jinghai Rao and Xiaomeng Su. A survey of automated Web service composition methods. In: Jorge Cardoso and Amit Sheth, editors, *Semantic Web Services and Web Process Composition*. Volume 3387 of Lecture Notes in Computer Science, pages 43–54. Springer, 2005.

[17] Florian Rosenberg, Francisco Curbera, Matthew J. Duftler, and Rania Khalaf. Composing RESTful services and collaborative workflows: A lightweight approach. *Internet Computing*, 12(5):24–31, 2008.

[18] Ruben Verborgh. RESTdesc composition benchmark, 2012. *https://github.com/RubenVerborgh/RESTdesc-Composition-Benchmark*

[19] Ruben Verborgh, Dörthe Arndt, Sofie Van Hoecke, Sam Coppens, Jos De Roo, Thomas Steiner, Erik Mannens, and Rik Van de Walle. The pragmatic proof: hypermedia-driven Web API composition and execution, 2014. *Submitted for publication.*

Chapter 6

Affordance

A room within a room
A door behind a door
Touch, where do you lead?
I need something more

— *Daft Punk, Touch (2013)*

The Web has made information actionable. For centuries, books and essays have been referring to each other; hypertext has turned those references into links that actually lead us to the other place. In order to scale globally, the Web had to limit the flexibility of links: they point in one direction and can only be created by the publisher of information. Our actions on a webpage remain constrained to those determined by its publisher. As ad-hoc interactions between different online applications become crucial, a linking model that allows a user-centered set of actions seems more appropriate.

The world around us is filled with *affordances*, properties of objects that allow us to perform actions. For instance, a door handle *affords* opening a door, hence we call it the *affordance* for opening that door. Similarly, a pen is the affordance that allows us to write any note. However, that same pen can afford stirring a cup of coffee and, with some skill, even opening a bottle. Originally coined by psychologist James Gibson [19], the term gained popularity among technologists through Donald Norman's book *The Design of Everyday Things* [29]. Norman wondered why people struggle with everyday appliances, and blamed common frustrations on the lack of properly designed affordances. Especially with the increasing amount of electronic devices that provide tactile capabilities, our intuition of what happens

Norman's definition states *"the term affordance refers to the perceived and actual properties of the thing, primarily those fundamental properties that determine just how the thing could possibly be used"* [29].

when we touch something (real or virtual) is challenged on a daily basis. This rapid evolution becomes all the more apparent when we find ourselves surpassed by young kids who use technology with a seemingly native ability, faster than we ever will.

Smartphones and tablets have made information even more tangible than it already was.

The virtue of hypertext is that it has transformed information into an affordance. Texts are no longer static and inert, but they can be *clicked*—on tactile screens even *touched*—to bring the reader to the next destination. A piece of information is no longer a wall but a door, actionable through a handle. Fielding emphasizes this [18]:

> *When I say hypertext, I mean the simultaneous presentation of information and controls such that the information becomes the affordance through which the user (or automaton) obtains choices and selects actions.* — *Roy Thomas Fielding*

When we introduced this definition in Chapter 2, we skimmed over the word "affordance". Together with the hypermedia constraint, we can rephrase the crucial corresponding part as *"the information must afford the next steps the client wants to take."* However, with the Web's implementation of hypertext, links can only be added to a piece of information by its publisher. Hence, the information merely affords the actions envisioned by the publisher, which don't necessarily coincide with those needed by the client. Therefore, on the Web, the information is only the affordance to the extent a publisher can actually predict the controls a client needs. While this might be the case within the closed context of a single application, it is virtually impossible on the open Web.

Recall that the Web's decision for unidirectionality allows global scalability: the Web works because links can break.

For instance, suppose you're reading a movie review on a webpage. Typically, this page will offer links to pages of cast and crew members and perhaps related movies. Yet, you want to watch the movie in a nearby theater—and someone else might want to buy it for a mobile device or stream it to a digital television. Since none of these actions are afforded by the page, the hypermedia interaction breaks. You will have to resort to another way, such as manually navigating to a search engine and trying to find it from there. While slightly inconvenient, this phenomenon happens frequently and we're used to deal with it. However, it touches on the essence of hypermedia: if we can't perform the action we need, the page becomes as non-interactive as any regular text or book. Furthermore, we have no way to repair it, as we cannot create links. Certainly on mobile devices, the omission of needed navigation controls can seriously disturb the interaction, as manual text entry on small devices takes considerable time.

In particular, it is impossible to afford actions that aren't possible today, but will be in the future. A few years ago, we couldn't yet add links to download the tablet version of a movie, even though it was clear that tablets could become popular one day.

The situation is substantially worse for machine clients that want to engage in a hypermedia-driven interaction. As the previous example shows, an *affordance* isn't an *enabler*: the availability of an action is independent of a control to execute it. However, if the action is not supported through hypermedia, a hypermedia-driven agent cannot perform it, even though it might be possible. Since machine clients lack the flexible coping strategies of humans, they cannot complete the interaction through alternative means. For that reason, agents are currently preprogrammed to perform tasks spanning different applications, leading to tight *conversational coupling* [31].

Whether REST or RPC are loosely or tightly coupled leads to intense debate; a precise definition of the different facets clears up the discussion [31].

This brings us to the inconvenient conclusion that the application of the hypermedia constraint on the Web's publisher-driven linking model is problematic: the sole party responsible for generating the affordance toward next steps is unable to do this optimally for any specific client. We have called this the Web's **affordance paradox** [41]. Similarly, while the REST architectural style decreases conversational coupling when compared to RPC-style interactions [31], it introduces *affordance coupling* [39]. The fact that a client should be able to complete *any* interaction through hypermedia puts a heavy constraint on the server, which cannot be fulfilled on the open Web. Clearly, we must either abandon the hypermedia constraint and the desirable architectural properties it induces, or find a way around its apparent contradiction with the Web's implementation of hypermedia controls.

Affordance coupling is an excellent example of the often overlooked trade-off. REST's architectural benefits indeed come at a cost.

The problem arises partly because "hypermedia as the engine of application state" implicitly assumes that this application state belongs to a single server. Given the relevant controls and semantic descriptions, autonomous machine clients can indeed use a single application in a hypermedia-driven way, as demonstrated before. Yet on the current Web, it has become impossible to confine application state to the boundaries of a single application. Instead, we should envision application state on a Web scale, where the affordance provided by a piece of information is *distributed* across different Web applications. This then transforms hypermedia affordance into a subjective experience, not imposed by the publisher, but created around the client.

As a matter of fact, the *Web* is the application.

In this chapter, we introduce our solution to the affordance paradox. First, we provide an overview of related approaches and their shortcomings. Next, the concept of our approach is detailed, followed by its architecture and two implementations. The proposed framework is then evaluated through a user study. We conclude with a discussion of its advantages and drawbacks and explain how semantic technologies and hypermedia work together.

Toward more flexible links

Affordance mismatches and the involved actors

Before inspecting methods to augment the affordance of hypermedia representations, we should understand why clients sometimes cannot complete actions. We identify three distinct possible causes [39]:

In contrast to Gibson [19], who considers *all* action possibilities, even those (seemingly) inaccessible for a subject, Norman is focusing on the *perceived* affordance [29].

1. **The affordance is present but unused.**
   Such a mismatch occurs when a person cannot find a link or when a machine doesn't understand its semantics.
2. **The affordance realizes the action with a different provider.**
   A client might have a certain action in mind that is afforded by the representation, yet not in the preferred way.
3. **The affordance is not present.**
   In this case, the action cannot be completed through hypermedia at all, so the user must fall back to other mechanisms.

All of the above three causes involve the following actor groups:

For example, a publisher offers a photograph that the client wants to crop with the online image application *ImageApp*, one of the many providers.

- The **publisher** offers a representation of a resource and, in the Web's linking model, its associated affordance.
- The **client** consumes this representation and depends on the affordance therein to perform subsequent actions.
- The **provider** is one of possibly many that offer an action desired by the user; this actor can be the publisher itself or a third party.

The first of the three causes is the result of the client's capabilities, which the publisher can accommodate for with various strategies, such as usability improvements for humans or semantic descriptions for machines. The second and third causes concern objectively missing affordances and therefore highlight those cases that require dedicated solutions.

The current Web closely couples the three actors because of its unidirectional linking model.

For the second cause, one option is to allow an interactive choice of the action *provider*. However, such solutions fall short for the third cause, as their implicit assumption is that, regardless of the provider, a publisher can foresee all possible *actions* a user might want to perform. Therefore, the second cause is actually a corner case of the third, especially if we consider the client's desired action the combination of an intention and a *specific* provider.

Compromises and trade-offs might of course prove necessary. Our goal is to maximize the affordance with minimal coupling.

Consequently, we should especially keep the third cause in mind when looking at possible solutions. For complete flexibility, resources should be able to afford any action with any provider, regardless of the specific application scenario the publisher had in mind when designing the interaction.

Adaptive hypermedia

Before the Web was invented, fundamental hypertext research was flourishing [11], yet the rise of a global hypertext system made much of it obsolete. At least one discipline survived the Web's revolution: **adaptive hypermedia**, the research field of methods and techniques for adapting hypertext and hypermedia documents to users and their context [7, 9]. Adaptive hypermedia originated in the context of closed hypermedia systems, in which the document set is under central control and hence modifiable according to an individual's properties. This is referred to as *closed-corpus adaptation*, in contrast to adaptation on open corpora such as the Web. Broadly speaking, we differentiate between **adaptive presentation**, modifying the *content* to the user's characteristics, and **adaptive navigation support**, changing the hypermedia controls inside documents. Solutions to the affordance paradox clearly belong to the latter group of techniques.

The invention of the Web heralded the end of core hypermedia research, to the extent that the Web is considered *the* hypertext system.

Adaptive navigation support systems can be subdivided into five categories: *direct guidance*, *link ordering*, *link hiding*, *link annotation*, and *link generation* [8]. The last category consists of three kinds of approaches: *discovery of new links*, *similarity-based links*, and *dynamic recommendations*. Our envisioned solution falls into the third group, but differs from existing solutions in the following aspects. Whereas adaptation techniques focus on linking related static documents together, we want to provide controls that afford actions on the current resource. Furthermore, adaptation methods are normally characterized by a specific kind of knowledge representation [9]. Instead, we strive to decouple the information needed for adaptation from specific representation formats in order to enable flexible reuse. But most importantly, a generation strategy that aims to solve the affordance paradox needs to be open-ended on both sides of the generated controls. This means that any resource should be able to afford any possible action, thereby allowing adaptive link generation on open corpora such as the Web.

A simple link annotation method commonly seen on the Web is coloring already visited hyperlinks to visually signal where a user has been before [8].

Open-corpus adaptive hypermedia has been identified as an important challenge [7], and Semantic Web technologies are considered a possible solution to help overcome the problem of adaptation on an open corpus [10]. In particular, ontologies and reasoning were deemed important [16], because of the initial interest in connecting static documents. Examples of ontology-based systems are COHSE [44], which has a static database for linking, and SemWeB [33]. Both of them can only generate links to related documents.

Leading adaptive hypermedia researchers identified adaptive Web-based systems as an important future direction [12].

Structure-based linking

OpenURL was created at Ghent University in the late 1990s and has now been adopted globally.

Identification and retrieval used to be a hard problem before the invention of the Web. URLs have solved this by coupling identification and location, which enables the Web's straightforward mechanism of hyperlinking. However, there are cases when we deliberately want to separate the two aspects. Bibliographical information is a prominent example, as many institutions have their own article library. When you click a link inside an information source towards an article, you want to consult the copy bought by your institution, not an external version that might require payment. The OpenURL standard [35] started as an initiative to provide dynamic and open links to bibliographical items [37]. Even though its broadest implementation pertains to bibliographical items, it evolved into a generic solution to provide various services on a specific piece of content [36]. OpenURL bears a strong resemblance to the concepts introduced in this chapter, the main difference being the technology stack and hence the possibilities for extension. Using semantic technologies, functionality-based matching and composition of services becomes possible.

"Documents containing collections of inbound and third-party links are called link databases, or linkbases." [15]

The drawbacks of the Web's choice for a simple linking model have been studied before: links are *static*, *directional*, *single-source*, and *single-destination* [25]. As these shortcomings could not be solved by modifying the original documents, the idea came to describe the relations between resources in separate documents called **linkbases**. The XML Linking Language (XLink) was created for this purpose [15]. It separates the concept of *association* from *traversal* by providing a structure to indicate the relatedness of several resources, and another to detail arcs from resources to others. A client can then augment a representation with additional links by consulting such a linkbase. To identify what exactly should be linked, the XML Pointer Language (XPointer) allows to indicate specific fragments in XML-based documents such as certain elements or words [14]. However, the concept has two inherent issues. First, the use of XPointer restricts the representations to XML documents, and in general, XPointer is highly dependent on a specific representation. As such, if the structure of a representation changes, the method breaks. Second, the linkbase concept implies that there is a party who is knowledgeable of the resources involved in the relation (and also of their representations). Hence, if it wants to connect resources from two applications, it needs to know both of them, so dynamic action generation on the open Web remains impossible.

The concept of relating resources has in a sense been reflected in RDF.

External interactions through widgets

Since around 2000, the Web started evolving toward an interactive medium in which visitors contribute to the content of websites. Particularly the advent of social networks, which encourage users to exchange various snippets of content with friends and acquaintances, have turned regular users into independent content creators. Part of the experience is to share and comment on content from elsewhere on the Web. To facilitate these activities, social networks offer *widgets*, such as Facebook's *Like* button [17] or Twitter's *Tweet* button [34], which form a very prominent form of external affordances on today's Web. We consider them external because they are commonly included in HTML representations as `script` or `iframe` tags with a source URL that leads to an external domain, classifying them as embedded link hypermedia factors [2]. Some of those widgets demonstrate personalized affordance; for instance, Facebook can personalize its button with pictures of the user's friends with links to their profiles. However, the decision as to what widgets should be included must still be taken by the information publisher, so the affordance remains publisher-driven. An additional issue is that different applications demand different metadata for optimal widget integration, which can make adding widgets costly [40].

An abundance of *Like* and *Tweet* buttons follows us around the Web. They are in fact affordances created by third parties, yet the publisher of information still has to decide on their inclusion on a page.

©Facebook / Twitter

In order to avoid the choice between different widgets and to vastly simplify their integration, services such as AddThis [1] offer personalized widgets to different social networking sites. Publishers only have to include one external script to provide access to many different interaction providers. Visitors who have an AddThis account may indicate their preferred sharing applications, which are then shown on visited pages that include the AddThis code. While solving the issue of interfacing to several providers, the offered actions still remain limited to what AddThis supports. Furthermore, the service doesn't exploit specific content characteristics, as all offered actions are very generic and mostly restricted to social network activities.

An undesired side-effect in the case of social network widgets is that users' privacy can be compromised. When share buttons are clicked, the social networking site of course has evidence of what content a user interacts with, which can be used for targeted advertising. Even more concerning is that users are already tracked by merely visiting a website with a social widget if they are logged in to their account [32], precisely because the widget script comes from an external source. Personalized affordance should not imply the exposure of one's personal preferences to third parties.

The discussion surrounding social networks and privacy is frequently featured in the media. An all too obtrusive integration of many social widgets in websites raises questions on their desirability.

Web Intents

The Web Intents proposal originated from Google. In response, Mozilla has coined *Web Activities* [26], specifying the delegation of actions, regardless of discovery or protocol.

A technology that allows specific actions to be embedded in websites is *Web Intents* [5, 24], which aim to offer a Web version of the *Intents* system found on Android mobile devices. There, Intents are defined as *"messages that allow Android components to request functionality from other components"* [3]. With Web Intents, Web applications can declaratively specify their *intention* to offer a certain action, and websites can indicate they afford this action. For example, social media sites can state they enable the action "share", and a photo website can offer their users to share pictures. When users initiate the "share" action on the website, the Web Intents protocol then allows them to share the photo through their preferred supporting application. In contrast to AddThis, more content-specific actions become possible, such as editing, viewing, subscribing, and saving.

Web Intents address the choice of a provider, but not users' preference for a certain action.

Although Web Intents' goals are similar to ours, there's a crucial difference in their architecture that severely limits their applicability. The benefit of Web Intents is that they are scalable in the number of *action providers*—without Web Intents, publishers have to decide which action providers they support. For instance, the publisher of the photo website would have to decide which specific sharing applications it would offer its users. With Web Intents, users can share photos through their preferred application, without the publisher having to offer a link to it. A major drawback of Web Intents is that they do *not* scale in the number of *actions*. Although the OpenIntents initiative allows to define custom actions [30], a publisher still has to decide which actions to include. In the photo website example, the publisher might opt to include a "share" action, but that is not useful if users want to order a poster print of a picture, download it to their tablet, or edit it in their favorite image application. Due to the design of Web Intents, there is no way to infer other possible actions on the current resource based on the publisher's selection.

While this strategy works on a platform such as Android, where the set of possible actions is limited to those offered by the device, such a closed-world assumption cannot hold on a Web scale. Summarizing, we can say that Web Intents do not solve the core issue: a publisher still has to determine what affordances a user might need. The problem thus shifts from deciding which action providers to support to deciding which actions to support. Therefore, Web Intents only offer personalized affordance to a limited extent—they don't offer a full solution to the affordance paradox, as the actions selected by publishers might not be those needed or expected by users.

Despite enthusiasm from its users and developers, Web Intents support has been removed from the Chrome browser.

Distributed affordance

Concept

Our solution to the affordance paradox is inspired by the typical user behavior when desired affordance is missing in the hypermedia representation. For example, suppose a user wants to edit a photo on a website through a specific online application. Unaware of the user's intentions, the publisher didn't supply a hypermedia control for this. Lacking an actual control, the user completes the interaction in an alternative way. One coping strategy would be to copy the image's URL, using the browser's address bar to navigate to the application, and paste the URL into a designated control there. This common scenario is possible because the user on the one hand knows the application supports photo editing, and on the other hand recognizes the current object as a photograph.

While users can construct actions manually, simply clicking through takes far less effort and is how the Web is supposed to work.

The above example illustrates that a lack of *affordance* to execute the action does not imply a lack of *information*. It does mean that the affordance for this action does not reside in the representation itself, but must rather be crafted manually by combining non-actionable information in that representation and out-of-band knowledge about the action provider. To automate this process, the representation should be machine-interpretable, and the provider's action should be described in a machine-interpretable way. Based on a match between a resource's content and the descriptions of actions, affordances to those actions can be generated.

Distributed affordance is the concept of automatically generating hypermedia controls to realize actions of the client's interest, based on semantic information about resources inside hypermedia representations [41]. Publishers should provide semantic annotations in representations, and action providers' services should be described semantically, so an automated client is able to infer which actions are applicable on the current resource. This allows the generation of affordances toward these actions, which are then intertwined with the representation. To account for the preferences of individual clients, the matching should happen in a personalized way.

Analogous to how human understanding of a representation allows to find actions, semantic annotations guide machines to make content actionable in a personalized way.

This method is distributed because the affordance originates from distributed sources, without requiring a central linkbase to connect documents and actions. Support for new actions and providers can be added without changing any components, as the decision whether an action matches a resource happens locally. Some form of understanding of the representation is required, but the annotations are not specific to distributed affordance.

Process

The task of a distributed affordance platform is to generate personalized hypermedia controls for the client. To this end, it needs to address the following subproblems:

- extracting non-actionable information from the representation;
- organizing knowledge about actions offered by providers;
- capturing a client's action preferences;
- combining non-actionable information and provider-specific action knowledge into possible actions;
- integrating affordance into the original representation.

All of the above should happen in a scalable way. Before the process can start, the preconditions below must be satisfied:

In case annotations are missing, they could be extracted using named-entity recognition [28].

- The representation contains some form of semantic annotations. Either the representation is structured in a machine-interpretable format such as RDF (if the client is a machine), or either it contains semantic markup (such as HTML with RDFa).
- Provider actions are described semantically in a functional Web API description format (such as RESTdesc). These descriptions can be created by the provider or by third parties.
- The client has a collection of such descriptions that correspond to preferred action providers. For instance, they could be obtained by a process similar to *bookmarking*; instead of a hyperlink to a provider's page, the action description is stored.

Automated affordance creation happens through the steps below:

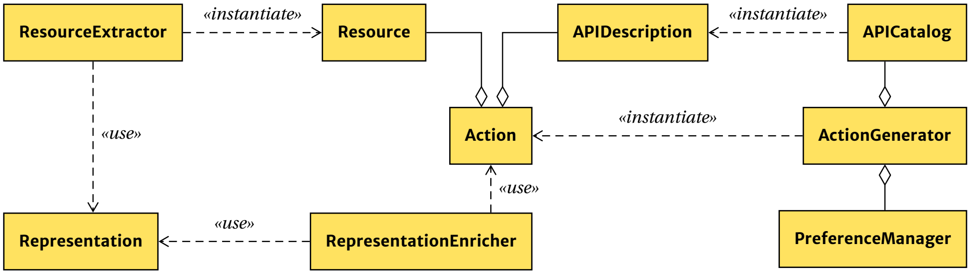
In all steps, the platform only needs access to local knowledge. This means that the affordance generation happens in a fully distributed way.

1. After the client has received the representation from the publisher, it is inspected by the distributed affordance platform.
2. The platform extracts semantic entities from the representation, using format-specific parsers (RDF, RDFa, Microdata, …) and converts them to triples. This allows to maintain the semantic information during the entire process.
3. Using Web API matching, descriptions that can act upon the extracted entities are selected.
4. Matching descriptions are instantiated with the specific entities found in the representation, thereby becoming a concrete action instead of an abstract description.
5. Controls toward the instantiated actions are created and interleaved with the representation.

After this process, the client has access to the augmented representation and can directly perform its preferred actions.

Architecture

The components of the platform's architecture can be grouped in five functional units, which are discussed below.



Information extraction A `ResourceExtractor` extracts RDF triples from a representation. `ResourceExtractor` itself is only an interface, as several annotations are possible. For textual representations, extractors could for instance use named-entity recognition techniques.

Representations can contain resource descriptions of people, movies, books, images, addresses, …

Action provider knowledge Functional Web API descriptions are maintained by one or multiple `APICatalog` implementations, each of which supports a specific method. The information in these descriptions should be structured in such a way that, given certain resource properties, it is simple to decide which APIs support actions on that resource.

User preferences A `PreferenceManager` keeps track of a user's preferences and thereby acts as a kind of filter on the `APICatalog`, typically selecting only certain APIs and sorting them according to appropriateness for the user. The role of the `PreferenceManager` can be taken care of by the `APICatalog`, which then only includes API descriptions that match the user's preferences.

A basic preference option is bookmarking; other implementations could use social recommendation.

Action generation Based on a user's preferences, `ActionGenerator` components instantiate possible actions, which are the application of a certain API on a specific set of resources. Thereby, every action is associated with one or more resources inside the representation.

Each of the action generator implementations is tied to a specific Web API description method.

Affordance integration Finally, `RepresentationEnricher` implementations add affordances for the generated possible actions to a hypermedia representation that is sent to the user. Through these affordances, clients can choose and execute desired actions directly. Implementations depend on the media type of the desired representation, as they need to augment its affordance in a specific way.

Action generation

In order to generate actions, we must match and instantiate an API description with extracted resources. Different implementations are possible; we will demonstrate the mechanism with RESTdesc descriptions. Recall from Chapter 4 that RESTdesc also offers a non-hypermedia-oriented way to describe Web APIs:

The antecedent does not contain a link (because there is none), but rather captures a resource, for which a possible action is described.

```
{ ?book dbpedia-owl:isbn ?isbn. }
=>
{
  _:request http:methodName "GET";
            http:requestURI⤶
              ("http://books.org/" ?isbn "/cover");
            http:resp [ http:body _:cover ].
  ?book dbpedia-owl:thumbnail _:cover.
}.
```

The ISBN number could be expressed in different vocabularies; ontologies can provide the mapping.

In this case, starting from a book's ISBN number, the description explains how to obtain its thumbnail image. Note how this can be *any* book resource from *any* application *anywhere* on the Web, as long as we know its ISBN number. For instance, suppose we extract the following triple from a representation:

The extraction result is independent of the original representation format.

```
<#catcher> a dbpedia:Book;
           foaf:name "The Catcher in the Rye"@en;
           dbpedia-owl:isbn "978-0316769488".
```

Then any N3 reasoner can automatically match and instantiate the Web API description above as:

```
_:request1 http:methodName "GET";
           http:requestURI⤶
     ("http://books.org/" "978-0316769488" "/cover");
           http:resp [ http:body _:cover1 ].
<#catcher> dbpedia-owl:thumbnail _:cover1.
```

To generate user-friendly links, we could add metadata to the API description, such as an action title like *"Buy this book"*.

Thus the book description affords a GET request to *http://books.org/978-0316769488/cover* to obtain the book cover. This allows the generation of a hyperlink toward the cover, which the user can activate if desired. The principle works the same for any kind of API call, such as buying the book or its e-book version, sharing it on social networks, finding reviews, … The possibilities are as endless as the number of descriptions, precisely because a machine can interpret that the current resource is a book, and that the action under consideration is possible on books.

Implementations

Because the method only needs local knowledge to generate affordance, we can choose between two implementation strategies [43]. On the one hand, we have the server-based approach, as necessarily followed by most adaptive hypermedia solutions and widgets such as AddThis. On the other hand, we can take a client-based approach like Web Intents, while maintaining full adaptation flexibility.

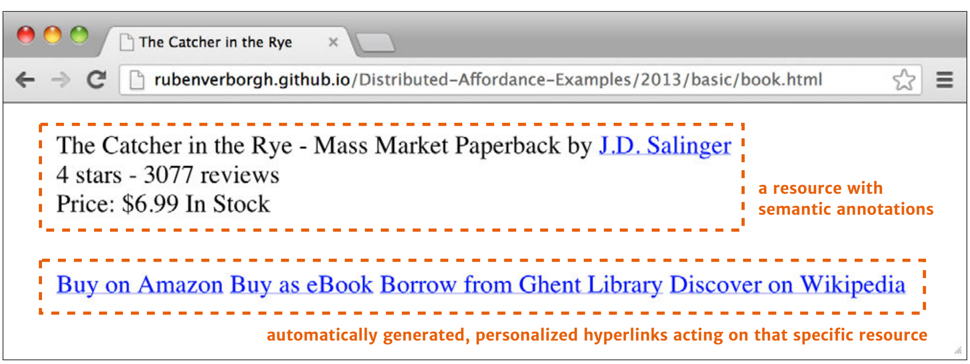
Implementations of the server-based approach can be considered **affordance as a service**. In this case, the publisher explicitly indicates that it wants to provide distributed affordance for a client. For instance, an HTML document could contain the following:

```
<div id="book" itemscope itemtype="http://schema.org/Book">
  <span itemprop="name">The Catcher in the Rye</span>
  written by <a href="/authors/salinger/" itemprop="author">J.D. Salinger</a>
</div>
<div class="affordances" data-for="book"></div>
<script src="http://shim.distributedaffordance.org/"></script>
```

Note the semantic markup with Microdata, which can serve other purposes besides generating affordance. In addition, the publisher has placed a `div` container with the marker class "`affordances`", and the "`book`" identifier that points to the information source. This container is a placeholder for generated affordance. Using a so-called *shim* script, the user's personalized affordances are generated. We have chosen for *http://distributedaffordance.org/* as a coordinating hub that can delegate to different platforms. As an example platform, we created *http://vyperlinks.org/*. The idea is that the user registers for an account with a platform of choice, which then inserts affordance to preferred actions inside the `affordances` container.

The central application *distributedaffordance.org* acts as a broker between different platform implementations from which the user can then choose. In the example scenario, only *vyperlinks.org* needs to know about the user's preferences.

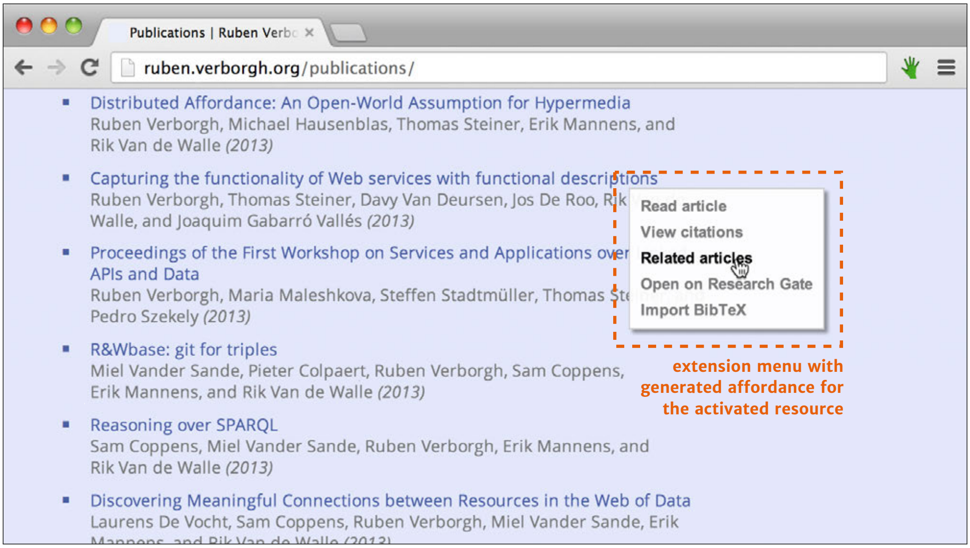
The screenshot below shows an example of affordances generated by *vyperlinks.org* on a page that contains information about a book.



The problem of affordance as a service is that the information publisher must explicitly ask for its support. The other option is to add **client-based distributed affordance** by extending the client software, for instance through a browser plugin. The benefit is that *any* page on the Web can be adapted, regardless of whether the publisher has foreseen an affordance placeholder. The drawback is that users need to install the extension to experience the generated affordances.

The shim script verifies whether the client offers affordance generation before deciding to activate the server-side version.

As it might be difficult to determine where the affordances should be placed on any given webpage, generated links can be offered in a context-sensitive way, for instance, in a popup menu or sidebar. The screenshot below shows a version of an extension for the Chrome browser. Every page that contains RDFa or Microdata markup is equipped with matching actions that can be triggered on demand.



A potential issue with this implementation is that the links are not intertwined with the content (as would be the case on webpages). When the user hovers over an item, the extension highlights the related links. Further usability testing should reveal whether sidebar-based hyperlinks are sufficient for day-to-day use.

A website can advertise it affords certain actions, which a distributed affordance platform can apply to any resource; similar to Web Intents, but without central coordination.

New actions can be added to the extension through the same affordance mechanism: a webpage or document describes a Web API, which is picked up by the extension. The last is then able to discover this API description, and can suggest to remember it for the user. For example, this could enable an online book store to offer the "buy this book" action. If users like purchasing through this store, they can add that action to their preferences for direct future use.

User study

While the properties of the platform have been analyzed during the architectural discussion, and the feasibility is demonstrated by the implementations, we still need to validate whether the generated links positively influence people's browsing behavior. We have conducted a user study to investigate the usage of links, assuming situations wherein people have a certain need that matches a previously created user profile. When designing the study, we needed to choose between a quantitative or a qualitative approach. At first, we were inclined to set up a quantitative experiment to obtain statistical data on users' efficiency increase. However, attempts to measure the time spent performing a task in early experiment trials revealed the timing variance for individuals on different tasks was far too high for generalizable conclusions. Instead, we focused on qualitative parameters in order to learn from people's experiences by performing an in-depth experiment with a smaller group.

This study was conducted together with the team of Peter Mechant at MICT.

Participants would spend remarkably more time on tasks they seemed to like, regardless of whether the platform was activated.

Setup

Following Degler [13], who evaluated methods for improving Semantic Web interaction design, we performed usability tests and interviews with sixteen users in their home or professional setting. The aim of the study was to evaluate the suitability of the distributed affordance platform for "ordinary" Internet users, and to explore how users experience and apply the affordances of the platform.

The exploratory study was designed as a *repeated-measures two-factorial quasi-experiment* with two levels for each factor, meaning participants were involved in every condition or factor of the research [20]. The first factor was the platform itself, where participants completed simple tasks *with* or *without* the platform enabled. The second factor was *briefed* or *non-briefed*, where the tasks were presented to Internet users who were briefed on distributed affordance and to Internet users who were not. In order to collect consistent data from each participant, we programmed a proxy server in such a way that for each scenario, the platform was activated in 2 out of 4 tasks the participants had to complete.

The study employed a multimethod approach [27] combining three research and analysis methodologies: *observation*, *survey*, and *interview*. These methods are complementary, yet offer different forms of data. We were particularly interested in the use and usability of distributed affordance, in observing which of the navigation options subjects used, and in gathering information about overall perceived usefulness and enjoyment of the platform.

The complete interview setup and questions are detailed in a complementary appendix [42].

Participants

Sixteen Web users participated and were subjected to the quasi-experiment in their home or professional setting. All participants were volunteers and received a gift voucher for taking part in the research. We briefly screened the participants beforehand to ensure that users with Web skills varying from low to high were included. All participants were observed while completing four simple tasks online. Afterwards, they were interviewed and asked to complete an online survey.

We were curious for the difference between low- and high-skilled users, as we assumed that the latter group would have better coping strategies when the affordance is missing.

The participants' mean age was 35.8 years ($\sigma = 15.2$), and 56% was female. On average, the participants have been using the Internet for more than 10 years. Among them, 13 owned a desktop computer and 14 a laptop, while 12 owned a smartphone and 7 a tablet. Chrome was the preferred browser of 9 people, followed by Internet Explorer (4 people) and Firefox (3 people).

Material

For each participant, we randomly selected two out of four tasks for which the platform would provide enrichment; for the remaining two tasks, we deactivated the platform. We used a proxy server to implement distributed affordance hyperlinks into the chosen websites, as not all of the websites provided the semantic annotations necessary for the regular platform. As these hyperlinks were embedded unobtrusively in the layout of the website, clicking the suggested links was intuitive, but not enforced. Furthermore, the participants had no explicit means of noticing whether the platform was active or not. They were allowed to use their browser of choice in order to replicate their usual browsing habits [38].

We would actively listen whether the participants noted the presence of the generated links (although they could not recognize them as such).

We asked participants to complete the following tasks in a varying order on a portable computer:

- ***book*** **task** – starting from a book review site, buy a book of choice;
- ***restaurant*** **task** – starting from a restaurant review site, find directions to a restaurant of choice;
- ***cinema*** **task** – starting from a cinema website, find the age of an actress in a specific movie;
- ***sharing*** **task** – starting from a cartoonist website, share a cartoon of choice on Facebook or Twitter.

These tasks were chosen to reflect common activities on the Web that many of the participants could relate to. For the sharing tasks, social media profiles were created as to not oblige participants to have and use a personal account.

Methodology

In the first phase of the study, we conducted semi-structured, in-depth interviews to gain insights in the browsing behavior of the participants. In addition to questions on media ownership, knowledge of Internet browsers, and Internet use, we implemented questions derived from media literacy research in order to assess the participants' Web skills in detail [13, 21].

The second phase of the study consisted of the participants—half of them briefed on the distributed affordance platform—executing the four tasks described above on a laptop in their home or professional environment, while a researcher observed. During the completion of these tasks, we used the Think Aloud Protocol [23], which involves participants explicitly stating their thoughts as they are performing the described tasks. Participants were encouraged to say whatever they are looking at, thinking, doing, and feeling as they go about their tasks. More specifically, by applying the Think Aloud Protocol, we tried to gain spontaneous user feedback on the platform.

The Think Aloud Protocol was particularly helpful to understand participants' reasoning regarding why certain links were clicked.

In a third and final phase, a short debriefing interview was held to gauge the participants' requirements, expectations, experiences, perceived advantages and disadvantages of the platform. Non-briefed participants were informed first on the distributed affordance concept. All participants were asked if they could distinguish the specific tasks for which the platform was activated. Next, we confronted them with the presence of the platform in each task, asking them what they would have done if the link was not suggested. To conclude the interview, the platform was evaluated using a short survey that implements the System Usability Scale [4, 6] as well as the Mean Opinion Score approach [22].

Results and discussion

Almost all participants, briefed or not, followed the links suggested by the platform as those enabled them to achieve and complete the tasks faster and more efficiently. Most participants expressed the feeling that Web links should be adjusted to their individual needs and were satisfied to find these direct links present on the websites in the distributed affordance-enabled tasks. When the platform was not activated for a given task, various participants spontaneously indicated or complained about the lack of direct hyperlinks.

While observing the participants executing the tasks, we noticed differences in self-efficacy and self-confidence between participants with high and low Web skills. However, these differences were not reflected in the interviews or answers to the survey and neither in participants' appreciation of the platform.

If the platform was not activated, almost all participants used Google when performing the restaurant, cinema, and book task. When performing the sharing task, some participants copied and pasted the image URL; others downloaded the image to the computer and subsequently uploaded it to the Facebook/Twitter profile page. Especially for users without prior social network experience, the direct link was a determining factor for success and therefore improved the Web browsing experience. According to the participants, the main added value of the platform is that it eliminates unnecessary steps in the act of browsing. This perceived advantage was especially stressed in the context of smartphones and mobile devices. Mark, a 29 year old software analyst, told us:

A participant exclaimed she couldn't complete the sharing task because she never used Twitter, only to then find the direct link and still succeed.

> *Of course, I clearly prefer distributed affordance, because it eliminates a number of extra steps […]. I always want to find things fast and it becomes very annoying if you need to take a lot of steps to reach your goal. On a fixed device, you have lots of screen space and input options, but on a smartphone, your screen is a lot smaller and the keyboard is a lot clumsier.*

It sometimes happened that distributed affordance links were present for the completion of the task, but they were not followed—the participant felt not triggered to click the link because he or she didn't notice it. In these cases, the platform misses its target. After all, the hyperlink affordances of the platform entail a relationship between an object on the Web and the intentions, perceptions, and capabilities of a person—and affordances point to both the environment and the observer [19]. In this context, participants mentioned that generated links could be embedded on designated spaces in the website layout or that they could be emphasized with special formatting to act as a reference point for the user.

Participants indicated that they did not perceive or experience the suggested links as annoying or cumbersome (in contrast to Web advertising links). However, some other concerns were voiced. One concern raised by almost half of the participants was privacy, and this was related to the private or personal information users need to disclose in order to experience the personalized character of the platform. Another concern raised during the interviews was on potential constraints the platform can impose. Concretely, because the platform eliminates the different steps that need to be executed

The demand for privacy can be met by the client-side implementation, as all preferences would be stored locally.

toward the completion of the tasks in a regular browsing context, the potential of accidentally finding new or unrelated information during this searching process is lost when using distributed affordance. Also, four respondents expressed concerns that, for the purchase of consumer products, the platform could be exploited by commercial organizations. In the words of Jenna, a 25 year old city official:

If a user's profile contains several providers, direct links to *different* shopping sites can actually appear; a "lowest price" service can even be one of them. Therefore, buying items at the best price would be a possible feature.

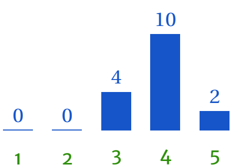
> *No, I don't think I would use this system to shop or buy products, shoes, or books for example… I would rather prefer to first have a look at various shopping sites, to compare prices and user comments. […] With this system, I would feel limited and I might pay too much for my books.*

A minority of three participants stated that the platform might prove to be "too much" and might give rise to an information overload for the Internet user. In this context, Piet, a 59 year old civil servant, told us the following:

> *Sometimes I do not need or want additional hyperlinks on the webpage because I know the solution or the appropriate link myself. In those cases, the hyperlinks provided by the platform become ballast.*

Although executing the task through the generated links might still be faster, finding the right link among many others might become difficult. Therefore, it is crucial the links are highly personalized and specific. An important future research task should thus be to investigate how preferences can be determined and applied to concrete situations.

Despite these concerns, the overall evaluation and the experiences and perceived advantages of the sixteen participants were quite positive and pointed to the platform as a functional system that is perceived as an enrichment for the Internet user (especially for those who want to browse faster or more efficiently). This is also reflected in the survey results: almost all of the participants rated their experience with the platform as good, as evidenced by a Mean Opinion Score of 3.875 ($\sigma = 0.62$) on a scale of 1 (poor) to 5 (excellent). The platform's score on the System Usability Scale, a scale for assessing system usability ranging from 0 to 100, was very high to excellent with an average of 84. This allows us to conclude that distributed affordance has the intended effect on users of the platform. Furthermore, the gained feedback will guide future developments.



The Mean Opinion Score's distribution reveals that participants appreciate the platform's functions.

Advantages and drawbacks

As a final step, we will provide an overview of the advantages and drawbacks of the proposed distributed affordance platform. Regarding the functional aspect, the platform offers the benefit of being able to combine any resource with any possible action. This contrasts with traditional adaptive hypermedia methods, which usually consider "consulting a related document" as the only action. Similarly, social widgets and related interfaces focus on variants of the "share" action, which applies to any resource type. Web Intents goes further by supporting an action set that allows extension; however, only actions explicitly indicated by the information publisher can be activated on any given resource. Because the matching for distributed affordance is based on the semantic interpretation of resources and actions, new action types can be supported directly.

The strongest feature of distributed affordance is its focus on open corpus adaptive navigation generation, which is realized through semantics.

On the architectural level, distributed affordance has the advantage that it does not require an omniscient server, as is the case with most open-corpus adaptive hypermedia methods, which generally use proxy servers. Since distribute affordance can run locally, it scales with the number of clients, without putting extra strain on any server. It shares this benefit with widgets and Web Intents.

The major drawback of the platform is its dependency on the same feature that gives it its power: semantic technologies. In all fairness, the potential benefits of semantic annotations have not sufficiently convinced Web publishers yet. Therefore, relying on the presence of these annotations can be troublesome. In that regard, a crucial decision for distributed affordance has been to rely on existing markup techniques instead of inventing a proprietary mechanism. We trust that the other features brought by annotations, such as better searchability and interaction, will provide the necessary incentives to provide some form of markup in the future [40].

Before entity extraction methods can replace semantic annotations, their accuracy must improve.

The need for semantic Web API descriptions is probably the most pressing: as discussed in Chapter 4, many approaches exist—and we introduced another one, striving to make something simple that is sufficiently expressive for automated agents. However, in general, semantic descriptions of Web API functionality are virtually non-existent. Could the incentive of potentially being used by any customer on any site be sufficient? And if so, what description format should be chosen? Pragmatic and lightweight approaches should be the best candidates: sufficiently straightforward to allow rapid integration, while still providing the means for dynamic discovery in various contexts. In the meantime, automated techniques for capturing the semantics of Web APIs could be explored.

An equivalent of named-entity extraction, targeting actions instead, could prove a viable direction.

The Web provides affordance on an unforeseen scale: any document can link to any other, regardless of where in the world the latter is located. Yet, the Web's publisher-driven linking model increasingly falls short as the need grows to act on resources through different Web applications in ways that the information publisher could not foresee. The proposed distributed affordance platform offers a linking strategy based on machine interpretation of a resource and its match to possible actions. The automatic generation of personalized, relevant affordances on the open corpus of the Web thereby becomes possible, but it depends on the availability—or extraction—of semantic annotations.

References

[1] AddThis. Share buttons, *https://www.addthis.com/get/sharing*

[2] Mike Amundsen. Hypermedia types. In: Erik Wilde and Cesare Pautasso, editors, *REST: From Research to Practice*, pages 93–116. Springer, 2011.

[3] Android. Intents and intent filters, *http://developer.android.com/guide/components/intents-filters.html*

[4] Aaron Bangor, Philip T. Kortum, and James T. Miller. An empirical evaluation of the system usability scale. *International Journal of Human-Computer Interaction*, 24(6):574–594, July 2008.

[5] Greg Billock, James Hawkins, and Paul Kinlan. Web Intents. Working Group Note. World Wide Web Consortium, 23 May 2013. *http://www.w3.org/TR/web-intents/*

[6] John Brooke. SUS: a quick and dirty usability scale. In: *Usability evaluation in industry*. Taylor and Francis, London, UK, 1996.

[7] Peter Brusilovsky. Adaptive hypermedia. *User Modeling and User-Adapted Interaction*, 11(1–2):87–110, 2001.

[8] Peter Brusilovsky. Adaptive navigation support. In: Peter Brusilovsky, Alfred Kobsa, and Wolfgang Nejdl, editors, *The Adaptive Web*, pages 263–290. Springer-Verlag, 2007.

[9] Peter Brusilovsky. Methods and techniques of adaptive hypermedia. *User Modeling and User-Adapted Interaction*, 6(2–3):87–129, 1996.

[10] Peter Brusilovsky and Nicola Henze. Open corpus adaptive educational hypermedia. In: Peter Brusilovsky, Alfred Kobsa, and Wolfgang Nejdl, editors, *The Adaptive Web*, pages 671–696. Springer, 2007.

[11] Jeff Conklin. Hypertext: an introduction and survey. *Computer*, 20(9): 17–41, September 1987.

[12] Paul De Bra, Lora Aroyo, and Vadim Chepegin. The next big thing: adaptive Web-based systems. *Journal of Digital Information*, 5(1), 2004.

[13] Duane Degler. Design 10:5:2 for semantic applications, 2011. *http://www.designforsemanticweb.com/*

[14] Steve DeRose, Eve Maler, and Ron Daniel Jr. XML pointer language (XPointer) version 1.0. Candidate Recommendation. World Wide Web Consortium, 11 September 2001. *http://www.w3.org/TR/2001/CR-xptr-20010911/*

[15] Steve DeRose, Eve Maler, and David Orchard. XML linking language (XLink) version 1.0. Recommendation. World Wide Web Consortium, 27 June 2001. *http://www.w3.org/TR/xlink/*

[16] Peter Dolog and Wolfgang Nejdl. Semantic Web technologies for the adaptive Web. In: Peter Brusilovsky, Alfred Kobsa, and Wolfgang Nejdl, editors, *The Adaptive Web*, pages 697–719. Springer, 2007.

[17] Facebook. Like button, *https://developers.facebook.com/docs/reference/plugins/like/*

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

[19] James J. Gibson. The ecological approach to visual perception. Houghton Mifflin, 1979.

[20] Frederick J. Gravetter and Larry B. Wallnau. Statistics for the behavioral sciences. Cengage Learning, 2009.

[21] Eszter Hargittai. Survey measures of Web-oriented digital literacy. *Social Science Computer Review*, 23(3):371–379, August 2005.

[22] ITU-R. Method for the subjective assessment of intermediate quality level of coding systems. Recommendation BS.1534-1. 2003. *http://www.itu.int/rec/R-REC-BS.1534-1-200301-I/e*

[23] Riitta Jääskeläinen. Think-aloud protocol. In: *Handbook of Translation Studies*. Volume 1 of pages 371–373. John Benjamins Publishing, Amsterdam, The Netherlands, 2010.

[24] Paul Kinlan. Web Intents, 2010–2013. *http://webintents.org/*

[25] David Lowe and Erik Wilde. Improving Web linking using XLink. *Proceedings of the Open Publish conference*, July 2001.

[26] Mozilla. Web Activities, *https://wiki.mozilla.org/WebAPI/WebActivities*

[27] Dhiraj Murthy. Digital ethnography: an examination of the use of new technologies for social research. *Sociology*, 42(5):837–855, 2008.

[28] David Nadeau and Satoshi Sekine. A survey of named entity recognition and classification. *Lingvisticæ Investigationes*, 30(1):3–26, 2007.

[29] Donald A. Norman. The Design of Everyday Things. Doubleday, New York, 1988.

[30] OpenIntents, *http://www.openintents.org/*

[31] Cesare Pautasso and Erik Wilde. Why is the Web loosely coupled? – A multi-faceted metric for service design. *Proceedings of the* 18^{th} *international conference on World Wide Web*, pages 911–920, ACM, New York, 2009.

[32] Riva Richmond. As "Like" buttons spread, so do Facebook's tentacles. New York Times Bits blog, 27 September 2011. *http://bits.blogs.nytimes.com/2011/09/27/as-like-buttons-spread-so-do-facebooks-tentacles/*

[33] Melike Şah, Wendy Hall, and David C. De Roure. Designing a personalized Semantic Web browser. In: Wolfgang Nejdl, Judy Kay, Pearl Pu, and Eelco Herder, editors, *Adaptive Hypermedia and Adaptive Web-Based Systems*. Volume 5149 of Lecture Notes in Computer Science, pages 333–336. Springer, 2008.

[34] Twitter. Tweet button, *https://dev.twitter.com/docs/tweet-button*

[35] The OpenURL framework for context-sensitive services. NISO standard ANSI/NISO Z39.88-2004. NISO. *http://www.niso.org/standards/z39-88-2004*

[36] Herbert Van de Sompel and Oren Beit-Arie. Generalizing the OpenURL framework beyond references to scholarly works. *D-Lib Magazine*, 7(7), July 2001.

[37] Herbert Van de Sompel and Patrick Hochstenbach. Reference linking in a hybrid library environment – part 2: SFX, a generic linking solution. *D-Lib Magazine*, 5(4), April 1999.

[38] Alexander J. A. M. van Deursen and Jan A. G. M. van Dijk. Using the Internet: skill related problems in users' online behavior. *Interacting with Computers*, 21(5–6):393–402, December 2009.

[39] Ruben Verborgh, Michael Hausenblas, Thomas Steiner, Erik Mannens, and Rik Van de Walle. Distributed affordance: an open-world assumption for hypermedia. *Proceedings of the 4th International Workshop on RESTful Design*. WWW 2013 Companion, pages 1399–1406, May 2013.

[40] Ruben Verborgh, Erik Mannens, and Rik Van de Walle. The rise of the Web for Agents. *Proceedings of the First International Conference on Building and Exploring Web Based Environments*, pages 69–74, 2013.

[41] Ruben Verborgh, Karel Vandenbroucke, Peter Mechant, Erik Mannens, and Rik Van de Walle. Addressing the Web's affordance paradox with Linked Data and reasoning, 2014. *Submitted for publication.*

[42] Ruben Verborgh, Karel Vandenbroucke, Peter Mechant, Erik Mannens, and Rik Van de Walle. Addressing the Web's affordance paradox with Linked Data and reasoning – user study, 2014. *http://distributedaffordance.org/publications/user-study.pdf*

[43] Ruben Verborgh, Mathias Verhoeven, Erik Mannens, and Rik Van de Walle. Semantic technologies as enabler for distributed adaptive hyperlink generation. *Late-Breaking Results, Project Papers and Workshop Proceedings of the 21st Conference on User Modeling, Adaptation and Personalization*, June 2013.

[44] Yeliz Yesilada, Sean Bechhofer, and Bernard Horan. Dynamic linking of Web resources: customisation and personalisation. In: Manolis Wallace, Marios C. Angelides, and Phivos Mylonas, editors, *Advances in Semantic Media Adaptation and Personalization*. Volume 93 of Studies in Computational Intelligence, pages 1–24. Springer, 2008.

Chapter 7

Serendipity

Tri martolod yaouank i vonet da veajiñ
E vonet da veajiñ, gê!
Gant 'n avel bet kaset beteg an Douar Nevez

— *Alan Stivell, Tri Martolod (1972)*

The Web's linking of information has alluring effects on the curious: starting from a page, we can click through to anywhere in the world. Many people can recall occurrences of online serendipity—when they coincidentally found something interesting when looking for something completely different. Yet, not everything on the Web works this way. Especially for machine clients, who are currently bound to rigid interaction patterns, using the Web in a flexible and dynamic way remains difficult. This chapter therefore explores the possibilities to achieve serendipitous applications.

Notoriously one of the hardest words to translate, "serendipity" stems from the ancient story *"The Three Princes of Serendip"*. The princes get involved in spectacular adventures, go in the direction of one goal only to arrive at another, but ultimately, everything always ends well. Serendipity seems to imply chance and luck—in particular, it appears mutually exclusive to planning or deliberate design. From that viewpoint, the following guidance [11] might come as a surprise:

Serendipity: the faculty or phenomenon of finding valuable or agreeable things not sought for.

©Merriam-Webster

> *Engineer for serendipity.* — *Roy Thomas Fielding*

Calling serendipity an engineerable property implies some systems are inherently more fit for it than others [30]. To a certain extent, the

Web itself was engineered for serendipity. The fact that links can point from one resource to any other, regardless of the server the latter is located on, accounts for many fortunate encounters that wouldn't be possible in systems with centralized linkbases. Yet at the same time, publisher-driven hyperlink creation deprives us from following connections that would have been created by third parties, which could bring new viewpoints to existing content [7].

In the previous chapter, we used distributed affordance to bring back interactions by providing controls toward preferred actions. Whether we call this serendipity depends on the interpretation of the word: is it coincidentally finding the things you want, or rather discovering those things you didn't know you wanted? During the user study we performed, one participant was concerned that the platform, through its attempt to add desired links, would actually remove the need to look around and thus reduce the occasions for discovering new things. This indicates that the way toward a goal might be as important as the goal itself. Fortunately, there are various ways to support both interpretations of serendipity. For instance, actions could be suggested using recommender systems, based on other users of the system with a similar profile.

So far, we only looked at implementations of distributed affordance with one single user's choices. Preference exchange in social graphs can lead to truly serendipitous links.

In this chapter, we will focus on bringing serendipity to Web applications and machine clients of those applications. As discussed before, many clients today are preprogrammed for a specific task, making it impossible for them to engage in spontaneous interactions. They can only perform the specific task they were designed for, and as a result, we end up with many applications for many tasks, as opposed to the single Web browser that allows us to manually solve *any* task. The genericness of REST's uniform interface lets different clients interact with different applications. Still, we almost always encounter single-purpose clients in practice—none of which can be reused in similar but slightly different situations.

The *Hydra* console shows a rare example of a fully generic, non-HTML hypermedia client. It is non-autonomous, providing a user interface over any REST API, given certain annotations [20].

Serendipity can be supported in hypermedia-driven cases if the client indeed tries to discover the possibilities of each representation sent by the server. This implies a degree of freedom in the representation's format, while still allowing to get a structured message across. Hypermedia can be the engine of application state to a certain extent, but could hypermedia also be sufficient as the engine of serendipity? This chapter will outline a strategic mindset we should adopt if we want to design applications that can collaborate in more flexible ways than currently possible. We start by advocating the combination of hypermedia and semantic technologies, followed by a discussion with examples of what serendipitous Web applications might look like.

Although it might seem paradoxical, *planning* for serendipity can lead to flexible reuse [30].

Semantic hypermedia

Semantic media types

Contracts are vital in distributed systems, as they determine the structure of interactions between different parties. In REST systems, the contract is partially fixed by the **uniform interface**, which stipulates resources as the unit of information, together with the rules on how resource manipulation should happen [30]. The other part is defined by the used **media types**, which detail the formats, processing model, and hypermedia controls [31]. In fact, REST API design should focus on defining media types and/or extending existing ones [12].

Well-designed contracts allow for an independent evolution of clients and servers.

As outlined in Chapter 3, there is an inherent trade-off between specificity and reusability: more specific media types carry more detailed semantics, at the cost of being less portable across situations. Therefore, the recommended strategy is to choose the media type with the least expressivity that still fulfills the task at hand. In increasing order of expressivity, we have generic hypermedia types, customizable patterns, and domain-specific solutions [25].

Another issue with generic hypermedia types is that API publishers often treat them as domain-specific types, but label them otherwise. For instance, the publisher of a certain API might label all of its responses as `application/json`, the standard JSON media type, even though they follow a structure with far stronger constraints than JSON. While technically correct—and helpful to a parser—this designation does not tell anything about the document's interpretation. Instead, it is highly likely that this interpretation will be communicated in an out-of-band way, so that clients need to know beforehand how to employ the information in a response. For machine clients, this necessitates a preprogrammed interpretation.

In nearly all use cases, `application/json` would be too vague: many APIS offer resources that need more accurate typing.

However, it is a fallacy that media types eliminate out-of-band information. For instance, that same API could choose instead to return responses in an `application/vnd.myformat+json` media type, which would provide an interpretation specific to the application. While this resolves the situation in which a client receives a resource in a media type it can parse but not interpret, it doesn't change the fact that the client must be preprogrammed for this interpretation. After all, media types are usually described in human-readable form. We thus arrive at the paradoxical situation that specific media types are created precisely to eliminate out-of-band information during the interaction, yet the interpretation of the media type itself remains out-of-band. This seriously hinders autonomous agents, which can parse those representations, but not grasp their semantics.

Suffixes such as `+json` can indicate the more generic media type to which representations conform [14]. The `vnd` prefix indicates a vendor-specific type [13].

One of the four constraints of the REST uniform interface is the use of self-descriptive messages [10]. As we explained in Chapter 2, this reflects in HTTP's limited method set and standardized metadata fields. We could also consider standardized media types as part of this, as their interpretation is widespread. Domain-specific media types can hardly be called self-descriptive because of the required out-of-band information. However, if we *embed* the interpretation into the representation of a resource, then the self-descriptiveness constraint becomes fulfilled. With human-targeted media types such as HTML, our understanding of natural language makes representations self-descriptive. For machine clients, we can rely on semantic media types such as RDF variants, which allow for automated interpretation.

Media types and their corresponding identifier can be registered at the Internet Assigned Numbers Authority (*www.iana.org*).

We define **semantic hypermedia** as the subclass of REST APIS that send and accept machine-interpretable representations using semantic media types (possibly in addition to others). Assuming a client that understands the generic base type (such as Turtle, RDFa, or RDF/XML) and a server that applies the Linked Data principles [2], the response can be interpreted without relying on out-of-band knowledge. However, we should be careful with the significance of this statement. While "interpretation" of course doesn't mean that autonomous agents suddenly obtain capabilities comparable to those of humans, it does lead to the following:

XML provides some level of structural extensibility through namespaces.

- Representations can describe resources at any desired level of detail. In contrast to structure-based formats such as JSON, where consumers expect specific keys and values organized in a rather strict way, RDF is entirely resource-centric and triples can detail any (sub-)resource as desired. Clients and servers can simply ignore triples irrelevant to their current task.
- Servers can allow their clients a flexible choice of vocabulary, as reasoning enables inferring certain properties from others. For instance, clients could indicate a resource's label with `rdfs:label`, `dc:title`, `foaf:name` or others; the server can infer equivalence. To facilitate interpretation, both parties could express facts in several vocabularies, as unneeded triples can be ignored anyway.
- Agents that receive instructions in a semantic way, like in the process detailed in Chapter 4, can relate a server's response to the query without needing application domain knowledge. For instance, if the query requests a `dbpedia-owl:Image` with certain properties, the agent can verify whether the server's response meets the criteria, without requiring a built-in notion of images.

Semantic media types can help realize Postel's law: *"be conservative in what you send, be liberal in what you accept"*.

"Interpretation" is again based on the matching to known things, the core idea behind Linked Data.

Semantic hypermedia thus enables a higher **autonomy** of clients.

We will contrast the approaches through an example. Suppose an API offers entity lookup: given properties about a topic, it tries to find a unique identifier. We could create a specific media type for this, based on JSON, that we name `application/vnd.rv.entities+json`. An example query document could be represented as:

The differences between various non-semantic and semantic media types are independent of a specific representation design.

```
{ "entities": [
    { "name": "Pete Townshend", "type": "person" },
    { "name": "Terry Riley",    "type": "person" } ] }
```

The server could then represent a response as:

```
{ "entities": [
    { "name": "Pete Townshend", "id": "dbpedia:Pete_Townshend" },
    { "name": "Terry Riley",    "id": "freebase:07qf7" } ] }
```

To understand these fragments, clients need to know the meaning of `entities`, `name`, `type`, and `id`, as well as their structure. Furthermore, this knowledge is not transferable to other media types, which might even have different interpretations for those fields.

Compare this to a possible RDF representation of the query:

```
_:p1 a foaf:Person; rdfs:label "Pete Townshend".
_:p2 a foaf:Person; dc:title "Terry Riley";
     schema:birthDate "1935-06-24"^^xsd:date.
```

Note how the knowledge needed for interpretation is independent of this specific media type: `rdfs:label` and `schema:birthDate` have a universal meaning. Furthermore, if the meaning would be unknown, a client can look it up through its property URL and relate it to known concepts. Note also how *different* properties indicate labels, and how an extra property `birthDate` was supplied to allow disambiguation. Maybe the server doesn't support it at the moment, but when it does, the property will be recognized. The server could respond with:

To add new properties in structure-based formats, the field name would have to be agreed on first.

```
dbpedia:Pete_Townshend rdfs:label "Pete Townshend".
freebase:07qf7 rdfs:label "Terry Riley";
               dc:title   "Terry Riley".
```

Again, this can be interpreted by any client that can parse Turtle. Note that the server can specify the label in multiple vocabularies.

This illustrates how semantic formats make representations self-descriptive, removing the need for specific media types. Conveniently, the transition to semantic hypermedia does not have to be disruptive: content-negotiation allows the client to request either a JSON- or an RDF-based representation of the resource.

The JSON-LD media type provides evolvable JSON representations by giving them RDF semantics [22].

The remaining question is how clients can construct representations in absence of the rigid structure imposed by a specific media type. While RESTdesc explains the functionality of an API by relating its preconditions to its postconditions, it purposely does not detail the representation of the exchanged messages. This allows clients to engage in content negotiation at runtime and to deal with non-textual content such as images and videos. In the previous example, RESTdesc could explain that properties of entities lead to identifiers of those entities, but it would not detail the format of either message. While a client can interpret a server's response automatically because of the embedded semantics, the RESTdesc description doesn't detail how the entities should be sent to the server. In this example, the RDF format is so simple it could be "guessed": it simply describes the available entity properties. In the general case, more possibilities exist and we need to understand the server's preferences without a specific media type.

Non-hypermedia formats can still allow hypermedia-driven navigation through `Link` headers in the HTTP response [24].

One solution is to explicitly describe the expected request and response triple patterns [19]. These techniques often relate to *lifting* and *lowering*, the transformation between non-semantic and semantic representations [18]. Unfortunately, pattern descriptions restrict the possibilities not enough on the one hand, such as when only certain value ranges are allowed, and too much on the other hand, since they block the flexibility that RDF brings. On the positive side, they can be considered a machine-interpretable equivalent of media type definitions, which the client can discover at runtime.

However, we believe that a hypermedia strategy is the appropriate solution here. Similar to how human-targeted representation formats offer *forms* to structure input (such as HTML's `<form>` element), hypermedia representations for machine clients should provide the controls that afford the desired actions. The *RDF forms* initiative [1] was a first attempt to achieve this in RDF, yet further developments are necessary [16]. The Hydra vocabulary [21] seems promising in this regard. An alternative approach is to semantically annotate HTML input fields, so machine clients can understand their purpose. Such techniques for hypermedia forms make the interaction fully happen in-band, similar to the mechanism for links. As a result, they integrate seamlessly into the hypermedia-driven process of Chapter 4.

Making an API machine-friendly means adjusting its affordance accordingly.

As a final remark, we shouldn't forget that error responses also require machine interpretation. Recently, a generic method to detail the cause of HTTP error responses in JSON was proposed [23]. Again, using a semantic media type for this would allow clients to interpret errors without any prior understanding. An interpretation of an error's cause could help in finding an alternative strategy.

An error response should also obey a client's media type preferences through content negotiation, so the client can act upon it.

Discovering semantic resources

For agents to become truly autonomous, they should not only know how to browse APIS, but also how to find them. On a distributed system such as the Web, efficient discovery relies on *indexes*. In the beginning days of the Web, a manual list of servers was maintained, which gradually became obsolete through the advent of keyword-based search engines [3]. Much of our daily online activities involve search engines: to find starting points for a task and, if the affordance toward the next desired step is missing, to find that step as well.

Centralized indexes seemingly contradict the nature of distributed systems, but they remain the quickest method. Future advances in distributed search techniques might change this.

Machine clients currently have only limited access to search. One could think this is not necessary because Linked Data leads to related resources, but the unidirectionality of Web linking prevents many interesting lookups. For instance, photos of a certain person are often annotated with an identifier of that person, but it's highly unlikely that the description of a person will link to all her photos. Hence, if an agent needs to find all those photos, an identifier of the person will not directly yield the needed information. We need the equivalent of a search engine, but with a focus on machine clients. *Sindice* is an index of machine-interpretable data on the Web that crawls semantic formats such as RDF and semantic annotations in HTML documents [26]. It allows finding documents about concepts using their URI or property values through a Web API or a SPARQL endpoint. However, *ranking*, the key feature of search engines to display the most relevant results first, is currently difficult with triples. Consequently, finding the relevant information to solve a certain task often involves trial and error.

Business plans for indexes targeting machine clients require special thought, as machines cannot generate revenue through watching advertisements.

Furthermore, Sindice only indexes static content. To search for Web APIS that offer a certain functionality, we need more advanced discovery mechanisms. Many solutions for service discovery have been developed [17, 27], but none of them were deemed the definitive answer. Given the performance of RESTdesc matching, we believe that a repository with RESTdesc descriptions could give fast replies to queries for a certain functionality. However, considering more complex matching operations that take vocabulary differences into account can require significantly more server resources. Perhaps functionality could be discovered in an indirect way by providing URIS of related concepts, similar to keyword-based search. An agent could then retrieve semantically related API descriptions and evaluate whether they match a task. Once a starting point has been given, the client could discover an API in a hypermedia-driven process, like the way developers browse an API's documentation. Yet, the discovery aspect of autonomous agents clearly still needs intensive research.

In many cases, full autonomy isn't required. Agents can simply receive access to a large API description collection, since selection of APIS happens fast.

Toward serendipitous Web applications

When automated clients have access to serendipitous interactions on the Web, they themselves become providers of serendipity: users can ask to achieve a certain goal and a client will do so, as if it was programmed for this specific task by chance. We define **serendipitous Web applications** as those applications that can use the Web in ways they were not explicitly designed for. Although slightly utopian today, we advance toward a Web on which machine clients can perform increasingly complex tasks. We consider *autonomous agents*, *semantics-driven applications*, and *client-side querying*.

Autonomous agents

Much of the thinking that underpins this work was inspired by the Semantic Web vision of agents [4]. Even acknowledging the fact that the authors were outlining an idea and not an actual plan, the achieved successes so far have only laid the bare foundations. Commercial personal digital assistants such as Apple's Siri seem to come closer to the envisioned agents than the current research of the scientific Semantic Web community. However, Siri isn't an agent in that sense, because it can only perform actions it has been preprogrammed for (admittedly in an intuitive and personalized way). In particular, it cannot interact with Web APIs it hasn't been designed for. We thus wouldn't call Siri a serendipitous application.

David Martin, one of the driving forces behind Siri, was also a co-author of the OWL-S specification, so some of Siri's roots lie in the Semantic Web.

What is it then that Semantic Web agents can do, and what does the technology introduced in the previous chapters add to that? The main goal is autonomy: having an agent perform a task without (or with minimal) assistance. As we've outlined, this includes discovery of data and functionality, an interpretation thereof, composition of a plan, the execution of its steps, and reporting back to the user. Thanks to the Web, agents can rely on knowledge and services from many different providers; the challenge is to do this intelligently. Hypermedia-driven execution is an important part of the solution, so agents don't need to know the steps of any interaction beforehand. Instead, they can follow the controls provided by servers to advance the application state. In case a representation doesn't contain the desired controls, they can be added through distributed affordance.

Autonomous agents can satisfy a goal on the Web, for instance through the hypermedia-driven process we introduced. The challenge is to make this work outside of controlled environments, as agents do depend on semantic descriptions, which are not commonly available.

As long as agents cannot fully interpret natural language, they need machine-readable representations of the content and functionality offered by Web APIs. These also allow agents to plan in advance. This semantic gap remains the most pressing issue, as it prevents users from interacting with autonomous agents in a more fluent way. The silver bullet is to allow the specification of tasks in natural language.

Semantics-driven applications

Linked Data is supposed to make the development of data mashups easier, because it can be flexibly shaped into different formats. However, applications developed with Linked Data often remain confined to the silos they were created in [8]. Although Linked Data should enable reuse in theory, few applications can readily switch to another dataset. For instance, it would be common practice to develop a new sightseeing application for every city—even though all such applications fulfill essentially the same function, only with different datasets. Those applications that do offer different cities tend to work with one centralized, non-linked dataset.

"Unlike Web 2.0 mashups, which work against a fixed set of data sources, Linked Data applications operate on top of an unbound, global data space." [5]

Where did we go wrong? An explanation can be found in the way Linked Data applications are currently developed. We notice that, despite adopting RDF's triple model, data is still treated the same way as with more rigid models. Developers make assumptions about what properties will be used, which values will be there, and how concepts can be identified. These assumptions have proven unportable across different datasets, which are structured according to slightly different design decisions. This illustrates how applications are primarily built in a **data-driven** way that highly depends on the data's structure, even though the data model possesses more flexibility. We should evolve toward a **semantics-driven** way, in which developers bind the application to the semantics rather than to the data.

In practice, there might be commercial reasons to develop multiple applications. From the software engineering viewpoint, it isn't a necessity.

We need to shift our perspective to realize such semantics-driven applications. While the current approach is to build applications on top of datasets, we should create applications to which different data streams can be connected. In other words, a specific dataset shouldn't influence the internal design of the application, but the application should shape incoming data streams instead. Concretely, a specific application must implement a certain service, and users should be allowed to choose the dataset on which the application provides that service—in a serendipitous way.

As different datasets are often expressed in different ways and varying levels of granularity, we need to put mechanisms in place to deal with this in a uniform way. By not querying the data directly but asking a reasoner to infer the desired triples, differences between ontologies can be bridged. This requires dereferencing the URIs of the used properties to supply input for the reasoner, which then relates them to properties that are known to the application. Therefore, the application only needs to be programmed against a specific set of properties, as the semantics in the dataset allow a reasoner to shape the data in the expected format.

Binding to data semantics will lead to higher development costs for a single application, yet only one application is needed for many different scenarios.

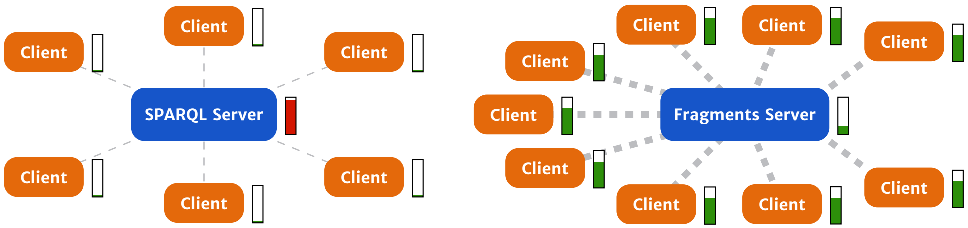
Client-side querying

The major challenge when consuming large quantities of data is to find those specific elements you are looking for. The SPARQL protocol [9] allows to execute SPARQL queries [15] on RDF data over the Web. The current principle is that a client sends a query to a server, which then executes this query over its internal RDF store. However, the *scalability* of this approach quickly becomes problematic. With public SPARQL endpoints, the server is not in control of the number of requests and their complexity. As a result, the availability of public SPARQL endpoints is notoriously problematic [6].

The load of HTTP servers is far more predictable, as the server is responsible for resource partitioning. SPARQL, in contrast, allows clients to send arbitrarily complex requests [28].

Serendipity can only happen if the client is sufficiently intelligent. I believe that, in order to develop intelligent clients, we should refrain from building intelligent servers. The SPARQL vision of having a single endpoint that will solve any query might work in closed environments, but not on a Web scale. Instead, we should offer clients the affordance to solve queries themselves. The idea of Linked Data Fragments [29] is to partition a data source into chunks of Linked Data, such that **client-side querying** becomes possible. While this necessitates more data transfer, each fragment is cacheable and reusable across multiple clients. Partitionings are designed to require only minimal server processing for a fragment. The conceptual difference is shown below.

We ran tests with complex queries, which, in medium to large numbers, quickly bring down SPARQL servers. Most of these queries could be executed client-side within a few seconds [29].



Note how, when using Linked Data Fragments, the clients perform the actual computation, whereas the server merely supplies the data. As a result, servers can handle many more clients because the complexity of each request is controlled, and the number of computing units increases linearly with the number of clients.

A concrete partitioning that minimizes server effort while still enabling powerful queries consists of fragments for all triple patterns `?s ?p ?o` of a dataset, wherein each component can be variable or fixed. In addition, the server should provide metadata such as counts, and controls such as links to other triples. A query for a basic graph pattern, consisting of many such triples, can then be solved at the client side by iteratively querying for those subpatterns with the lowest member count [29]. This enables dynamic, Web-scale querying.

Additionally, client-side query execution facilitates querying from distributed data sources. A client then simply needs to combine fragments from different servers—the same way it does for a single server.

By combining the strengths of hypermedia and semantic technologies in a pragmatic way, we can develop a new generation of Web applications that serendipitously reach goals they were not specifically programmed for. Such applications deliver the flexibility promised by Linked Data, if we are willing to take the additional effort to bind our application not to the data itself but to its semantics. In addition to functional descriptions, semantic hypermedia types can play a fundamental role in the autonomous consumption of APIs by serendipitous applications. This paves the way for autonomous agents, semantics-driven applications, and scalable client-side querying on the Web.

References

[1] Mark Baker. RDF forms, May 2003. *http://www.markbaker.ca/2003/05/RDF-Forms/*

[2] Tim Berners-Lee. Linked Data, July 2006. *http://www.w3.org/DesignIssues/LinkedData.html*

[3] Tim Berners-Lee. Weaving the Web: The original design and ultimate destiny of the World Wide Web by its inventor. HarperCollins Publishers, September 1999.

[4] Tim Berners-Lee, James Hendler, and Ora Lassila. The Semantic Web. *Scientific American*, 284(5):34–43, May 2001.

[5] Christian Bizer, Tom Heath, and Tim Berners-Lee. Linked Data – the story so far. *International Journal on Semantic Web and Information Systems*, 5(3):1–22, March 2009.

[6] Carlos Buil-Aranda, Aidan Hogan, Jürgen Umbrich, and Pierre-Yves Vandenbussche. SPARQLWeb-querying infrastructure: ready for action? *Proceedings of the 12th International Semantic Web Conference*, November 2013.

[7] Vannevar Bush. As we may think. *The Atlantic Monthly*, 176(1):101–108, July 1945.

[8] Anastasia Dimou, Pieter Colpaert, Raphaël Troncy, Erik Mannens, and Rik Van de Walle. Cataloguing open data applications using Semantic Web technologies. *EU networking session at the 10th Extended Semantic Web Conference*, June 2013.

[9] Lee Feigenbaum, Gregory Todd Williams, Kendall Grant Clark, and Elias Torres. SPARQL 1.1 protocol. Recommendation. World Wide Web Consortium, 21 March 2013. *http://www.w3.org/TR/sparql11-protocol/*

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

[11] Roy Thomas Fielding. Re: "I finally get REST. Wow." The REST architectural style list, 8 May 2007. *http://tech.groups.yahoo.com/group/rest-discuss/message/8343*

[12] Roy Thomas Fielding. REST APIs must be hypertext-driven. Untangled – Musings of Roy T. Fielding. October 2008. *http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven*

[13] Ned Freed, John Klensin, and Jon Postel. Multipurpose Internet Mail Extensions (MIME) part four: registration procedure. Request For Comments 2048. Internet Engineering Task Force, November 1996. *http://tools.ietf.org/html/rfc2048*

[14] Tony Hansen and Alexey Melnikov. Additional media type structured syntax suffixes. Request For Comments 6839. Internet Engineering Task Force, January 2013. *http://tools.ietf.org/html/rfc6839*

[15] Steve Harris and Andy Seaborne. SPARQL 1.1 query language. Recommendation. World Wide Web Consortium, 21 March 2013. *http://www.w3.org/TR/sparql11-query/*

[16] Kjetil Kjernsmo. The necessity of hypermedia RDF and an approach to achieve it. *Proceedings of the Workshop on Linked APIs for the Semantic Web*, May 2012.

[17] Matthias Klusch, Benedikt Fries, and Katia Sycara. Automated semantic Web service discovery with OWLS-MX. *Proceedings of the* 5^{th} *International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 915–922, ACM, 2006.

[18] Jacek Kopecký, Dumitru Roman, Matthew Moran, and Dieter Fensel. Semantic Web services grounding. *Proceedings of the International Conference on Internet and Web Applications and Services*, pages 127–127, February 2006.

[19] Reto Krummenacher, Barry Norton, and Adrian Marte. Towards Linked Open Services and processes. In: *Future Internet Symposium*. Volume 6369 of Lecture Notes in Computer Science, pages 68–77. Springer, 2010.

[20] Markus Lanthaler. Creating 3^{rd} generation Web APIs with Hydra. *Proceedings of the* 22^{nd} *international conference on World Wide Web – Companion*, pages 35–38, 2013.

[21] Markus Lanthaler and Christian Gütl. Hydra: a vocabulary for hypermedia-driven Web APIs . *Proceedings of the* 6^{th} *Workshop on Linked Data on the Web*, May 2013.

[22] Markus Lanthaler and Christian Gütl. On using JSON-LD to create evolvable RESTful services. *Proceedings of the Third International Workshop on RESTful Design*, pages 25–32, ACM, April 2012.

[23] Mark Nottingham. Indicating problems in HTTP APIs, 15 May 2013. *http://www.mnot.net/blog/2013/05/15/http_problem*

[24] Mark Nottingham. Web linking. Request For Comments 5988. Internet Engineering Task Force, October 2010. *http://tools.ietf.org/html/rfc5988*

[25] Leonard Richardson, Mike Amundsen, and Sam Ruby. RESTful Web APIs. O'Reilly, September 2013.

[26] Giovanni Tummarello, Renaud Delbru, and Eyal Oren. Sindice.com: weaving the Open Linked Data. *Proceedings of the 6^{th} International Semantic Web Conference*, pages 552–565, Springer, November 2007.

[27] UDDI version 2.04 API specification. Technical report. 19 July 2002. *http://uddi.org/pubs/ProgrammersAPI-V2.04-Published-20020719.htm*

[28] Ruben Verborgh. Can I SPARQL your endpoint?, 30 September 2014. *http://ruben.verborgh.org/blog/2013/09/30/can-i-sparql-your-endpoint/*

[29] Ruben Verborgh, Pieter Colpaert, Sam Coppens, Miel Vander Sande, Erik Mannens, and Rik Van de Walle. Web-scale querying through Linked Data Fragments. *Submitted for publication*, 2014.

[30] Steve Vinoski. Serendipitous reuse. *Internet Computing*, 12(1):84–87, January 2008.

[31] Jim Webber, Savas Parastatidis, and Ian Robinson. REST in Practice. O'Reilly, September 2010.

Chapter **8**

Conclusion

I thought I saw down in the street
The spirit of the century
Telling us that we're all standing on the border

— Al Stewart, On the Border (1976)

The goal of this thesis has been to investigate how machine clients can use the Web in a more intelligent way. We introduced RESTdesc to capture the functional semantics of Web APIs, followed by a proof-based Web API composition mechanism. We then used semantic descriptions as a solution to the Web's affordance paradox. Finally, we zoomed in on serendipity for today's Web applications. In this concluding chapter, we review the initial research questions and outline opportunities for future research.

Giving machine clients more autonomy essentially comes down to offering them similar affordance as we provide to people, given that both excel in different capabilities. This last fact is captured beautifully in what became known as *Moravec's paradox* [3]:

> *It is comparatively easy to make computers exhibit adult level performance on intelligence tests or playing checkers, and difficult or impossible to give them the skills of a one-year-old when it comes to perception and mobility.* — *Hans Moravec*

Nobody knows exactly how many years away machines are from natural language understanding, and what "understanding" means in that context. Until then, we will have to assist machines if we want them to assist us. We do what we are best in, and they do the same.

Review of the research questions

In Chapter 2, I introduced the three research questions that have guided this thesis. I will now review how they have been answered.

In the context of the Web's affordance paradox—the client depends on links created by a publisher who doesn't know the needs of that client—the following question arose:

How can we automatically offer human and machine clients the hypermedia controls they require to complete tasks of their choice?

While allowing global scalability, the Web's linking model restricts link creation to content publishers. Yet, hypermedia only works to the extent it actually affords the actions clients want to perform (as opposed to solely those envisioned by publishers). For human information consumers, our answer is distributed affordance: automatically generating links based on the interpretation of a resource. Through semantic technologies, a machine can decide locally which actions are possible on a piece of content. This way, hypermedia controls can be generated without an omniscient component, providing a solution to the outstanding open-corpus adaptation problem [1].

It remains a tough challenge to incentivize publishers to enhance their representations with semantic markup, which is a prerequisite for distributed affordance. Fortunately, minimal markup can be sufficient in many cases. For instance, specifying that a representation contains a book with a certain title or a person with a certain address already allows many matches. This markup can additionally serve as input for many other purposes, precisely because semantics allow for application-independent metadata. In absence of explicit annotations, text processing algorithms could identify entities in a representation, although this can require external knowledge.

For machine clients, we can apply the same mechanism. However, even though hypermedia controls indeed turn information into an affordance [2], part of the total affordance is inherent to the content itself. For instance, the fact that a certain text mentions the author of a book indirectly allows us to look up more books written by this author—even if no direct link exists. An interpretation of the content thus allows the execution of actions upon it. Therefore, the primary focus of content publishers should lie on affording an interpretation of the content to machines by providing either a semantic representation or semantic annotations to existing representations. This then allows creating distributed affordance as well as following the hypermedia-driven process introduced in Chapter 4, which is the topic of the next research question.

How can machine clients use Web APIs in a more autonomous way, with a minimum of out-of-band information?

Since REST APIs are actually interconnected resources, we interpret "Web API" here as any site or application on the Web, regardless of whether it has been labeled as such. Autonomous use of a Web API means on the one hand the interpretation of the resources themselves, and on the other hand an understanding of the possible actions that can happen on those resources. The interpretation of resources is possible through representations in semantic formats such as RDF, the usage of which we've referred to as *semantic hypermedia*.

Conform to the REST architectural constraints, HTTP implements a limited set of well-defined methods. Hence, the understanding of actions starts with the specification. The semantics of `GET`, `PUT`, and most other methods doesn't require any clarification, but the `POST` method has been loosely defined on purpose. Hence, machine-interpretable descriptions must capture the semantics of possible `POST` requests. In addition, descriptions can capture the expectations of requests that use the other methods in order to look beyond the actions offered by a single hypermedia representation.

I believe that having some kind of action plan is unavoidable; after all, people always have a plan when realizing a complex task. In contrast to most planned interactions between clients and servers, we want the client to respond dynamically to the current application state. This is why, even though a plan indicates the steps, the interaction happens through hypermedia by activating the relevant controls. "Hypermedia as the engine of application state" remains a valid interaction model, as long as we accept that the information publisher cannot foresee all possible actions, and thus adjust accordingly (for instance, through distributed affordance). Furthermore, the active use of hypermedia reduces the amount of out-of-band information to a minimum. If the API descriptions are offered and consumed in a dynamic way, the interaction happens entirely in-band.

In order to satisfy a given goal, N3 reasoners can instantiate RESTdesc descriptions into a composition. This composition is generated by their built-in proof mechanism without requiring any plugins, as RESTdesc descriptions act as regular N3 rules. The resulting proof should be interpreted as a pre-proof, which assumes the used Web APIs behave as described. After executing the composition through the hypermedia-driven process, we obtain a post-proof with the actual values obtained through the APIs. This proof-based mechanism enables agents to autonomously determine a plan and execute it in a fully hypermedia-driven way.

How can semantic hypermedia improve the serendipitous reuse of data and applications on the Web?

In Chapter 7, I reflected on the various applications that become possible when Web APIs add semantics to the representations of the resources they offer. The need for application-specific media types, which require clients to be preprogrammed, can be eliminated by semantic media types. These allow generic clients to interpret resources based on the task assigned to them. Thereby, clients can perform tasks as if they were designed for it, which turns them into serendipitous applications.

Semantic technologies are currently not expressive enough to capture the nuances of natural language and therefore not a definitive solution for all possible autonomous and serendipitous applications. Indeed, we should always keep in mind that the RDF family of technologies remains a means to an end, not a goal in itself, nor the only means to reach that end. However, semantic hypermedia can considerably simplify the development of such a novel generation of applications. Instead of aiming to directly find the definitive solution to autonomy and serendipity, we should first try to maximize the usage of the current technologies. As I have illustrated, many useful scenarios are supported with today's mechanisms. Therefore, we must support the initiatives that aim to convince data publishers and application developers to adopt the technology that is already there.

Future work

To end this thesis, I will list future work that builds upon the introduced technologies.

As far as RESTdesc is concerned, I plan to investigate its possibilities in smart environments, where devices dynamically react to a user. REST APIs can play an important role in such environments because they make a conceptual abstraction of different functionalities as resources. RESTdesc descriptions can enable the automated integration of new devices, without requiring existing components to be preprogrammed. As this could complicate the interactions, we might need to move from a pure first-order logic to modal logic, which allows different, mutually exclusive states to exist separately. N3 supports this through formulas, but their impact on RESTdesc descriptions and the composition process must be examined.

Another question is to what extent we can generate RESTdesc descriptions in an automated or semi-automated way. Since the lack of functional descriptions on the current Web remains an important obstacle for automated agents, we should facilitate description creation as much as possible. Furthermore, we need research on the organisation of such descriptions into next-generation repositories, which would allow fast and flexible querying of functionality.

For distributed affordance, I want to search optimal ways of capturing user preferences and displaying those links that are most relevant. Existing work on adaptive hypermedia, and adaptive navigation support systems in particular, could be applied to this new platform. The combination with recommender systems, especially in a social context, should provide new insights. To assess the effectiveness of various personalization methods, I want to design new user studies that focus on day-to-day use, which is a challenge because the platform offers open-corpus adaptation.

One aspect I haven't considered so far, but which will become important when striving for significant adoption, is the role of generated affordance in commercial contexts. As I argued before, information publishers are uncertain about the added value of providing semantic annotations. Unfortunately, such annotations could prove a competitive disadvantage: if machines can automatically determine the vendor with the lowest price, then most of the established marketing techniques ought to be replaced. The same holds for distributed affordance: what if the user prefers a link to a competitor?

There will always be a tension between the goals of users and the goals of the party that provides the information (and thus might expect something in return). This doesn't mean that semantics and new forms of affordances, both of which are ultimately designed to make people's lives easier, wouldn't be commercially viable. The Web has already brought us entirely different business strategies, some of which are still not well understood. Analogously, those new technologies could bring strategies of their own.

Perhaps the most significant contributions in future work can be made through the development of serendipitous applications. Agents with an increasing degree of autonomy can show the potential of Semantic Web technologies and assist people with various tasks. They can contribute to a new mindset for application development that embraces the openness of the Web, instead of trying to constrain it to the more established development models.

With this thesis, I aim to show how hypermedia and semantics can lead to an unprecedented level of pragmatic serendipity on the Web. Even though we hope automated interpretation of natural language will eventually obsolete the current technology, the Semantic Web still offers plenty of underexplored opportunities in the meantime.

References

[1] Peter Brusilovsky. Adaptive hypermedia. *User Modeling and User-Adapted Interaction*, 11(1–2):87–110, 2001.

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

[3] Hans Moravec. Mind Children: The Future of Robot and Human Intelligence. Harvard University Press, 1988.