Markus Köykkä

Intelligence as a Service: Designing Semantic APIs to Intelligent Software

Bachelor's Thesis
in Information Technology
November 29, 2011

University of Jyväskylä

Department of Mathematical Information Technology

Jyväskylä

Author: Markus Köykkä
Contact information: markus.koykka@jyu.fi
Title: Intelligence as a Service: Designing Semantic APIs to Intelligent Software
Työn nimi: Semanttisten ohjelmointirajapintojen suunnittelu tekoälykkäille palveluille
Project: Bachelor's Thesis in Information Technology
Page count: 29
Abstract: The purpose and focus of this Bachelor's Thesis is to explore, examine and (to some extent) analyze the design heuristics and guidelines of an API that is to be implemented for a software or service that facilitates AI. An essential challenge in the creating of the thesis is that there are no papers to be found on the particular subject. General API design guidelines, related W3C recommendations and basic requirements of AI are explored, while trying to form some kind of a cohesion about what a semantic API for an intelligent software would or should be like.
Suomenkielinen tiivistelmä: Tutkielman tarkoituksena on yrittää löytää periaatteita ja hyödyllisiä ohjeistuksia tai havaintoja tekoälykkäille palveluille ja ohjelmistoille tarkoitettujen semanttisten ohjelmointirajapintojen (APIen) suunnittelua ja toteuttamista varten. Aiheesta ei löydy valmiita tutkimuksia, mikä tekee asian eksaktin käsittelyn varsin haasteelliseksi. Asiaa lähestytään yleisen API-suunnittelun kautta, tuoden mukaan relevantit W3C:n suositukset sekä tekoälyn lisäämät vaatimukset. Lopuksi yritetään muodostaa näkökulma semanttisten ohjelmointirajapintojen suunnittelusta tekoälykkäille palveluille.
Keywords: API design, API usability, Semantic API, Artificial intelligence, AI, IaaS, Intelligence as a service
Avainsanat: API suunnittelu, API käytettävyys, Semanttinen API, Tekoäly, AI, IaaS, Ohjelmointirajapinta

Contents

1 Introduction

The purpose and focus of this Bachelor's Thesis is to explore, examine and (to some extent) analyze the design heuristics and guidelines of an API that is to be implemented for an IaaS-type software, and how to enable semantic APIs to automate the utilization of intelligent software as a service.

The domain of *Intelligence as a Service* (abbreviated IaaS) here should not be mistaken for *Infrastructure as a Service* (which is also abbreviated IaaS): while the former means combining Service-Oriented Architectures (SOA) with intelligent algorithms, the latter means delivering a computer infrastructure as a service (e.g. a platform virtualization environment) [1]. The focus of this thesis is entirely on the former meaning, thus from henceforth the term IaaS is used exclusively in the meaning "Intelligence as a Service".

Furthermore, Service Oriented Architecture (SOA) is defined by the Organization for the Advancement of Structured Information Standards (OASIS) reference model as "*a paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains. In general, entities (people and organizations) create capabilities to solve or support a solution for the problems they face in the course of their business.*" [2]

"*SOA is a means to an end: delivering systems that are more manageable, flexible, and agile. But the primary goal must be to support the ever-changing needs of the business. IT groups should spend less time focusing on technology and infrastructure, and instead focus on delivering systems (aka services) that deliver measurable business value.*" — Anne T. Manes, when asked to briefly summarize what it takes to make SOA successful for a business.

Therefore a real world manifestation of an IaaS-type service is a particular intelligent software that is published in the Web as a service, which can be accessed by humans or by other software applications. It can be used for acquiring relevant intelligent support (e.g. decision making, intelligent search, machine learning, pattern recognition). Network of such services could be considered as a kind of "distributed brain" to which even a simple piece of software application can connect and utilize. In the world of distributed computing, one computer agent's requirements are possibly met by a computer agent belonging to a different owner. [2]

Currently no studies on this particular topic can be found, effectively making elaborate research on the topic rather challenging. The hypothesis is the existence of general API design principles, guidelines or good practices. Therefore the exploration of the topic in this thesis is structured in a bottom to top manner: Towards

the end the particular IaaS API design characteristics are gradually approached by starting with a broad set of general API design guidelines, followed by exploring the features of the semantic API and the requirements of facilitating artificial intelligence (AI) services, and finally by attempting to combine all of the previous to form a general idea of the guidelines by which an API for an IaaS-type service could be implemented.

2 General API Design Guidelines

2.1 What is an API?

API stands for an application programming (or program) interface [3].

Generally, an API is a published interface to software-based functionality that abstracts system or program calls [4]. In other words, an API provides the access interface to the contained services for other software, analogically much in the same way as an user interface facilitates interaction between humans and computers. An API is essentially a set of function calls and all packaged or modular applications usually have their own proprietary APIs.

APIs offer productivity gains by supporting code reuse, providing high-level abstractions that facilitate programming tasks, and by helping to unify the programming experience. [5]

Many types of APIs exist: operating system APIs, application APIs, toolkit APIs and web site APIs. The size of an API can range from a single line to an enormous definition (e.g. Java Software Development Kit's or Microsoft's .NET framework's APIs). As APIs grow larger and inevitably more complex their usability may become debatable [6]. As we will see in the following chapters, these different types of APIs may still share common design values and heuristics.

2.2 API Design

By examining the available academic publications, a gradually increasing amount of interest and research towards APIs can be seen within the last decade. [7] Many of the sources used for this thesis assert that designing good, usable APIs is critical (or at least increasingly important) for any type of software development that has a programmatic user interface. [8, 9, 7]

Java expert Joshua Bloch is often quoted, cited and agreed with in API design -related studies. According to him API design is important for a number of reasons:

1) Good APIs create long-term customers, while bad ones create long-term support nightmares. 2) Customers invest heavily in the APIs in terms of buying products for it, writing for it, and spending considerable time learning it. 3) Public APIs are also hard or even impossible to change once they're published and in use, as users that have started using them will expect all of the features to be continuously available for the entire lifespan of the API. New features can be added, but it is problematic to remove the old ones. [8]

Programmers of all types make use of APIs within their various designs [9], and most programmers are, to some extent, also API designers. Good reusable programs are modular, and the intermodular boundaries are defined by APIs. [8]

While the use of APIs is ubiquitous, there is relatively little research about their design. Recently though, a number of researchers and practitioners have begun to treat API design as a first-order object of study and practice. [9]

Following is a rather long list of the most relevant practical considerations or guidelines of API design that Bloch mentions [8]:

Architectural:

- "Most APIs are overconstrained". When hesitant whether to include a feature in the API, Bloch's advice is simple: to not include it. He emphasizes the rule: "When in doubt, leave it out" as probably the most important thing to remember about API design. This is due to the fact that in API design you can always add things later, but you can't take them away. Minimizing conceptual weight is paramount. Accessibility should also be minimized to simplify the API.

- APIs should be easy to use and hard to misuse: doing simple things should be simple, doing complex things should be possible, and doing wrong things should be close to impossible.

- Early drafts should be short, typically one page with class and method signatures and one-line descriptions. API-design mistakes should be expected. Users will most likely find uses that the designers haven't thought of. To avoid this, in addition to keeping it simple (at least at the beginning), the API design should also be shown to as many people as you can, and the feedback taken seriously.

- APIs requirements should be gathered having a healthy degree of skepticism: instead of accepting solutions clad as requirements, the actual problems should be found and solved. Requirements should be structured as use-cases by

which the API can be eventually measured. Additionally, the use-cases against the API should be coded before implementing or entirely specifying the API, to prevent implementation or even specifying a fundamentally broken API. The code of the use-cases should be maintained as the API evolves to prevent "rude surprises", and to have a basis for tutorials, tests and examples. The quality of the example code should be exemplary, as the examples will be the archetypes of thousands of real world solutions. According to a study by Martin Robillard the examples can become more of a hindrance than a benefit when there's mismatch in the tacit purpose of the example and the goal of the example user. [5]

- APIs should be self-documenting: documentation should be rarely required to understand the code of the API. This can be achieved essentially by maintaining the design quality — most importantly in naming conventions. Perhaps seemingly a bit conversely he does later add that documentation does matter, and that every exported API element needs to be well documented.

- Names matter — they need to be consistent, symmetric and intelligible. He compares every API to a little language that people must learn to read and write. An API that's well designed will read like prose. If problems with naming arise, it paraphrases of a fundamental flaw in the design: it might be in order to split or merge the API, or embed it in a more general setting.

- The user should not be surprised by the action of the API: the method should do what the users expect it to do, given its name. Overloading should be avoided when the behaviour of the methods differ. His phrase "Obey the principle of least astonishment" is often quoted and worth remembering about API design. (Robillard does add that this can be hard to put in practice [5].)

- Fail fast. The common widely accepted programming principle is that when an error or bug appears it should be reported as soon as possible, preferably at compile-time.

- Performance consequences of the API design details needs to be considered, but the API should not be warped to achieve performance gains.

- Client shouldn't be made to do anything the library could do to avoid error-prone boiler plate (stereotyped or formulaic [3]) coding in the client-side.

- All different APIs must coexist peacefully with the platform. It is almost always a bad idea to "transliterate" an API from one platform to another.

Implementation Related:

- APIs should be kept free of implementation details, as they confuse users and inhibit the flexibility to evolve. Mutability should be minimized, as immutable objects are simple, thread-safe and freely shareable. (Mutability in general is "disposition to change" [3], but in here he refers to mutability as it is featured in the Java language). Although Bloch doesn't mention it: it's worth noting that the high-level design of the API is not an implementation detail.

- Parameters should be always ordered consistently and return values that demand exceptional processing should be avoided (e.g. zero-length arrays or collections should be returned rather than nulls). Long parameter lists and especially multiple parameters of the same type can cause user misunderstandings and mistakes, and should therefore be avoided.

- Fixed limits on input sizes should be avoided, as they limit usefulness and hasten obsolescence.

- Programmatic access to all data that is in string form should be made available. This is to avoid programmers from being forced to parse strings, and to avoid turning the string forms to a de facto standard for the API. That being said, the most appropriate data type should be used: strings should not be used if there is a more appropriate type.

- Exceptions should be thrown only to indicate truly exceptional conditions. Unless clients can realistically recover from the failure the exceptions should be unchecked. The API client code cannot reasonably be expected to recover from them or to handle them. [8] Runtime exceptions can occur anywhere in a program and having to add runtime exceptions in every method declaration would reduce a program's clarity. [10]

Note that even though these guidelines are void of formal definition and could thus be of subjective and contentious by nature, they are indeed widely accepted and cited in API design and usability related studies.

While many of these design key points originally relate to the Java environment, I believe that they are abstract enough to be implementation (and language) independent at least to a great degree.

Bloch also claims that "API design is an art, not a science" [8]. Although there certainly is art to science (and science to art), this, if any, is the thing to disagree with Bloch. There definitely are better and worse ways to designing, even though it may

not always be that apparent. Adhering to sensible design conventions in general enhance the capability to design and produce an API that is more easily understood and therefore more applicable by the users (and all who review or re-design it for any reason).

Additional guidelines of various level detail and importance can also be found from many seemingly unrelated sources, such as *Framework Design Guidelines: Conventions, Idioms and Patterns for Reusable .NET Libraries* by K. Cwalina and B. Adams [11]. Being less generic and more environment specific their usefulness is harder to evaluate in regard to API design.

2.3 API Usability

A good API design is essentially the way to build a well *usable* API. Based on an API resource site and "API for APIs" -type service called *ProgrammableWeb*, the number and facilitation of APIs on the web is both increasing and dynamic, and a wide array of different kinds of services is already available on the web [12]. Many of the industry's largest players are creating new, competing APIs, and the web itself is becoming the next development platform, as programmers are starting to write applications "*on Google*" and "*on Amazon*" the same way they used to write "*on Windows*" and "*on Unix*". [13]

Although usability research in programming has already had a long and productive history, the API usability and design aspect of it has not been treated as an important subject within the field until the recent years. APIs present an interface that is purposed for reuse by others, and as stated in the Conference on Human Factors in Computing Systems (CHI) held in 2009, it has become apparent over time that a theoretical base for API usability could have a great positive impact on the software development profession. [9] They also claim that much of the academic research towards APIs relates to usability of design patterns and programming paradigms, and essentially applying them to API design (unfortunately I was unable to get access to the relevant source material they referred to).

In the same manner as recognizing and condemning poor usability decisions in applies programming it also applies to API usability.

In "Building more usable APIs" [6] the following product attributes that affect API usability are listed:

- How easy the API is to learn.
- How efficiently the API can be used for specified tasks.

- How easy the API calls are to remember.
- What misconceptions or errors programmers make using the API
- How programmers perceive the API.

These attributes go well hand in hand with the design ideas that Bloch listed [8]. Both base much emphasis on how humans perceive the API. It's obvious that the areas of usability and design overlap enough to make it difficult to really separate the two.

A very relevant study by Martin P. Robillard examined obstacles and challenges that the professional developers at Microsoft faced when learning to use APIs [5]. Instead of focusing on specific design aspects, Robillard interviewed and surveyed the experienced developers[1] about the obstacles they faced while learning APIs. He discovered many issues that complement those mentioned in API design textbooks and articles, and in particular found that the API learning resources are actually as worthy of attention as the structural aspects of the API.

He mentions these three as the most important findings:

1. Developers need to be well informed of the high-level design of the API in order to be able to use the API efficiently and structure their code accordingly.
2. While examples are generally considered very useful, they can become a hindrance when the actual purposes of the designer and the examples mismatch.
3. Subtle design features that affect the behavior of the API can confuse developers.

Other interesting findings of the survey reported included that the most important ways of learning the API were (in order of method popularity percent among the respondent developers):

1. Reading API documentation (78%)
2. Code examples (55%)
3. Experimenting with the API (34%)
4. Reading articles about the API (30%)
5. Asking colleagues about the API (29%)

[1]The developers of various titles and positions all worked at Microsoft, had on average 12.9 years of experience, and their aggregated learning experiences covered 54 distinct APIs.

6. Other methods such as reading books or tracing code through a debugger had a lower frequency

About the actual obstacles in learning the API 74% of the respondents reported having at least one obstacle. The biggest two of the obstacles were related either to the examples or to the API's structural design. According to Robillard, to mitigate (all of) the obstacles, API *documentation* must "include good examples, be complete, support many complex usage scenarios, be conveniently organized and include relevant design elements." [5]

He also later reminded that the whole architecture of the API (the "high-level design") needs to be understood by the developers for them to be able to facilitate the API beyond simple usage scenarios. This kind of understanding is needed for example when choosing the most appropriate way to do things between multiple options.

About the purposes of the code examples according to the developers themselves he lists the following: "Providing best practises of an API's use; informing the design of code that uses the API; providing rationale about an API's design; and confirming developers' hypotheses about how things work."

These guidelines (all of which have been actually mentioned earlier by Bloch) appear similar to the guidelines that have been covered earlier in this thesis, thus this valuable study by Robillard appears to confirm their general validity.

According to the article "Studying the documentation of an API for Enterprise Service-Oriented Architecture" [14] that was published in *Journal of organizational and end user computing*, many issues and recommendations for improving the usability of API documentation were found in an user study of a large and complex eSOA-API. It was found that the users background influenced how they navigated the documentation — for example the lack of familiarity with business terminology was a barrier for developers without business application experience. All tested user groups avoided areas of the documentation that had an inconsistent visual design [14], which can yet again be seen as another proof of how important it is to not ignore the human factors in computer science.

2.4 About API usability testing

According to several sources, API usability tests *in the lab* are time and resource intensive, effectively allowing only a relatively small percentage of a large API's namespace to be evaluated. As APIs can indeed be very large (again Microsoft's .NET framework that has hundreds of namespaces and libraries is used as an ex-

ample), the usability testing is far from trivial and the adequacy of formal empirical evaluations is being questioned. [7]

Farooq and Zirkler of Microsoft suggest an agile-like (aka SCRUM-like) method for approaching this problem: including API Peer Reviews in API usability testing [7]. Essentially they seem to quite sensibly build on many of the basic ideas Bloch presented. [8]

Without going into too much detail, a short description of the method, that the authors claim to be effective for testing the usability of .NET framework at Microsoft, is as follows: The usability inspection is done by group-effort of these interdisciplinary actors: feature owner, feature area manager, usability engineer and reviewers. The first three roles are typically implemented by one person each, while 3–4 is the goal amount of organizational peers for the reviewers' group. The process itself goes through three quick (all performed within a few hours) steps: Planning, Review and Bug filing.

Of these, the most interesting phase is clearly the review phase, which has all the people of the group together in a discussion that is driven by two questions presented for the reviewers: 1) What do you think this code snippet (e.g. for a method) does? 2) Does the implementation in the code module make sense? Although they do not mention it, many of the Bloch's design guidelines [8] could actually be directly used within this phase by the reviewers. The authors do name the problem with method naming as an example. Ultimately actionable feedback is what the review phase discussion strives for, which will later (in the Bug filing -phase) be used to record the usability breakdowns and bugs. In conclusion, one such API peer review takes 6–7 people and only from 1.5 to 3.5 hours per person. [7] Due to these features this method looks like the one most worth mentioning for testing the usability of an API.

3 The Semantic API

3.1 The Semantic Web and W3C

The traditional Web is often compared to a book with links to other books. Similarly, the Semantic Web is often compared to a database — the data is linked, but is more comprehensible by computers.

The word *semantic* is generally defined by: ”relating to signification or meaning” [3]. The purpose of adding semantics in the computer world is to equip the computer (that is: software agents and services etc.) with the possibility of ”under-

standing" the actual meanings behind the words. In theory, machine-processable *metadata* ("data about data") enables computers to be able to distinquish meanings and relations behind the raw data, at least to some extent. Additionally, the Semantic Web is not about links between web pages, but instead describes the properties of things and their intermediate relationships. [15]

Currently there is a variety of "Semantic Web browsers" available, that can be used to actually browse the linked data. For example: DISCO, Marbles, the OpenLink Data Explorer, Tabulator, the Zitgist Data Viewer or the Fluidops Information workbench.

The vision of the Semantic Web was introduced in 2001 by T. Berners–Lee et al. [16] and since then, to actualize that vision, a great amount of study and turbulent methodological and technological evolution has occurred in both scale and diversity.

The World Wide Web Consortium (W3C) is working to standardize the Web. It was created in 2004 as a collaboration between the Massachusetts Institute of Technology (MIT) and the European Organization for Nuclear Research (CERN), with support from the U.S. Defense Advanced Research Project Agency (DARPA) and the European Commission. W3C is a member organization that includes a variety of software vendors, content providers, corporate users, telecommunications companies, academic institutions, research laboratories, standards bodies and governments. [17] Some of the most well known members include Microsoft, Apple, Nokia, Adobe and Sun Microsystems. Digital Media Institute of Tampere University of Technology hosts the W3C's Finnish office. [18]

The WWW standards that W3C creates and maintains are called *W3C recommendations*. Although a myriad of Semantic Web technologies, languages and service frameworks already exist, the focus of this thesis will lie essentially on the W3C recommendations. On Web Services the OASIS Web Services Interoperability (WS-I) Member Section needs to be mentioned as it also advances Best Practices for Web services standards across platforms, operating systems, and programming languages. [20]

According to the *W3Schools Semantic Web Tutorial* [15] the semantic web is not a very fast growing technology. One reason for this is that *RDF*, a language for describing semantic web resources, is not very easy to understand by traditional developers who lack the academic background in logic and artificial intelligence. In the same tutorial it is also noted that the semantic web will not work all by itself, it will need work to become a reality. That means that the semantic definitions and semantic relations between different concepts and all data need to be laid out by

someone (or something).

It is hard to form a consensus on how to define true semantic meaning, as there is no RDF format that everyone agrees can fully describe a document — each individual can come up with their own RDF schema, as the RDF specification allows for a RDF schema to be published, but does not specify a "one true schema". [36]

The semantic heterogeneity of component descriptions (in terms of I/O variables, operations, events) is a well known issue, and it is thus crucial to enable a standard way for their abstraction. [37]

3.2 The W3C Recommendations

The core W3C recommendations that are related to the Semantic Web are RDF, SPARQL, OWL and SKOS. In addition to these four technologies however, the Web of Services in general is based on technologies such as HTTP, XML, SOAP and WSDL. All these recommendations relate to the Semantic Web when Services in the Semantic Web are considered. [19]

These can also be used in tandem with mainstream object-oriented languages. Both RFD Schema and OWL are now supported by tools, parsers and programming APIs. [21]

Following is a short introduction to all the W3C recommendations mentioned above (except for HTML and XML of which the reader is assumed to have knowledge of):

- The Resource Description Framework (RDF) is a language for describing resources in the web. RDF/XML (the XML-based syntax that RDF provides) is machine processable and quite exchangeable between different types of operating systems and application languages. It is aimed to be the general conceptual method of description and modeling of information that is implemented in the Semantic Web, and is designed to be first and foremost read and understood by computer applications [33].

  RDF can be also used to represent information about things that can be identified on the Web with Web identifiers (URIs), even when they cannot be directly retrieved. Resources with properties and property values can be defined by RDF. [22] These properties can be defined as elements, attributes and resources, depending on the purpose and environment. The RDF namespace defines only the core framework, and is generally extended by other namespaces.

For example the RDF itself defines concepts such as container and collection definitions. A container is a resource that contains *members* — the types of basic containers are: *Bag*, *Seq*, and *Alt*; and a collection is a container that only allows the specified members. The application specific classes and properties need to be defined using extensions to RDF.

RDFa is a specification for attributes to express structured data: it provides a set of XHTML attributes to augment visual data with machine-readable hints. [23]

RDF Schema (RDFS) provides the framework to *describe* application-specific classes and properties. With RDFS the resources can be defined as instances and subclasses of classes. [33]

RDF is ideal for representing predefined metadata such as the properties of *The Dublin Core*, that defines a set of predefine properties for describing documents. RDF and its Schema constitute the base infrastructure to represent classes, properties and instances in a Web compliant format. [21]

From a perspective (by Bill Roberts at semanticweb.com) the RDF accessed via HTTP can even itself be seen as an API.

- The OWL 2 Web Ontology Language (OWL 2) is an ontology language to be used in the Semantic Web for representing knowledge about entities, groups of entities, and relations between entities. In other words, OWL 2 is a knowledge representation language, designed to formulate, exchange and reason with knowledge about a domain of interest. [25] Basically OWL extends RDF Schema with richer expressivity. [21]

  OWL 2 is a compatible extension to the 2004 version of OWL, and can also be used along with information written in RDF. OWL documents, known as ontologies, are primarily exchanged as RDF documents. These are defined to use datatypes of the XML Schema Definition Language (XSD). There are various syntaxes available for OWL 2 which serve various purposes. [24, 25]

  OWL 2 is not a programming language. Instead it is a declarative language: it describes a state of affairs in a logical way. Appropriate tools (so-called reasoners) can then be used to infer further information about that state of affairs. Neither is OWL 2 a schema language for syntax conformance or a database framework, although there are some similarities to the latter. [25]

- SPARQL can be used to express queries from RDF documents. SPARQL contains capabilities and syntax vaguely similar to SQL for querying required and

optional graph patterns along with their conjunctions and disjunctions. Extensible value testing and constraining queries by source RDF graph are also supported by SPARQL. The results of SPARQL queries have four different query forms, including sets, boolean values or RDF graphs. The result sets can be accessed by a local API but also can be serialized into either XML or an RDF graph. [26]

- The Simple Knowledge Organization System (SKOS) is an RDF vocabulary for representing semi-formal knowledge organization systems (KOSs). These machine-readable representations can be exchanged between software applications and published on the Web. The aim of SKOS is to enhance the portability and interoperability of conceptual vocabularies.

  "SKOS has been designed to provide a low-cost migration path for porting existing organization systems to the Semantic Web. SKOS also provides a lightweight, intuitive conceptual modeling language for developing and sharing new KOSs. It can be used on its own, or in combination with more-formal languages such as OWL. SKOS can also be seen as a bridging technology, providing the missing link between the rigorous logical formalism of ontology languages such as OWL and the chaotic, informal and weakly-structured world of Web-based collaboration tools, as exemplified by social tagging applications." [27]

- Simple Object Access Protocol (SOAP) is a format for sending messages and for accessing a Web Service, especially intended to be used in a distributed environment. [17] It "provides the definition of the XML-based information which can be used for exchanging structured and typed information between peers in a decentralized, distributed environment. SOAP is fundamentally a stateless, one-way message exchange paradigm, but applications can create more complex interaction patterns (e.g., request/response, request/multiple responses, etc.) by combining such one-way exchanges with features provided by an underlying protocol and/or application-specific information." [28] The current recommended version of SOAP is 1.2.

- Web Services Description (formerly: Definition) Language (WSDL) is a format for describing network services and how to access them. The concrete network protocols and message formats are combined into the services that act as abstract endpoints. The current version of WSDL is 2.0. [29]

3.3 Semantic Web Services

Although the technologies for semantic Web have not been fully developed yet, the essential related technologies such as RDF and OWL are receiving a lot of both academic and commercial interest. However, to facilitate the possibilities of the semantic web, *semantic web agents* and *semantic web services* are needed.

There are claims that the automation of Web Services has failed: ”The Web Services technology failed to meet its promises of fully automating the communication between applications. It necessitated human intervention in the selection and invocation of the most appropriate web service.” [43]

Earlier technologies, such as pre 2.0 WSDL, had the limitation of lacking syntactic interoperability — it focused on defining the structure of the messages being exchanged, but was blind to the semantics of these messages. IBM submitted a specification for WSDL-S in 2005 to add semantic expressibility to WSDL, but it did not come to be a recommendation as itself. [30, 43]

Many other related submissions were made in 2005. The Web Service Modeling Ontology (WSMO) and the Web Service Modeling Execution environment (WSMX) are two techniques supporting the Semantic Web Services technology that have been found able to automate the whole communication process including both selection and invocation of the most matching Web Service. [43] Further examples of submissions to W3C include: Semantic Web Services Framework (SWSF), Semantic Web Services Language (SWSL), The Semantic Web Services Ontology (SWSO), and OWL-S. However, none of these have become W3C recommendations.

Instead, Semantic Annotations for WSDL and XML Schema (SAWSDL) is a W3C Recommendation (since 2007) that defines how to add semantic annotations to various parts of a WSDL documents. [31]

SOAP has the same ”problem”, no semantics are attached to a SOAP request or response [43], but ”SOAP is silent on the semantics of any application-specific data it conveys, as it is on issues such as the routing of SOAP messages, reliable data transfer, firewall traversal, etc. However, SOAP provides the framework by which application-specific information may be conveyed in an extensible manner.” [28]

On the other hand there can be found plenty additional criticism that is pointed towards SOAP and the host of other web service protocols. One viewpoint to the reason of this can be that the current semantic descriptions in all their complexity may still be too limited. In the conclusions of a study on the evolution of web architecture it is stated as follows: ”The tower of web service standards and protocols stacked above it utterly fail; computational exchange requires the full semantics of

powerful programming languages: conditionals, loops, recursion, binding environments, functions, closures, and continuations, to name only a few. Without these tools, web service developers are condemned to recapitulate the evolution of programming languages." [32]

There are some attempts to list the available Semantic Web Services, one of which is the Semantic Web Service Directory (maintained by Dr. Khalid Belhajjame of the University of Manchester) that can be useful as an index for semantic web service investigators, although the list has not been updated since 2008. [42]

Another related standard that never made to be a W3C recommendation is the OASIS standard Universal Description, Discovery And Integration (UDDI), of which the newest version is currently 3.0.2. Although the idea seems sensible: to provide a platform-independent XML-based registry for listing, registering and locating web services, it has been mockingly called "A solution looking for a problem", and it has also been said that "there is no vendor left who cares about it. The standard isn't being extended, there is no working group, there are no new initiatives, and even if you are a strong believer in SOA governance and associated tools, you'll find out that most of them no longer use UDDI (except as a checkbox feature)." [35]

3.4 Semantic Web Service APIs

There are some free semantic API services available, a prominent example of which is *DBpedia.org*, which is a RDF version of Wikipedia. The DBpedia data set can be accessed online via a SPARQL query endpoint and as Linked Data. Several papers on DBpedia.org can been found, one good example of which is the presentation of DBpedia Spotlight, a tool for detecting mentions of DBpedia resources in text that "enables users to link text documents to the Linked Open Data cloud through the DBpedia interlinking hub." [34]

Various commercial semantic API services also exist, to name a few: Zemanta, Ontos Semantic Technologies, Semantic Engines LLC, Textwise LLC, ClearForest (of Thomson Reuters) all claim to offer a semantic API for their customers.

How to design a Semantic API? The short answer is to design an API that has, receives and transfers semantics. What are the components of a Semantic API? Again, the short answer is that a Semantic API has all the same components as a similar non-semantic API, but includes the semantics. However, in practice the issue is perceived to be quite a bit more complex.

One problem still often presented is how to choose the best method (recommendation or standard) for implementing the semantics. In the Semantic Web envi-

ronment the latest W3C recommendations are strong candidates, but besides those there are numerous projects outside W3C that use various methods for adding semantics to different platforms and languages: in addition to those already previously mentioned, there exists also ongoing research on how to add semantics to programming languages themselves, such as to C++ (project SemantiC++). [38]

There are also methods and technologies for design and implementation that can improve the quality and effectiveness of the design. For complex systems there's the suggested Semantic Engineering Design Environment (SEDE) that consists of Engineering Ontology, Design Rules, Design Database, and Design Data Web Service, is a suggestion for a collaborative engineering design environment based on the Semantic Web technologies that W3C recommends. [39] Without going into much detail, using this design environment the dynamic distributed design participants interface each other using design web services that are maintained in a web service list. Relationships between design parts are encoded among design parts in OWL, and the integrity constraints are encoded among design parts and stored as database triggers. Consistency of the design data is validated before any design change is stored in a database.

3.5 About SOA and APIs

In SOA the software functionalities are encapsulated into services, which are then cataloged to a service registry or repository (similarly to the idea of UDDI). More advanced registries that are capable of intelligent search, data classification or business logic are identified as brokers, aggregators or gateways.

Dion Hinchcliffe says that SOA and Open APIs are close cousins. They have similar goals, but SOA is usually "an overhead effort between IT and the business which ultimately allows businesses to achieve improved results and even serendipitous outcomes when it comes to the integration and leverage of existing investments in systems and data". SOAs also tend to be designed for internal use and are not charasterisable as facilitating rapid improvements. The APIs instead are considered to have an increased reach to new customers on the network, and tend to be designed for consumption by the broader world. [41]

Hinchcliffe's six most relevant lessons for SOA to take from open APIs are, in essence, the following [41]:

1. Ease of use: instead of using the traditional SOA's complex and less Web-friendly technologies, services should be made consumable from any platform, tool or programming language.

2. Good means for the customers understanding and measuring the costs for reporting and billing are needed.

3. In addition to bringing security, the account management in Open APIs is strongly keyed to tracking who is using the API and how. This information can be used to improve quality of service.

4. Public APIs can be used without lengthy company-to-company negotation and partnership process, which makes it possible to acquire far more partners more quickly and cost-effectively.

5. Part of the customer-centric aspect of Open APIs is that the developer community needs to be nurtured by providing them with the tools and information they need to successfully learn to use the service and to discuss all things related to it.

6. An ideal license is a flexible one that gives permission for the consumers of API services to re-use its capabilities in running their business. Needlessly limiting the use of data elsewhere can kill the business value of services.

According to Chet Kapoor of Sonoa, SOA and APIs are very different: SOA is a technology approach, one with sound computer science principles originated by architects, and API projects on the other hand are driven mostly by product managers. Technologically SOA and API are different in simplicity and scale, as most SOA is based on Web services standards that solve every corner case known to mankind, and APIs are mostly representational state transfer (REST) based. He claims that a change can be seen: many are adopting REST/Web API approach to solve their SOA projects, because there is value in applying API principles to SOA projects. Mainly because of the increased simplicity of the design, use, visibility and reporting. [40]

It seems that providing sensible externally accessible APIs for a selection of services can possibly improve the overall benefits and usability of the SOA-based system. Online APIs are so useful because it makes sense to have the service offered in as simple and accessible format as possible for a wide variety of users, and in a sense, the adding of semantics is also a way to add more possible users: the computers agents themselves.

4 Requirements of AI to API Design

4.1 Intelligent Agents

Distributed AI is a subfield of AI that studies how to solve problems requiring some kind of intelligence through distributed or parallel computations [44]. An intelligent agent is important concept in distributed AI. It is an autonomous entity which perceives its environment through its *sensors* and then reacts or acts upon (the same or another) environment using its *actuators* or *effectors*. In other words an intelligent agent is simply an agent that is capable of intelligent or rational behavior.

Many agents are required in most distributed solutions [44]. In a multi-agent based system the basic component is an independent agent that acts in an environment without external control logic. An important aspect of these agents is that they are relatively independent — there is no centralized ”brain” that controls all individual agents. However, this does not mean that the agents would not communicate with other agents and the rest of the system. That being said, the field of multi-agents is theoretically dominated and actual applications that facilitate intelligent multiagent systems are so far almost non-existant. [44]

4.2 Ontologies

Ontology in general is a major branch of metaphysics concerned with the nature or essence of being or existence, as well as a theory or conception relating to the nature of being. In logic an ontology is a system similar in scope to modern predicate logic, which attempts to interpret quantifiers without assuming that anything exists beyond written expressions. [3]

In computer science it is quite the same: a specification of a conceptualization such as the description of abstract categories of concepts or entities and their relations. Defining and applying ontologies is an essential part of the semantic web, and ontologies are often also called semantic data models. The previously mentioned OWL 2 family of ontology languages is the most prominent technical context example for mapping the ontological information.

However Defining ontologies is still far from being without problems: definitions for all possible concepts does not exist (and making the definitions of all the required concepts in a single ontology remains challenging), ”*the restrictions imposed on the instances, types and relations are opaque to the software trying to understand the Ontology's knowledge*” [44]. It can also be difficult to derive new concept restrictions by processing old restrictions, and a problem with some formal approaches is that

almost every assertion has exceptions which also need to be expressed. ”Rabbits usually have four legs, but a rabbit may have just three legs and still be a rabbit” [44] is an analogy to illustrate this problem.

This relates to a semantic API’s design as the ontologies need to be rather explicitly expressed to convey the necessary semantics: selection of an already existing or definition of a new ontology is relevant to designing the API for a semantic AI service.

Fortunately there are tools that support the creation and editing of OWL ontologies, reasoning over ontologies, and using ontologies in applications and majority of these tools require some kind of underlying API that allow ontologies to be loaded, manipulated and queried. For working with OWL 2 Ontologies there is a rather well documented open source Java API available. [45, 46]

4.3 Expert Systems

Expert systems are programs that act by a set of rules that attempt to capture a certain portion of the knowledge of the human experts. Being usually narrow in their scope, expert systems focus on some well-defined problem domains that they have a knowledge base of.

The term ”Expert systems” is old (many papers from the 80s can be found) and may be outdated as expert systems can be seen to have been evolved further to joint cognitive systems, intelligent systems, intelligent assistant systems, and context-based intelligent assistant systems. [47]

The paper titled ”Artificial Intelligence Arrives to the 21st Century” by A. Guzman-Arenas that was referenced previously might not be the most scientifically accomplished one, but it presents an interesting listing of AI applications that he claims to have not only academical but also commercial value: ”Expert Systems, visual inspection systems, and many commercial systems (such as those in data mining of large amounts of data) using neural networks and genetic algorithms, or fuzzy sets, to cite a few. More could have been produced by AI, if it were not for the fact that as a domain matures, it abandons AI.” [44] Although many might disagree about the last remark, it is an interesting claim. In essence, he suggests that when a deeper (or formal) understanding is developed on a problem previously handled with AI the need for an AI solution is removed.

4.4 IaaS, Intelligence as a Service

SOA is an approach to build distributed systems that deliver application functionality as services to end-user applications or to build other services. In using SOA to design distributed applications, the use of web services can be expanded from simple client-server models to systems of arbitrary complexity. Parts of the whole system can include IaaS (which can be thought as AI services in SOA), that function as more or less independent services that the whole system uses in for example any form of intelligent analysis or decision making.

An IaaS service would use the same way of communication as SOA architecture. Semantics could be included by using for example XML Schema. However the particular details and ontologies etc. used have little to do with the API design itself. There is an "Intelligence as a Service" Master's thesis by Viljo Pilli-Sihvola [48] that serves as a more detailed introduction to IaaS.

5 Designing Semantic APIs to Intelligent Software

If Bloch's general API design guidelines described in chapter 2.2 are valid, which seems to be feasible, something can be said about the design of *any* good API. Without repeating all the implicit guidelines here I infer that the semantic API to an AI software would consist of a core API which would be essentially unchangeable, unambiguously defined and as minimalistic as is reasonable while still offering all the required features. It is also obviously essential to include and transfer the semantics, and for this purpose the technologies recommended by W3C are the most relevant candidates.

But in addition to this core API an intelligent service might benefit from having individual APIs for each of it's users or user groups. These higher level APIs would utilize the core API and possibly some additional features or services, in the manner of a custom interface. An API like this could vary depending on the client's particular requirements, or be defined by the API's own learning process related to it's use. Monitoring and customization tools are often useful in similar applications, they might be useful in optimizing the APIs to better suit each particular user group. Adding these layers to an API might also be a way to increase security assuming that the access to the core API itself would be limited.

Furthermore if certain users of the API need additional features and functions the API can be expanded on top of the core API. I see no reason why the idea of modularity and inheritance could not be applied to APIs. Intricacies of a more com-

plex API could be built on a hierarchy of layers. A client would access its particular API on a higher layer, and the services logic would allow the higher layer API to access the core API (possibly via further intermediate APIs) maintaining the possibility of facilitating additional user related logic in the process. This could be a sensible approach when for example there was the need to slightly alter an API for each user individually. There could even be an automated method of formulating an API for a certain client according to some parameters possibly obtained by intelligent learning.

However, dynamically increasing the features of an API like this violates a number of the general API design principles: it is likely to be harder to learn and document, and therefore also harder to utilize, even if meant to be automatically used by other applications. To not violate these design guidelines, even a semantic API that is designed to be used by other software would at least need to communicate it's structure to the user software accessing it. This seems to apply whether the code be machine or man generated. Also if an API is built modular like this the core API should never be found to be too limited for accessing the core service, otherwise the additional services built on top of it could be rendered useless.

6 Conclusion

The automatic utilization of intelligent software via APIs and Web Services is a topic having not only academic but also great practical value. Services like these could be used for example for diagnostics, prediction and monitoring of large chunks of real-time data. An example of such a system could be a security video analysis software that analyzes the raw video footage from a security camera, and is able to reliably and automatically distinguish the movement caused by wind from an actual intruder. Closed versions of similar systems actually already exist, but an explicit and well suited method for publishing the wide variety of AI software as functional Web services cannot be found in the available literature. There is however definitely the need for such research.

Admittedly the results of the thesis may appear thin to the reader. The problem appears to be quite difficult to properly explore without actually implementing, testing and describing concrete prototypes of semantic APIs to real functional AI services. That was unfortunately not feasible for this thesis, but would be the logical next step in exploring the subject. I suspect that the real intricacies of the problem would only then be truly revealed.

References

[1] *Cloud Computing Glossary*, The Cloud Pyramid by GoGrid, 2010. Available Online at <URL: `http://pyramid.gogrid.com/glossary/`>, referenced 29.11.2011.

[2] *OASIS, Reference Model for Service Oriented Architecture 1.0*, available in WWW <URL: `http://docs.oasis-open.org/soa-rm/v1.0/soa-rm.pdf`>, p. 8, referenced 29.11.2011

[3] *The Oxford English Dictionary*, Oxford University Press, Oxford, 2011. Online version available at <URL: `http://www.oed.com/`>, referenced 29.11.2011.

[4] *WikiWikiWeb*, Cunninham & Cunningham, Inc., 1.12.2011. Available Online at <URL: `http://c2.com`>, referenced 29.11.2011.

[5] Robillard, Martin P. *What Makes APIs Hard to Learn? Answers from Developers*. IEEE Software vol. 26, no. 6 (Nov/Dec 2009), p.27. 2009.

[6] McLellan, S.G., Roesler, A.W., Tempest, J.T., and Spinuzzi, C.I., *Building More Usable APIs*, IEEE Software, 15(3), 1998.

[7] Farooq, Umer and Zirkler, Dieter, *API peer reviews: a method for evaluating usability of application programming interfaces*, Proceedings of the 2010 ACM conference on Computer supported cooperative work, CSCW '10, 2010.

[8] Joshua Bloch, *How to design a good API and why it matters*, Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications, OOPSLA '06, Google Inc., 2006, Also the lecture at Google Tech Talks January 24, 2007.

[9] Daughtry, John M. and Farooq, Umer and Stylos, Jeffrey and Myers, Brad A., *API usability: CHI'2009 special interest group meeting*, Proceedings of the 27th international conference extended abstracts on Human factors in computing systems, CHI EA '09, 2009.

[10] The Java Tutorials, *Unchecked Exceptions — The Controversy*, Oracle, 1995,2011. <URL: `http://docs.oracle.com/javase/tutorial/essential`>, referenced 29.11.2011

[11] Cwalina, K. and Abrams, B. *Framework design guidelines: Conventions, Idioms and Patterns for Reusable .NET Libraries*. Pearson Education, 2005.

[12] *API Dashboard*, ProgrammableWeb, available in WWW <URL: http://www.programmableweb.com/apis>, referenced 29.11.2011

[13] *The API Scorecard*, ProgrammableWeb, available in WWW <URL: http://www.programmableweb.com/scorecard>, referenced 29.11.2011

[14] Myers, Brad A; Jeong, Sae Young; Xie, Yingyu; Beaton, Jack; Stylos, Jeff; et al. *Studying the Documentation of an API for Enterprise Service-Oriented Architecture* Journal of Organizational and End User Computing 22. 1 (2010): 23.

[15] *The W3Schools Semantic Web Tutorial*, w3schools.com ©1999-2011 by Refsnes Data. Available online at <URL: http://www.w3schools.com/semweb>, referenced 29.11.2011.

[16] T. Berners–Lee, J. Hendler and O. Lassila, *The semantic Web*. Scientific American, 284 5 (2001), p. 34.

[17] *The W3Schools W3C Tutorial*, w3schools.com ©1999-2011 by Refsnes Data. Available online at <URL: http://www.w3schools.com/w3c>, referenced 29.11.2011.

[18] *Current Members of W3C*, The World Wide Web Consortium (W3C). Available online at <URL: http://www.w3.org/Consortium/Member/list>, referenced 11.12.2011.

[19] *W3C Standards*, W3C 2010. Available online at <URL: http://www.w3.org/standards>, referenced 29.11.2011.

[20] The OASIS Web Services Interoperability (WS-I) Member Section, ©2011 OASIS. Available online at <URL: http://www.oasis-ws-i.org, referenced 29.11.2011.

[21] *A Semantic Web Primer for Object-Oriented Software Developers*, W3C Working Group Note 9 march 2006. Available online at <URL: http://www.w3.org/TR/2006/NOTE-sw-oosd-primer-20060309/>, referenced 29.11.2011.

[22] *RDF Primer*, W3C Recommendation 10 February 2004. Available online at <URL: http://www.w3.org/TR/2004/REC-rdf-primer-20040210/>, referenced 29.11.2011.

[23] *RDFa Primer*, W3C Working Group Note 14 October 2008. Available online at <URL: http://www.w3.org/TR/xhtml-rdfa-primer/, referenced 29.11.2011.

[24] *OWL 2 Web Ontology Language Document Overview*, 3C Recommendation 27 Oct 2009. Available online at <URL: http://www.w3.org/TR/2009/REC-owl2-overview-20091027/>, referenced 29.11.2011.

[25] *OWL 2 Web Ontology Language Primer*, W3C Recommendation 27 Oct 2009. Available online at <URL: http://www.w3.org/TR/owl-primer/>, referenced 29.11.2011.

[26] *SPARQL Query Language for RDF*, W3C Recommendation 15 January 2008. Available online at <URL: http://www.w3.org/TR/rfd-sparql-query/>, referenced 29.11.2011.

[27] *SKOS Simple Knowledge Organization System Primer*, W3C Working Group Note 18 August 2009. Available online at <URL: http://www.w3.org/TR/2009/NOTE-skos-primer-20090818/>, referenced 29.11.2011.

[28] *SOAP Version 1.2 Part 0: Primer (Second Edition)*, W3C Recommendation 27 April 2007. Available online at <URL: http://www.w3.org/TR/2007/REC-soap12-part0-20070427/, referenced 29.11.2011.

[29] E. Christensen, F. Curbera, G. Meredith, S. Weerawarana, *Web Services Description Language (WSDL) 1.1*, W3C Note 15 March 2001. Available online at <URL: http://www.w3.org/TR/wsdl>, referenced 29.11.2011.

[30] *Web Service Semantics - WSDL-S*, W3C Member Submission 7 November 2005. Available online at <URL: http://www.w3.org/Submission/WSDL-S, referenced 29.11.2011.

[31] *Semantic Annotations for WSDL and XML Schema*, W3C Recommendation 28 August 2007. Available online at <URL: http://www.w3.org/TR/sawsdl, referenced 29.11.2011.

[32] Justin R. Erenkrantz, Michael Gorlick, Girish Suryanarayana, Richard N. Taylor: *From Representations to Computations: The Evolution of Web Architectures*. Proceeding ESEC-FSE '07. 2007.

[33] *The W3Schools RDF Tutorial*, w3schools.com ©1999-2011 by Refsnes Data. Available online at <URL: http://www.w3schools.com/semweb>, referenced 29.11.2011.

[34] Pablo N. Mendes, Max Jakob, Andrés García-Silva, Christian Bizer: *DBpedia spotlight: shedding light on the web of documents*. Proceeding I-Semantics '11. 2011.

[35] *UDDI R.I.P.*, Stefan Tilkov's Weblog, May 2010. inoQ Deutschland GmbH. Available online at <URL: http://www.innoq.com/blog/st/2010/03/uddi_rip.html>, referenced 1.12.2011.

[36] Justin R. Erenkrantz, *Web Services: SOAP, UDDI, and Semantic Web*, ICS 221 University of California, Irvine. May 2004.

[37] Melchiori, Michele, *Hybrid techniques for web APIs recommendation*, Proceedings of the 1st International Workshop on Linked Web Data Management, LWDM '11, 2011.

[38] Zouev, Eugene. *Semantic APIs for programming languages* (Russian), Software Engineering Conference (CEE-SECR), 6th Central and Eastern European 2010.

[39] Xun Li, Sang Bong Yoo, *Integrity validation in semantic engineering design environment*, Inha University, South Korea, 2010.

[40] Chet Kapoor, *SOA and API - many differences, but coming closer*, edgeofthecloud.com, 22nd June 2009. Available online at <URL: http://edgeofthecloud.com/2009/06/>, referenced 29.11.2011.

[41] Dion Hinchcliffe, *Running your SOA like a Web startup*, Enterprise Web 2.0 blog at ZDNet. 13th June 2009. Available online at <URL: *http://www.zdnet.com/blog/hinchcliffe*>, referenced 29.11.2011.

[42] *Semantic Web Service Directory*. Maintained by Dr. Khalid Belhajjame ©2008. Available online at <URL: http://www.cs.man.ac.uk/~khalidb/sws/sws_directory>, referenced 29.11.2011.

[43] Abed, Mariam Abed Mostafa, *Automatic selection and invocation of a WSMO: based semantic web service*, Proceedings of the 2011 International Conference on Intelligent Semantic Web-Services and Applications, ISWSA '11, 2011.

[44] Adolfo Guzman-Arenas, *Artificial Intelligence Arrives to the 21st Century*, Lecture notes in Computer Science Volume 4293/2006 MICAI 2006: Advances in Artificial Intelligence. 2006.

[45] The OWL API. Sourceforge.net. Available online at <URL: `http://owlapi.sourceforge.net`>, referenced 14.1.2012.

[46] Matthew Horridge, Sean Bechhofer, *The OWL API: A Java API for Working with OWL 2 Ontologies*, The University of Manchester, UK, 2009.

[47] Patrick Brezillon, *From expert systems to context-based intelligent assistant systems: a testimony*, Knowledge Engineering Review 26. 1 (Mar 2011): 19-24. Cambridge University Press, 2011.

[48] Viljo Pilli-Sihvola, *Intelligence as a Service*, Master's thesis in Information Technology, December 15, 2010. University of Jyväskylä, Department of Mathematical Information Technology.