CORE
Metadata, citation and similar papers at core.ac.uk
Provided by ePublications

Technology
Arts Sciences
TH Köln

REST
Why it became the new norm on the web

Literature Review

Sebastian Faust
Bachelor Student of
Media Technology and Computer Science,
Technical University Cologne

Written while studying abroad in Kristiania University College
for the course Systems Development

Overseen by
Prof. Tor-Morten Grønli,
Kristiania University College

Oslo, Norway in September 2018

ABSTRACT

REST became the go to approach when it comes to large scale distributed systems on, or outside the World Wide Web. This paper aims to give a brief overview of what REST is and what its main draws and benefits are. Secondly, I will showcase the implementation of REST using HTTP and why this approach became as popular as it is today. Based on my research I concluded that REST's advantages in scalability, coupling, performance and its seamless integration with HTTP enabled it to rightfully overtake classic RPC based approaches.

1. INTRODUCTION

Software and its complexity have been rapidly growing over the last few decades. A solution for developing ever expanding and evolving systems was in dire need. A logical step in tackling this problem was the approach of "Separation of Concerns", which is the idea of splitting one system into smaller subsystems that can be developed and maintained by separate teams. With the birth of the World Wide Web, these systems and their teams did not even have to be in the same geographical location, but could instead be spread throughout the globe, working independently. With this approach, one big problem was solved, but another one arose: what is the best and most efficient way for these subsystems to interact with each other? One common approach to solve this problem was to have one system execute procedures on another system and then receive the result of that procedure remotely (RPC). This simple solution was the norm before 2000 and it worked beautifully. But with ever-expanding systems and ever-growing complexity, this solution revealed some inherent problems.

REST is an architectural style first introduced by Roy T. Fielding in his dissertation.[1] It is one solution to the problem of largescale distributed systems, that has rapidly evolved to be the norm since its publication in 2000. REST and its derived adjective "RESTful" have become buzzwords and the source of heated arguments ever since it started gaining popularity. The google searches of the term "REST" under the topic of "Computers & Electronics" have been steadily increasing since 2004 (Figure 1). The highly prestigious conference "Oracle Code One" has 37 scheduled events on or related to REST within its four-day runtime in 2018.[2] The "ProgrammableWeb" is the biggest library of web APIs. In 2017, 82% of their listed APIs were based on a RESTful architecture (Figure 2). The two biggest publishing houses in computer science, IEEE and ACM have collectively published ~2200 pieces of academic literature on and related to RESTful architecture.[3] [4]

Figure 1 Google Trends search of the term: "REST" in the category "Computers & Electronics" [5]

Clearly, REST is a highly relevant topic. This paper aims to explore the reasons why it became so popular, what exactly are its advantages, and how it can be implemented with HTTP. In section 2. I will explain what exactly a RESTful Architecture is. In section 3 I will review four highly relevant papers on the topic of REST. The first three focused on the advantages of REST and the last one on how to properly implement REST with HTTP to gain those advantages. In section 4 I will discuss the reviewed papers and in section 5 I will draw my conclusion on RESTful architecture, its advantages and its rise on the web.

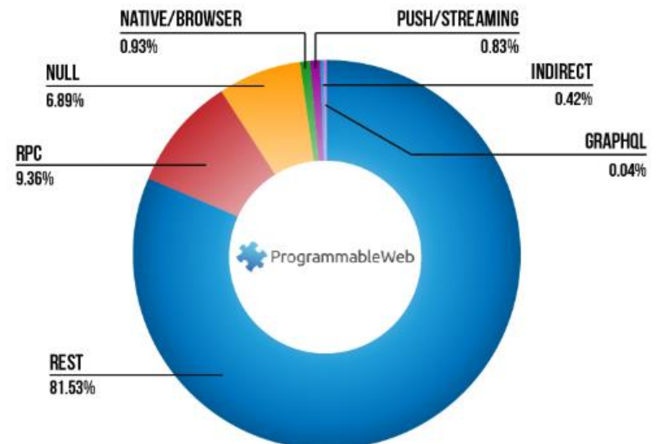

Figure 2 the percentages of API architectural styles for profiles in the "ProgrammableWeb" API directory [6]

[1] Fielding, "Architectural Styles and the Design of Network-Based Software Architectures."

[2] "Session Catalog | Oracle Code One 2018."

[3] IEEE, "IEEE Xplore Digital Library."

[4] ACM, "ACM Digital Library."

[5] Google LLC, "Google Trends - REST since 2004."

[6] Santos, "Which API Types and Architectural Styles Are Most Used?"

2. WHAT IS REST [7] [8]

The REST architectural style can be broken down into a list of seven constraints (rules) a given service must follow. When these constraints are fully embraced within the service, it is considered RESTful. A RESTful service has many advantages that will be showcased in section 3 and 4, here I will exclusively focus on the constraints themselves. One thing to keep in mind is that REST describes the interface over which two separate systems interact with each other and not how those two systems work internally.

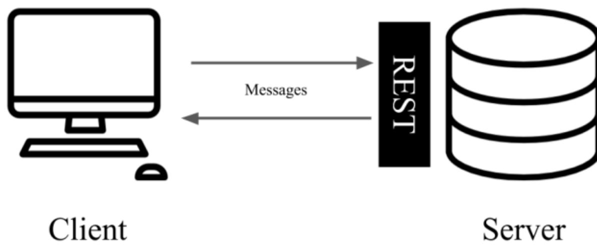

Figure 3 diagram showcasing where exactly RESTful architecture lies in a client-server module

Client–Server

Every REST based system is message based. It consists of two entities:

Client

The system sending requests.

Server

The system receiving those requests and processing them.

Statelessness

The Server should not save any state or session information. All information about the current interaction is saved on the client. Every message a client sends to the server needs to contain all the information necessary to process it and cannot rely on any previously sent messages.

Resource Based

REST is resource based. This means that every information a server provides must be modeled as a resource. But what exactly is a resource? A resource is any significant part of your system, that can be labeled with a noun. In the context of a calendar application for example, every individual day could be a resource. Every event a given user has added to their calendar could be a resource. The most important thing when designing a RESTful service is to think in terms of nouns instead of verbs. Modeling resources instead of procedures. Another thing to keep in mind when designing resources is, that the resources themselves are separate from their representations. The resource "user" for example, could be made available in multiple representations, such as JSON, XML or HTML.

Uniform Interface

This constraint aims to achieve one goal: One uniform service interface for all clients to communicate through. Every resource of a given service is addressable through a unique identifier (URI). Every resource a client receives from a service should include all the information the client needs to manipulate that resource. The only way to interact with a given resource should be through a fixed set of clearly defined "verbs", such as the set of HTTP verbs: GET, PUT, POST, DELETE.

Hypermedia as the Engine of Application State

This constraint is closely linked to the uniform interface constraint. The idea is to build up an API in a similar way as a web page, enabling a user to navigate through relevant resources with hyperlinks. To go back to the example of a calendar application, a given day-resource, could include links to all events that take place on that day. If this concept is fully embraced, a service should be usable by a client through only one entry point URI, from which the client can navigate through the API and find all relevant information or perform all relevant procedures.

Caching

The system should include a caching mechanism. That enables a client to request noncritical data with a lower frequency and thus lower the traffic between client and server significantly.

Layered System

A given service might have a multitude of layers that process incoming requests. These layers might have various responsibilities, such as security or cashing. But the important thing is, that this layered architecture stays hidden from the client.

3. LITERATURE REVIEW

In this section, I will review four highly relevant papers on the topic of REST. The section is structured into two parts, the first one focusing on what the exact advantages of RESTful APIs are and the second one exploring how to implement the REST architecture using HTTP.

Advantages of REST

In their Paper *"REST: An Alternative to RPC for Web Services Architecture"* [9] Xinyang Feng, Jianjing Shen, and Ying Fan explored the advantages of REST in comparison to the classic RPC approach. This paper was published in 2009 when the APIs based on RPC were still the norm. The paper starts by outlining both architecture styles. Here I will only showcase RPC because I have already extensively explained REST in section 2.

[7] Xinyang Feng, Jianjing Shen, and Ying Fan, "REST - An Alternative to RPC for Web Services Architecture."

[8] Fielding, "Architectural Styles and the Design of Network-Based Software Architectures."

[9] Xinyang Feng, Jianjing Shen, and Ying Fan, "REST - An Alternative to RPC for Web Services Architecture."

REST's main competitor: RPC

RPC is short for Remote Procedure Call. The idea behind RPC is, that a client can call procedures on a different machine to fulfill some sort of task. Machine A could, for example, call machine B with the command "getAllUsers()". Machine B would then execute some internal logic and then send a list of all users back to machine A. Protocols like SOAP are based on this idea of RPC with some additional constraints and features nested on top. These specifics however, where not further explored in the paper.

Feng, Shen, and Fan continue by comparing REST and RPC on six axes. Here I will only showcase four of them: Scalability, security, performance, and coupling. I will use the axes defined by Feng, Shen, and Fan to structure my review, exploring their view on each axis and subletting it with reviews of related papers by other authors.

Scalability

In RPC every service has its own unique interface. A client needs to know the specifics of that specific interface to interact with it. This is sufficient for small-scale or enclosed systems, but it does not work well on a large scale. Imagine a World Wide Web in which every website would have to be read by the browser differently or would require the download of a specific plugin to function properly. REST, with its uniform interface, does not face this problem. Most REST services use the HTTP verbs. To interact with any RESTful service implemented in this manner, a developer only has to know the 4 operations HTTP provides and never learn the specific operations of that domain area.

The statelessness of REST also provides a great advantage when it comes to scalability. A server never saves any session information and every message, the server receives, holds all the information the server needs to process it. Thus, if there are ever too many clients for the server to handle, more servers can just be added to balance out the incoming requests. Load balancing in RPC style systems is more complex and often leads to redundancy in saved data.

Security

In RPC style Systems that use HTTP to transfer commands over the web, every command is wrapped into an HTTP "envelope". The envelope passes through the firewall and the real intention of the command is unwrapped when it arrives in the system. If a REST system is based on HTTP, unwanted commands can be blocked on a firewall level. If a resource is GET only, a request to DELETE it will never go past the firewall itself. This has clear benefits from a security standpoint.

Performance

This is one of the main draws of REST. Because most REST services used on the web are based on HTTP, no unpacking of commands from envelopes or packing of commands into envelopes is required. REST also has an emphasis on cashing, which helps lower the messages exchanged between client and server. This difference is not only theoretical. Amazon.com hosts both REST and SOAP services and they state that REST services run six times faster than SOAP-based ones. This performance difference was also extensively explored and documented by Hatem Hamad, Motaz Saad, and Ramzi Abed in their journal article *"Performance Evaluation of RESTful Web Services for Mobile Devices"*[10]. In this article, they evaluated SOAP and REST services in both message size and computation speed. They used very simple services: one that adds all the floating-point numbers in a given array and sends its result back to the client and another one that appends all strings in a given array and sends that result back. They have implemented these services in both REST and SOAP. Their results clearly show the performance benefit of the RESTful services (Figure 4).

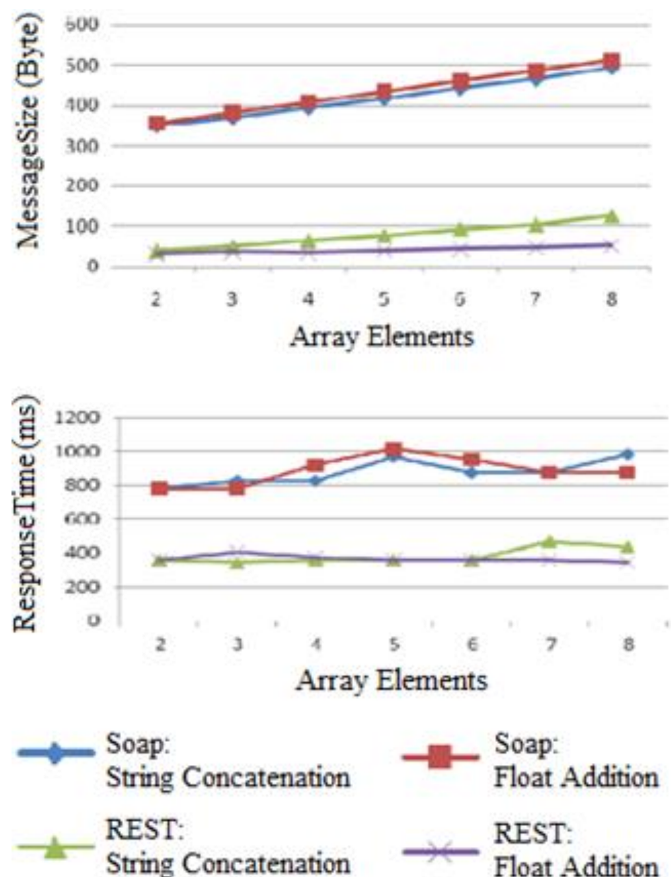

Figure 4 results of REST and SOAP performance comparison

[10] Hamad, Saad, and Abed, "Performance Evaluation of RESTful Web Services for Mobile Devices."

Coupling

Another axis Feng, Shen and Fan used to compare REST with RPC is "coupling". This part of their paper was primarily based on the journal article *"Demystifying RESTful Data Coupling"*[11] by Steve Vinoski. Instead of summarizing the abbreviation of Feng, Shen, and Fan I will showcase the work of their source. Vinoski's work is an in-depth analysis of the decoupling of server and client in RESTful architectures. Coupling, from a software architectural standpoint, is the amount of dependency one system has on another. With low coupling the systems are independent and changing one of the two is not relevant for the other. Therefore, low coupling is very beneficial for distributed systems. Vinoski argues, that the biggest source of coupling in distributed systems is specialized data-types. Specialized data-types are a set of rules in which a set of data is structured, that is only applicable for one specific problem or domain area. When two systems interact through one specialized data-type, they are coupled together by there shared understanding of how that data-type functions. If the data-type changes, all systems using that type must be adjusted. This type of coupling affects both REST and RPC, but REST has some inbuilt mechanics to alleviate it. Because REST supports multiple data-representations for each resource, a client is not bound to one specific format, but can instead choose which format is best for their application. Vinoski also adds, that hypermedia greatly decreases coupling between client and server. Because all possible operations that can be executed on a resource are showcased within the resource, no outside knowledge of the API is required to work with it.

REST on the Web

Those advantages do sound alluring. But how should we approach the development of a RESTful service when designing a HTTP-based system? In his journal article *"RESTful Web Services Development Checklist"*[12], Steve Vinoski showcases which features of HTTP can be used to fulfill the constraints listed in section 2. HTTP supports "content negotiation" which is a good way to implement multiple representations for a single resource. The HTTP-header has a field called "content-type". A client can put their preferred content type in this header field and the server can respond with a representation in that format or with a list of supported formats if the wanted format is not supported. HTTP has a list of well-defined verbs, that can be used to fulfill the uniform interface constraint (Table 1).

Verb	Definition
GET	Retrieve a resource in a chosen representation.
PUT	Overwrite a resource or create one if there is none to be overwritten.
POST	Can be used to perform virtually any action. In REST it is a common practice to use it to create a new resource in a collection.
DELETE	Delete a resource.
OPTIONS	Show the available operations for a given resource.

Table 1 the list of HTTP verbs and their respective definition [13]

One thing to keep in mind when using these verbs is that GET should always be "save", which means that no changes on the server should occur when it is executed. GET, PUT and DELETE should be "idempotent", which means that they can be executed multiple times without changing their effect. HTTP also has an inbuilt mechanism for cashing, that can be implemented very easily. The E-Tag field in the HTTP-header should contain a hash, that is changed every time the resource stored on the server is changed. If a client wants to GET a resource multiple times, they can send the E-Tag they received the last time they retrieved that specific resource. If the resource was not modified, the E-Tag is identical and the server sends an empty response with the status-code: "Not Modified". Resources in REST with HTTP should be addressable through hyperlinks (https://www.example.com/calander). The hypermedia constraint can be easily fulfilled by linking the relevant resources and their operations together through such hyperlinks.

4. DISCUSSION

In this section, I will critically discuss the points made in the papers that I have reviewed in section 3.

I fully agree with what Feng, Shen, and Fan said about the advantages of REST in terms of scalability. This advantage is undeniable. Using a load-balancer to distribute incoming requests to any number of RESTful services makes handling large-scale services easy in comparison to SOAP. This is one of the main reasons why REST became as popular as it is today.

[11] Vinoski, "Demystifying RESTful Data Coupling."

[12] Vinoski, "RESTful Web Services Development Checklist."

[13] Fielding, Irvine, and Gettys, "HTTP: Method Definitions."

Security, on the other hand, is not. Feng, Shen, and Fan only briefly went over this topic and did not do it justice in my opinion. REST messages can be encrypted when it is used with HTTPS and not with HTTP. The message will be encrypted in its entirety while traveling through the web on HTTPS. SOAP in combination with WS-Security[14] is more flexible. A developer can choose to encrypt only parts of a message or even encrypt a message in such a way that some part is readable by one party and other parts are readable by another. In short, REST's approach is more lightweight, easier to handle and often sufficient while SOAP holds more options but also requires a higher development effort.

Another related topic that was not explored by Feng, Shen, and Fan is authentication. Because every message has to be self-descriptive in REST, every message needs to carry the authentication information of a given user. In SOAP, this is handled through sessions. Authentication happens once, and critical authentication data only has to be sent between client and server once, which has clear benefits from a security standpoint.

The performance advantages of REST were described by Feng, Shen, and Fan and then proven by Hamad, Saad, and Abed. I have nothing more to add to this other than to say that this discrepancy in performance undoubtedly helped REST become the new standard on the web. I would have liked to add an exploration of cashing and how that REST principle effects performance in the long term, but I could not find any papers or studies related to this.

I can only agree with Vinoski's exploration of decoupling in RESTful services. Strong dependency between different components of software is a problem that all developers working on a larger system will face at some point during their career. And REST does help to alleviate this problem. REST's constraints force a developer to build a clearly defined interface that, at its best, is even self-explanatory. "REST-Chart"[15] is a module that embraces this self-explanatory nature to its fullest. Any REST API designed with the REST-Chart approach can be navigated by a generic client solely through hypermedia without any prior knowledge of the API. So far this is the peak of client-server decoupling and it is only possible because of REST and hypermedia.

In the second part of my review, I looked at the approach most commonly used to implement REST: HTTP. Roy T. Fielding, the designer of REST, also was one of the main contributors to the definition and specification of HTTP[16], so it is not surprising that REST and HTTP work well together. REST is built to fully embrace all the feature HTTP has to offer. For every constraint that requires an underlying technology, there is a corresponding HTTP feature (Table 2).

REST Constraint	HTTP Features
Client-Server	HTTP is the web protocol. It is inherently used for client-server connections
Statelessness	Requires no underlying technology
Resources	Multiple resource representations are enabled through content-type header field
Uniform Interface (URIs)	Hyperlinks
Uniform Interface (verbs)	HTTP verbs
Hypermedia	Hyperlinks
Cashing	E-Tag / Last-Modified header field
Layered System	Requires no underlying technology

Table 2 linking REST constraints to HTTP features

REST embraces all the aspects of the World Wide Web. Using a well-designed REST API is more like browsing a website than it is giving instructions to a remote computer. As Feng, Shen, and Fan put it:

"RESTful Web services are "in" the Web instead of just "on" the Web."[17]

HTTP is the most common way to approach the REST architectural style for a reason. They work perfectly together. Many large-scale systems have been implemented in this manner. A good example for this is "The Web of Things"[18], a promising web-framework that links IOT-devices to REST resources.

[14] OASIS, "OASIS Web Services Security (WSS) TC."

[15] Li and Chou, "Design and Describe REST API without Violating REST."

[16] Fielding, Irvine, and Gettys, "Hypertext Transfer Protocol -- HTTP/1.1."

[17] Xinyang Feng, Jianjing Shen, and Ying Fan, "REST - An Alternative to RPC for Web Services Architecture."

[18] Paganelli, Turchi, and Giuli, "A Web of Things Framework for RESTful Applications and Its Experimentation in a Smart City."

5. CONCLUSION

Is REST the long sought-after silver bullet of software engineering? [19] No, it is not. It is an architectural style mindfully designed to solve common problems in large-scale distributed systems, by using all the features of HTTP to its fullest extent. Many proponents of REST describe it as being "simple" or "easy", but with this however, I do not agree.[20] Designing a stateless system, that is fully resource based without any operations other than a fixed set of verbs is not easy, it is highly unintuitive for developers used to classic programming paradigms. And I am not the only one with this opinion. The sheer number of guides that explain what exactly REST is and what the constraint "really" means, speaks for itself. And most of those articles being incomplete or even contradictory, does not help this problem either.[21] [22] But overcoming REST's unintuitive nature yields a wide range of benefits: great scalability, sufficient security, great performance, and low coupling. All in all, there is a good reason REST became the new norm. And it is no coincidence that every major web-based company (Facebook, Netflix, Amazon, PayPal …) switched from a SOAP to a REST API within the last decade.

6. REFERENCES

ACM. "ACM Digital Library." Accessed September 20, 2018. https://dl.acm.org/.

Avraham, Shif Ben. "What Is REST — A Simple Explanation for Beginners, Part 1: Introduction." *Medium* (blog), September 5, 2017. https://medium.com/extend/what-is-rest-a-simple-explanation-for-beginners-part-1-introduction-b4a072f8740f.

Brooks, Frederick. "No Silver Bullet – Essence and Accident in Software Engineering," 1986, 16.

Fielding, R., UC Irvine, and J. Gettys. "Hypertext Transfer Protocol -- HTTP/1.1." Specification. Hypertext Transfer Protocol -- HTTP/1.1. Accessed September 19, 2018. https://www.w3.org/Protocols/rfc2616/rfc2616.html.

Fielding, Roy Thomas. "Architectural Styles and the Design of Network-Based Software Architectures," 2000, 90.

Fielding, Roy Thomas, UC Irvine, and J. Gettys. "HTTP/1.1: Method Definitions." Specification. Accessed September 20, 2018. https://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html.

Google LLC. "Google Trends - REST since 2004." Google Trends. Accessed September 15, 2018. https://trends.google.com/trends/explore?cat=5&date=all&q=REST.

Hamad, Hatem, Motaz Saad, and Ramzi Abed. "Performance Evaluation of RESTful Web Services for Mobile Devices." *International Arab Journal of E-Technology* 1, no. 3 (2010): 7.

IEEE. "IEEE Xplore Digital Library." Accessed September 20, 2018. https://ieeexplore.ieee.org/Xplore/home.jsp.

Li, Li, and Wu Chou. "Design and Describe REST API without Violating REST: A Petri Net Based Approach." In *2011 IEEE International Conference on Web Services*, 508–15. Washington, DC, USA: IEEE, 2011. https://doi.org/10.1109/ICWS.2011.54.

OASIS. "OASIS Web Services Security (WSS) TC." Open Standard. Accessed September 19, 2018. https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wss.

Paganelli, Federica, Stefano Turchi, and Dino Giuli. "A Web of Things Framework for RESTful Applications and Its Experimentation in a Smart City." *IEEE Systems Journal* 10, no. 4 (December 2016): 1412–23. https://doi.org/10.1109/JSYST.2014.2354835.

Rouse, Margaret. "What Is REST (REpresentational State Transfer)? - Definition from WhatIs.Com." SearchMicroservices, November 2017. https://searchmicroservices.techtarget.com/definition/REST-representational-state-transfer.

Santos, Wendell. "Which API Types and Architectural Styles Are Most Used?" ProgrammableWeb, November 26, 2017. https://www.programmableweb.com/news/which-api-types-and-architectural-styles-are-most-used/research/2017/11/26.

"Session Catalog | Oracle Code One 2018." Accessed September 15, 2018. https://oracle.rainfocus.com/widget/oracle/oow18/catalogcodeone18?search=rest.

Vinoski, Steve. "Demystifying RESTful Data Coupling." *IEEE Internet Computing* 12, no. 2 (March 2008): 87–90. https://doi.org/10.1109/MIC.2008.33.

———. "RESTful Web Services Development Checklist." *IEEE Internet Computing* 12, no. 6 (November 2008): 96–95. https://doi.org/10.1109/MIC.2008.130.

Xinyang Feng, Jianjing Shen, and Ying Fan. "REST - An Alternative to RPC for Web Services Architecture." In *2009 First International Conference on Future Information Networks*, 7–10. Beijing, China: IEEE, 2009. https://doi.org/10.1109/ICFIN.2009.5339611.

[19] Brooks, "No Silver Bullet – Essence and Accident in Software Engineering."

[20] Vinoski, "RESTful Web Services Development Checklist."

[21] Avraham, "What Is REST — A Simple Explanation for Beginners."

[22] Rouse, "What Is REST (REpresentational State Transfer)?"