Information Technology

A History and Future of Web APIs

--Manuscript Draft--

Manuscript Number:	ITIT-13-1035R2
Full Title:	A History and Future of Web APIs
Article Type:	Special Issue
Keywords:	D.1.3 [Programming Techniques: Distributed programming]; D.2.11 [Software Architectures]; K.2 [History of Computing: Systems]
Corresponding Author:	Jacek Kopecky, Ph.D. University of Portsmouth UNITED KINGDOM
Corresponding Author Secondary Information:	
Corresponding Author's Institution:	University of Portsmouth
Corresponding Author's Secondary Institution:	
First Author:	Jacek Kopecky, Ph.D.
First Author Secondary Information:	
Order of Authors:	Jacek Kopecky, Ph.D.
	Paul Fremantle
	Rich Boakes, Dr.
Order of Authors Secondary Information:	
Abstract:	Distributed information systems predominantly have client-server architectures, as does the Web itself. In this article, we review the evolution of the interface of client-server distributed systems, from Messaging and RPC systems that predate the Web, to RESTful Web APIs. We highlight the often overlooked importance of the client-server interface in Web applications, and we reference historic and current systems to discuss the roles of "Web Service" technologies and Service-Oriented Architectures. Considering the future, we point out four directions in which we can see Web APIs moving, including the incorporation of hypermedia and semantics.

Powered by Editorial Manager® and ProduXion Manager® from Aries Systems Corporation

Manuscript (-.tex, -.word)
Click here to download Manuscript (-.tex, -.word): cameraready.tex

Information Technology

A History and Future of Web APIs

Dr. Jacek Kopecký: School of Computing, University of Portsmouth, Buckingham Building, Lion Terrace, PO1 3HE Portsmouth, UK
Tel: +44-23-92846428, E-Mail: jacek.kopecky@port.ac.uk
Jacek Kopecký is a Lecturer in Information Systems at University of Portsmouth, United Kingdom. He was involved in Web Services and Semantic Web research at the Open University (UK) and at the University of Innsbruck (Austria), where he received his doctorate for work on automation supported by lightweight semantic descriptions. Jacek has contributed to Web services standardization at the W3C, and he chaired the W3C Semantic Annotations for WSDL (SAWSDL) working group.

Mr. Paul Fremantle: School of Computing, University of Portsmouth, Buckingham Building, Lion Terrace, PO1 3HE Portsmouth, UK
E-Mail: paul.fremantle@port.ac.uk, paul@wso2.com
Paul Fremantle is CTO and Co-Founder of WSO2, a company providing Open Source enterprise middleware. Paul is one of the founders and committers on the Apache Synapse project, as well as participating in a number of other open source initiatives. Paul chaired the OASIS WS-RX technical committee and was previously a Senior Technical Staff Member at IBM. Currently, Paul is pursuing a doctoral degree at University of Portsmouth.

Dr. Rich Boakes: School of Computing, University of Portsmouth, Buckingham Building, Lion Terrace, PO1 3HE Portsmouth, UK
E-Mail: rich.boakes@port.ac.uk
Rich Boakes teaches at University of Portsmouth and is Course leader for BSc (Hons) Web Technologies. Prior to entering academia he worked at IBM and Netscape in consultancy roles, helping customers design, deploy and maintain open and web-based systems.

Keywords: D.1.3 [Programming Techniques: Distributed programming]; D.2.11 [Software Architectures]; K.2 [History of Computing: Systems]

MS-ID: Heft: / ()	jacek.kopecky@port.ac.uk	February 26, 2014

Abstract

Distributed information systems predominantly have client-server architectures, as does the Web itself. In this article, we review the evolution of the interface of client-server distributed systems, from Messaging and RPC systems that predate the Web, to RESTful Web APIs. We highlight the often overlooked importance of the client-server interface in Web applications, and we reference historic and current systems to discuss the roles of "Web Service" technologies and Service-Oriented Architectures. Considering the future, we point out four directions in which we can see Web APIs moving, including the incorporation of hypermedia and semantics.

Zusammenfassung

Ebenso wie das Web besitzen verteilte Informationssysteme vorwiegend Client-Server Architekturen. In diesem Artikel untersuchen wir die Entwicklung von Schnittstellen verteilter Client-Server Systeme beginnend mit dem Web vorausgehenden Messaging- und RPC-Systemen, bis hin zu RESTful Web APIs. Wir beleuchten den oft vernachlässigten Stellenwert der Client-Server-Schnittstelle bei Webanwendungen, zudem beziehen wir uns auf historische sowie gegenwärtige Systeme, um die Rollen von "Web-Service"-Technologien und serviceorientierten Architekturen zu diskutieren. In Hinblick auf die Zukunft zeigen wir vier Richtungen auf, in die sich Web APIs unserer Ansicht nach bewegen können, die Einbindung von Hypermedia und Semantik eingeschlossen.

1 Introduction

The Web was proposed as a hypertext system for sharing data and information among scientists [4], and has grown into an unparalleled platform on which to develop distributed information systems. Most new information systems are now Web applications, and many older systems (e.g. government, military and enterprise systems) have been given web-based APIs and interfaces.

Web architecture is predominantly client-server, a model that is often extended to 3-tier and n-tier architectures [8]. Client-server models allow us to isolate clients and servers to analyse their architectures, and to separately consider the interfaces between them. We focus here specifically on these interfaces.

Principled, forward-looking interface design benefits the distributed applications that use it by: 1) enabling the independent evolution of systems on either side of the interface; 2) affording optimal use of the underlying network, and; 3) spurring unforeseen adoption and growth (e.g. through third-party clients, or the adoption of the interface itself in wider reaching protocols). This final aspect leads to the "network effect", where the effectiveness of a network of connected systems grows with each additional connected system.

Therefore, Web application architects should understand the importance of the public-facing programmatic APIs exposed by their applications. Much has been written about good practices for so-called RESTful Web APIs, esp. [26], along with many online sources.

In this article, we present client-server interfaces in historical context, highlighting how technologies and approaches (that are sometimes complementary, divergent or contradictory) have been embraced and assimilated in the Web-based systems and applications.

The evolution of distributed systems began with message-passing primitives that evolved from OS support for interprocess communication, therefore we start by looking at Messaging Systems *that expose network communication to programs* (Section 2). We further discuss Remote Procedure Calls (RPC) that hide network communication behind programming interfaces, but introduce architectural and practical issues (Section 3). Then we consider Service-Oriented Architectures (SOA) through their realisation as Web Services which agglomerate loosely-coupled and coarse-grained components aligned with business (not IT) needs (Section 4). A growing understanding of HTTP, along with ubiquitous JavaScript clients, enabled true emergence of services on the Web, in the form of Web APIs (discussed in Section 5). Finally, we will look at the most recent progress of Web APIs, adopting further traits of the Web, especially hypertext and linked data (Section 6).

In the end, in Section 7 we look back at the developments in this space. We identify a number of characteristics of Web interfaces that make them highly effective and reusable; in short, we call such interfaces *Webful*.

2 Messaging systems

The original communication primitives for distributed systems came from the operation systems concept of asynchronous messaging. Extending the principle of directly sending a message from one process to another, there are two very common higher-level paradigms for messaging systems: *queueing* and *publish/subscribe*. Both aim to decouple the client from the server (or in messaging terms the *producer* from the *consumer*).

Queues comprise three useful aspects: firstly, decoupling producers and consumers with named queues. Secondly, queues can enfoce *ordered delivery*, ensuring that messages are dealt with in the same order they were sent. Finally, a queue may feature Qualities of Service (QoS) such as reliable, *exactly-once* delivery.

Queues have become particularly popular in cloud deployments because they support elastically scaled servers: clients can add messages to a queue, and servers can take messages from the queue and act upon them. Each server can thus be maximally loaded, pulling work when it has capacity to handle it. When the queue is long, more servers can be started to handle the work.

The publish/subscribe ("pub-sub") model is another way of decoupling producers from consumers. Pub-sub systems differ from queues by (a) defining *topic names* that can be subscribed to by multiple consumers, and (b) enabling topic hierarchies, so a subscription to any tree node gives an aggregate of child topics.

Messaging systems in general focus their attention on message transmission, obscuring the intended functionality and behaviors of the clients and servers. As commented by Nelson in [24], the message-passing approach "can have several disadvantages from a language design standpoint. The first is that messages introduce a control primitive that is quite different from procedure-oriented mechanisms. This can be a problem for procedural [...] languages where a message-passing operation is a new communication primitive."

However, recently a number of programming languages and paradigms have emerged based on the messaging model, in particular the Erlang language with the Actor model [31], and derivatives such as the Akka framework in the Scala language [16].

3 RPC Systems

RPC (Remote Procedure Call) implementations encapsulate all communication code making communicating programs easier to understand. This complexity-hiding is popular, supporting the original intent of Birrell and Nelson whose "primary purpose of our RPC project was to make distributed computation easy" [5].

Arguably, RPC made distributed computation *too easy*, and RPC systems suffered from a number of issues, both architectural and practical, including the "Fallacies

of Networked Computing" [18]. Here, we show five issues that pertain to the client-server interface:

1. RPC systems tend to be *tightly coupled*: remote calls look like calls in the same program, but two codebases are involved, leading to issues when one is updated.

2. RPC hides network communication, so developers may forget or ignore potential *network failures* that may lead to systems in inconsistent states.

3. RPC obscures the inherent *insecurity* of networks, where multiple parties may observe, manipulate or inhibit data-flows.

4. As illustrated in [5], RPC systems were designed to perform close to the speed of native procedure calls. This promotes *fine-grained, conversational*, and often *synchronous* procedure interfaces, whereas in distributed systems, coarse grained, stateless, asynchronous interfaces are more scalable, more resilient to latency and better able to *exploit inherent parallelism*. [12, 2].

5. Finally, RPC enables the deployment of clients and servers on diverse hardware and software platforms. This creates space for *interoperability issues*, an inherent weakness of the RPC paradigm. The similarity with native procedure calls encourages the use of native data types and structures that do not always map well between platforms.

These issues are avoidable, whether through careful design and planning, and (for example) by using extensions to RPC (such as using asynchronous calls), however, RPCs tends to hide these issues, rather than exposing them and promoting good practice. These issues were not solved in RPC's successors, such as CORBA.[1]

4 Web Services

The Web was initially built as a client-server system for human-oriented hypertext documents [3]. Early on, Web application interactivity was constrained to following links and reading documents. Later, *forms* were incorporated into HTML, and HTTP was extended to include the ability to *send* data as well as request it. Subsequently, the W3C developed XML and the potential of using Web technologies as a platform for distributed systems, gained recognition.

In 1999, two related technologies were released: XML-RPC [32] and SOAP 0.9 (ultimately standardized as SOAP 1.2 in [28]), and a great standardization and technological push movement was born, under the name *Web Services*. The movement was led by major software vendors including IBM, Microsoft, Sun, Oracle and BEA, whose aim was to create a standard interoperable approach for interconnecting large systems.

A key aspect of Web Services was the description of services via the Web Services Description Language (WSDL). WSDL made it very simple to take existing objects and map them into services, and to take existing WSDLs and map them into objects on the client side. The result was a success for ease of use, but it furthered an RPC view of SOAP. The second major editions of both core Web Services specifications (SOAP 1.2 and WSDL 2.0) made significant changes to promote a more message-oriented approach, but were finalised too late, and even today, six years after the final version of WSDL 2.0, there is almost no adoption of it [23].

One of the biggest challenges to the success of the Web Services standards stemmed from the political arguments between vendors. This led to the creation of multiple alternative standards (e.g. WS-ReliableMessaging vs WS-Reliability). Even worse, WSDL 1.1, considered a core part of the standards, actually captured two completely alternative ways of doing the same thing due to the influence of two vendors with different approaches. All of these issues led to additional complexity and less interoperability.

Despite these issues, SOAP is widely used within organizations (see e.g. [22]). At the public interface, there have been major uses, for example eBay uses SOAP to communicate with apps on PCs. SOAP is also used in many cases where security and reliability are required, such as a European public procurement project PEPPOL.eu. However, while many internal systems continue to be developed with SOAP and Web Services, most new Web APIs are written without those technologies.

We can summarize the reasons for the shift away from SOAP for Web APIs:

- Complexity of the SOAP stack made it unpopular with developers. The simplicity of HTTP-based APIs makes it a preferred approach.

- The rise of mobile apps: Android and iOS development tools come with good support for JSON and HTTP, but little support for SOAP.

- A better understanding of RESTful design principles (see below) has attracted developers.

5 Web APIs, RESTfulness

As we've mentioned in the preceding section, the early Web only had one type of client program: the browser. Some early systems tried to "scrape" websites in order to create application-based clients, but the focus on browsers really changed in 2000, when Salesforce and eBay both released their Application Program Interfaces

[1] `http://en.wikipedia.org/wiki/Common_Object_Request_Broker_Architecture`

(APIs) [21], which were intended for desktop tools, and for integration with other existing systems.

As noted in [21], "XML APIs were part of Salesforce.com from day one. Salesforce.com identified that customers needed to share data across their different business applications, and APIs were the way to do this." For eBay, "the launch of [its] API was a response to the growing number of applications that were already relying on its site" [21], for example to make it easier for sellers to post listings, especially in bulk.

If Salesforce.com and eBay were among the first of a trickle of Web APIs, a true flood came with the rise of more dynamic websites. As the browser support for JavaScript and XMLHttpRequest matured into Ajax [13], it became practical for the browser to load parts of a web page on the fly, and submit changes from the user without triggering a reload of the web page. This change is often referred to as *Web 2.0*. The application that truly showcased this new capability was Google Maps, which was soon followed by a Web API that let other websites embed and customize maps.

As Web 2.0 websites implement interactivity in their web pages, the JavaScript code in those pages becomes a program that needs an API to access the server. Once a website has such an API, it can be a relatively small step to make the API public for use by other websites, and indeed by other software as well.

Another factor that contributes to the growing number of available APIs is mobile computing. With the dramatic spread of smartphones and tablets, there is a new wave of software, almost all of which is supported by servers online. There are even mobile apps supported by a server with a Web API that isn't even used on their website. A case in point is Instagram.com, a photo-sharing app for smartphones, which only added a Web version of its functionality when users called for it, and when third-party websites gained popularity by using the Instagram API to give Instagram users Web access to their photos [21].

5.1 RESTfulness

The core protocol of the Web – HTTP – has a well-known limited set of operations: GET, POST, PUT, DELETE, and the newest one PATCH[2]. In Web terminology, this is called the *uniform interface*, see [12]. As HTML forms only supported the GET and POST methods, and the most visible difference between these methods was in how parameters are passed to the server, the two methods were easily misunderstood and misused. This led to unexpected behavior and loss of data.

Let us illustrate with a typical such mistake: an API allows its clients to delete a record (for instance, unsubscribe a person from a mailing list) by simply following a link (making a GET request against, for example, `http://example.com/remove?email=joe@example.com`). This works perfectly in testing, but as soon as the system is deployed to the public, a search engine crawler will try to follow the links and the records are gone. This is because the search engine relies on the HTTP specification which says the GET should have no side-effects and therefore be safe to call.

Eventually, the criticisms of such misuse of HTTP united under the banner of *"RESTfulness"*, adopting the acronym REST invented by Roy Fielding in his PhD thesis [12], which describes the architecture of the Web. REST stands for Representational State Transfer, which alludes to the fact that the HTTP verbs are used to transfer representations (in a well-defined media type) of the state of a resource. Many arguments are put forward for why Web APIs should *be RESTful*, i.e., they should follow the recommendations of Fielding's thesis and the various standards that define the Web.

5.2 RPC and Messaging in Web APIs

Some Web APIs can be described as RPC-like. The strongest manifestation of an RPC-like Web API is the presence of procedure names in the API's URIs. For instance, Flickr.com API call URIs contain a `method` parameter.[3] In effect, the resource space of the API is shaped by the functionalities that Flickr provides, not by the kinds of data that is being manipulated.

Some proponents of RESTfulness criticize RPC-like APIs as not being RESTful (e.g. [27]). Indeed, the term "Resource-oriented Architecture" is being put forward as alternative to SOA (cf. [6]), emphasizing the principled structuring of an API URI space, and the full use of the HTTP uniform interface for resource manipulation, along with HTTP's caching and security mechanisms.

Some Web APIs also need to employ messaging, mainly for updates and push notifications. These have typically been implemented either as inefficient polling or as limited streaming (for instance, Comet[4] maintains a one-directional HTTP stream that can be adversely affected by buffering proxies). The result has been the creation of the WebSocket standard [10] to allow true streaming over HTTP. For example, MQTT over WebSockets[5] is growing in popularity as a way of providing pub-sub in compliance with Web architecture.

5.3 Managed Web APIs

A major trend over the past few years has been the introduction of *managed Web APIs*. The main externally-visible difference in managed Web APIs is that they re-

[2]Two further operations are worth mentioning: HEAD and OPTIONS for retrieving representation and resource metadata

[3]Example full URI: `http://api.flickr.com/services/rest/?method=flickr.activity.userPhotos`

[4]`http://infrequently.org/2006/03/comet-low-latency-data-for-the-browser/`

[5]`http://www.hivemq.com/mqtt-over-websockets-with-hivemq/`

quire the client to provide an API Key which must be passed as part of every request. API Keys are required by many popular APIs including those provided by organizations such as Google, Twitter, eBay, Netflix and Facebook. Research during 2012 found that there were at least ten API providers that handle more than 1 billion API calls per day each [7], and need to be managed.

API Keys are typically used to authenticate the user and/or identify the client system thereby enabling numerous management functions, including:

- authorization, access control, and monitoring of API use,
- throttling access per client, or per application (for example, preventing a bug in a one client application from causing an outage of the API through an inadvertent distributed denial of service attack),
- enforcing remote applications to upgrade to a new version of an API through deprecation of the key,
- routing API requests to different servers (e.g. production vs sandbox, or high- vs low-priority),
- and the ability to monetise the API.

Recently the OAuth2 specification [9] has been used to provide a standard approach for issuing and checking API Keys. In this model, the API Key is an OAuth2 Bearer Token. This supports two models: i) two-legged, where only the calling application is identified; and ii) three-legged, where both the calling application and the calling user are identified by the key.

API Key issuance depends upon a client application *subscribing* to an API. Early API Key initiatives adopted manual web-based processes to approve subscriptions, however, widespread use of Web APIs has meant that providers of popular services may have many thousands of subscribers, so API Portals (where an organization may advertise its APIs, provide documentation & examples, handle billing, etc.) have become necessary.

API Portals are designed to enable developers to use APIs effectively with minimum direct help from the API creators, instead relying on web-based documentation. Some API Portals also provide statistics or monitoring, especially if there are commercial limits on API use.

While API Keys have been deployed primarily on the public Web, managed APIs can bring many of the same benefits within large organizations.

5.4 Security

The topic of web security is beyond the scope of this article; however, one important aspect is highly relevant. Web security has evolved considerably from the early days of SSL and Basic authentication – the current Web architecture for security now supports a number of federated identity models with API access.

A typical example is the social network Facebook. Social networks aim to connect users, and one way to do that is to look at the users' email contact lists. Previously, the system would ask the user to enter their Gmail or Hotmail account details, including password, and would "web scrape" the contact list from there. However, this gives access to the whole account and is highly insecure. From this requirement emerged a new standard *OAuth* which aimed to allow API access.

Interestingly, OAuth is now being used the other way round – as a *single-sign-on* mechanism, using Facebook identities in lieu of accounts on other websites. A newly developed specification, *OpenId Connect*[6] defines an API-friendly identity layer built over OAuth 2.0.

The pleasing aspect about this is that these standards are themselves Web APIs. The result is that they are synergistic to the success of Web APIs, and become part of the linked set of APIs that work together.

6 Web APIs in the (near) future

Having reviewed the current landscape in Web APIs and their degrees of embrace for the Web, we will highlight four directions in which Web APIs will be moving. We start from an observation that Web APIs are increasingly the basis for websites; then we look at a part of REST that isn't commonly implemented in APIs – using hypertext to control the application state; next we discuss whether and how Web APIs could be described in a machine-readable form; and finally we look at connections to Linked Data and semantic interoperability.

6.1 APIs as the basis of User Interfaces

An interesting development that we have noticed amongst many newly built websites is that the websites are built on the Web APIs. This is a reversal of the early days where parts of the website were treated as APIs by enterprising developers. Instead, the team builds the functionality as a set of APIs, and then creates the website as a client to those APIs.

This often comes from a "mobile-first" approach where the mobile client is built first (such as the aforementioned case of Instagram.com). It can be described as *API-first*, and it puts a premium on development of an excellent set of APIs, especially as the first users of those APIs are part of the same organisation.

6.2 Hypertext State Control

Today, Web APIs are moving towards RESTfulness, especially by considering the aforementioned Resource-oriented Architecture, with full use of HTTP. There is,

[6] `http://openid.net/connect/`

however, one part of REST [12] that is not commonly reflected in Web APIs: *hypertext as the engine of application state* (often abbreviated as HATEOAS).

HATEOAS is an indispensable part of the human Web. People follow links; each link points to a resource that is viewable, and links that are not valid in the given situation are not included and hence the client does not interact with them. The client only sees the valid operations. In contrast, programming interfaces traditionally offer a fixed set of operations, and the client must be programmed with the knowledge of which operations make sense in any given application state.

The hypertext model translates well into APIs [30, 11, 1]. For example, a document retrieved for a shopping transaction might contain links to parcel tracking information. Retrieving that resource may lead to further links, for example geographic URIs, and so forth.

A further important aspect of HATEOAS is that the links between resources mirror the relationships between objects in the real world, and between the state representations of those objects. Since those linkages are browseable both by application clients and humans, this makes the semantics of the application evident through exploration of the data and links.

There are three main benefits to this approach:

- *reliability* is improved by blocking invalid state transiton calls (because a client cannot attempt a state transition for which it does not have a link), and by the server controlling the links rather than having the clients construct URIs.
- *evolution* is simplified because the server may remove no-longer-supported state transitions and they won't be attempted by clients any more, and if any new transitions are added, existing clients will ignore them. These benefits are illustrated by existing hypertext-driven Web APIs.[7]
- *re-use* is encouraged through the ability to embed links into third-party websites or APIs. This is exemplified by the re-use of APIs such as Facebook Connect and Google Maps.

6.3 Web API Descriptions

Despite the aforementioned issues with WSDL, machine-readable standard API/interface description is one of the best aspects of the Web Services approach. However, in the RESTful space, there has been push-back against description languages because, in theory, the Web already has mechanisms for description. For example, a client can make an OPTIONS request on any resource to list the *content-types* available there.

In practice this is not effective because the content-type is usually application/json or application/xml, neither of which gives enough information to proceed. XML may have a namespace, but it seldom resolves to the correct schema. The result is that the available RESTful mechanisms for self-description are not used in practice.

An early comprehensive attempt at a WSDL-like language for RESTful services, the Web Application Description Language [15], has not seen significant uptake. We have observed wider interest in a more recent informal specification for RESTful description known as *Swagger*.[8] Its success can be tied to its focus on creating both human and machine-readable documentation incorporated into the Web API, as well as tools for integration with popular development frameworks.

Another attempt to provide description for RESTful services is the OData specification[9] which offers extensive metadata about accessible data sources. Despite some success for Swagger and OData, there is no consensus on a generic description approach for RESTful services. We expect to see both theoretical and pragmatic progress in this area.

6.4 Automation

The Web can be seen as the largest information system ever created. Increasingly, it is also becoming a large repository of open data, available for rich exploratory querying, for machine processing such as generating visualizations (esp. putting data on the map), and for combining multiple data sources. For example, a person moving to a new city can now use a map that shows houses for sale along with schools, public transport, and crime statistics.[10]

Some of this open data is being published as *Linked Data* (`http://linkeddata.org/`): "using the Web to connect related data that wasn't previously linked, or using the Web to lower the barriers to linking data currently linked using other methods." Linked Data builds on the Semantic Web standard RDF, which is a graph data model that uses Web URIs to capture pieces of data. Originally, the Semantic Web was a vision of the Web being machine-understandable (so that machines could intelligently answer complicated structured queries), and from the start, this vision also included machine-driven interoperability between services [17]. Linked Data is the first true sign of the work on the Semantic Web bringing fruit.

Service descriptions, inputs and outputs can be seen as data, therefore it is natural to consider the connection of Linked Data with Web APIs. Various efforts under the umbrella term Linked Services[11] have been

[7] E.g. `https://kenai.com/projects/suncloudapis/pages/Home`, `http://developer.netflix.com/docs/REST_API_Conventions`, and [20]

[8] `https://github.com/wordnik/swagger-core/wiki`

[9] `http://en.wikipedia.org/wiki/Open_Data_Protocol`

[10] `http://data.gov.uk/apps/soa4all-real-estate-finder`

[11] See `http://linkedservices.org/wiki/Main_Page` and `http://people.kmi.open.ac.uk/carlos/research/linked-services`

looking at how to represent service descriptions in RDF while working with the lightweight service description approaches common with Web APIs (e.g. [19]); and how to deal with APIs that themselves can produce and process linked data: e.g. [29] sees APIs as data sources and uses the RDF query language SPARQL to automatically decide which APIs to call, and [20] proposes a resource-oriented structure and a self-description schema for custom APIs to Linked Data sources.

One of the main goals of research in Semantic Web Services (and more recently in Linked Services) has always been to support automatic composition of services and APIs based on client requirements. This has proven to be an elusive goal, but efforts like [19, 29] show potential for *design-time composition* – while implementing a Web application, the developer may seek recommendations for existing data sources and processing APIs, given the kinds of data or functionality that is desired. Helping the developer lowers the bar on automation – the developer is able to evaluate the suitability of recommended APIs for a given problem, and to map incompatible data formats when needed. This is in contrast to trying automated composition of Web APIs at runtime, directly for a user who cannot be expected to be technologically skilled.

7 Summary

We have covered a great deal of ground, from the use of messaging and RPC approaches through to the RESTful design of Web APIs, HATEOAS, and Linked Data. It is noticeable looking at these approaches that designs that further the ecosystem of the Web (for example RESTful design, OAuth Security) are much more successful at Web API reuse than designs which do not. We attribute this to the network-effect and Metcalfe's Law [14]. These designs exhibit properties that include *RESTful*ness but also extend upon it, to what we call *Webful*ness. These properties include:

- *RESTful* – including use of the uniform interface, HATEOAS and proper use of links and Linked Data, well-defined content-types, and cache-ability.
- *API-first* – providing a first-class Web API that is sufficient to build any browser-based Web sites and applications ensures that the application is fully available to the wider ecosystem.
- *Managed* – Managed APIs ensure that the API is well documented, accessible to third-parties, throttled and available under an SLA, and versioned.
- *Federated Identity* – the use of federated identity models such as OAuth and OpenID Connect allows users and applications to utilize existing identities rather than re-create them for each new Web site. This promotes the network effect.
- *Automatable* – Linked Data and Linked Services enable services and APIs to be combined into new services and/or APIs. This recursive nature adds to the ecosystem of APIs in a consistent way.

Corporations and Lean startups [25] (a new type of organizations that outsource all business functions to Web-based services and APIs) can maximise the return on their systems investment by using and developing systems with Webful properties, because they encourage wider ecosystems where multiple connected systems collaborate.

References

[1] Rosa Alarcon, 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 Berlin Heidelberg, 2011.

[2] Akkihebbal L Ananda, BH Tay, and Eng-Kiat Koh. A survey of asynchronous remote procedure calls. *ACM SIGOPS Operating Systems Review*, 26(2):92–109, 1992.

[3] Tim Berners-Lee. Information Management: A Proposal. CERN, W3C. Available at `http://www.w3.org/History/1989/proposal.html`, March 1989.

[4] Tim J Berners-Lee. The world-wide web. *Computer Networks and ISDN Systems*, 25(4):454–459, 1992.

[5] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[6] Dominic Duggan. *Resource-Oriented Architecture*, pages 359–415. John Wiley & Sons, Inc., 2012.

[7] Adam DuVander. Which APIs Are Handling Billions of Request Per Day? ProgrammableWeb blog[12], May 2012.

[8] Wayne W Eckerson. Three tier client/server architectures: Achieving scalability, performance, and efficiency in client/server applications. *Open Information Systems*, 3(20):46–50, 1995.

[9] D. Hardt (ed). The OAuth 2.0 Authorization Framework. RFC 6749, IETF, October 2012. Available at `http://www.rfc-editor.org/rfc/rfc6749.txt`.

[12] `http://blog.programmableweb.com/2012/05/23/which-apis-are-handling-billions-of-requests-per-day/`

[10] I. Fette and A. Melnikov. The WebSocket Protocol. RFC 6455, IETF, December 2011. Available at http://www.rfc-editor.org/rfc/rfc6455.txt.

[11] Roy T. Fielding. REST APIs must be hypertext-driven. Blog article, available at http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven, October 2008.

[12] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, Irvine, 2000. Chair: Richard N. Taylor.

[13] Jesse James Garrett. Ajax: A New Approach to Web Applications. Blog article, available at http://www.adaptivepath.com/ideas/ajax-new-approach-web-applications, February 2005.

[14] George Gilder. Metcalf's law and legacy. *Forbes ASAP*, 27, 1993.

[15] Marc J. Hadley. Web Application Description Language (WADL). Technical report, Sun Microsystems, November 2006. Available at https://wadl.dev.java.net/.

[16] Philipp Haller. On the integration of the actor model into mainstream technologies. In *Proceedings of 2nd International Workshop on Programming based on Actors, Agents, and Decentralized Control, colocated with the SPLASH Conference*, Tucson, Arizona, October 2012.

[17] James Hendler, Tim Berners-Lee, and Eric Miller. Integrating Applications on the Semantic Web. *Journal of the Institute of Electrical Engineers of Japan*, 122(10):676–680, October 2002. In Japanese; English reprint available at http://www.w3.org/2002/07/swint.

[18] Ingrid Van Den Hoogen. Deutsch's Fallacies, 10 Years After. JAVA Developer's Journal, online publication at http://java.sys-con.com/node/38665, January 2004.

[19] Jacek Kopecký, Karthik Gomadam, and Tomas Vitvar. hRESTS: an HTML Microformat for Describing RESTful Web Services. In *Proceedings of the 2008 IEEE/WIC/ACM International Conference on Web Intelligence (WI-08)*, Sydney, Australia, December 2008.

[20] Jacek Kopecký, Carlos Pedrinaci, and Alistair Duke. RESTful Write-oriented API for Hyperdata in Custom RDF Knowledge Bases. In *Proceedings of the International Conference on Next Generation Web Service Practices (NWeSP)*, Salamanca, Spain, November 2011.

[21] Kin Lane. History of APIs. Available at http://history.apievangelist.com/, June 2013.

[22] Ole Lensmar. Is REST losing its flair – REST API Alternatives. ProgrammableWeb blog[13], December 2013.

[23] Maria Maleshkova, Carlos Pedrinaci, and John Domingue. Investigating Web APIs on the World Wide Web. In *Proceedings of he 8th IEEE European Conference on Web Services (ECOWS 2010)*, 2010. Available at http://oro.open.ac.uk/24320/.

[24] Bruce Jay Nelson. *Remote Procedure Call*. PhD thesis, Carnegie-Mellon University, May 1981. Published as CMU Technical Report CMU-CS-81-119, XEROX PARC Technical Report CSL-81-9.

[25] Eric Reis. The lean startup: How today's entrepreneurs use continuous innovation to create radically successful businesses, 2011.

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

[27] Gareth Rushgrove. Sorry, but the Flickr API Isn't REST. Blog entry available at http://www.morethanseven.net/2008/02/21/sorry-but-the-flickr-api-isnt-rest/, February 2008.

[28] SOAP Version 1.2 Part 1: Messaging Framework. Recommendation, W3C, June 2003. Available at http://www.w3.org/TR/2003/REC-soap12-part1-20030624/.

[29] 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 Berlin Heidelberg, 2011.

[30] Joshua Thijssen. What is HATEOAS and why is it important for my REST API? The RESTful CookBook, available online at http://restcookbook.com/Basics/hateoas/, 2012.

[31] Steve Vinoski. Concurrency with erlang. *IEEE Internet Computing*, 11(5):90–93, 2007.

[32] Dave Winer. XML-RPC Specification. Technical report, June 1999. Available at http://www.xmlrpc.com/spec.

[13] http://blog.programmableweb.com/2013/12/19/is-rest-losing-its-flair-rest-api-alternatives-2/