# An Overview of REST
# by Vivin Suresh Paliath


## Abstract

REST, which stands for Representational State Transfer, is a type of software architecture designed for distributed systems. In this paper, I will examine REST as an architectural pattern, focusing on its genesis, rationale, uses, relation to other technologies, criticisms, and future direction.


## Overview of REST

REST was introduced and defined by Roy Fielding in his doctoral dissertation as an architectural style for distributed hypermedia systems[1] and was designed in parallel with HTTP/1.1. It arose out of a need for the Internet Engineering Task Force (IETF) to describe how the World Wide Web should work. As such, it describes an ideal model of the different interactions within a web-based application. During the late 90's and early 2000's, the World Wide Web was emerging as a software platform, rather than just a channel to serve media. As a result, there were different ways (which were incompatible and non-interoperable) to communicate with different resources. These architectures were often insular and proprietary (using Remote Procedure Calls) and often limited to one provider or technology. REST is an attempt to describe a generalized architectural-model that enforces scalability, generality, and interoperability as core concepts. It is important to note that REST is *not* a protocol[2]. Rather, it is an architectural style that is protocol-agnostic and therefore can be used with *any* protocol. The REST specification only mandates that there be a uniform interface to all resources. The most-common protocol that is used with REST is HTTP. REST resources can be accessed or modified using HTTP methods such as GET, POST, PUT, or DELETE.

The two primary components of a RESTful architecture are the client and the server. Clients send requests to servers and servers process the requests and return appropriate responses. Requests and responses essentially denote a representation of resources and present a way to transfer these representations. Furthermore, requests can effect a change in state of the application or resource and the response is a representation of that state (of the application or resource) and may also contain links to other, available state-changes. In his dissertation Fielding describes the following goals for REST:

- Scalability of component interactions.
- Generality of interfaces.
- Independent deployment of components.
- Intermediary components to reduce latency, enforce security, and encapsulate legacy systems.

The basic unit of data in REST is the *resource*. Resources can refer to anything, such as an image, a video, an order, a person, or even a bank transaction. Each *resource* in REST will have at least one identifier that uniquely identifies that resource. Each *resource* also has at least one *representation*. This representation can be in the form of JSON, XML, CSV, or even HTML.

REST enforces a client-server separation. This separation has numerous advantages, such as simplifying the design and implementation of components, reducing the semantic complexity of the interfaces between different components, and allowing the addition of intermediary layers (such as firewalls, proxies, or gateways) without having to change the interfaces between components. The client-server separation has the added advantage of enforcing loose-coupling between components, which leads to easier maintenance, improved scalability, and the ability to swap out one or more components without affecting others. In a sense, the interaction between components only exists in an abstract form and therefore hides the underlying implementation details. This is important due to the complexity inherent in large networks; components are frequently changed or upgraded and therefore it is important that individual components are able to communicate with each other in a general manner[3]. What this means is that as long as the interface remains consistent and adheres to REST principles, the software implementation or hardware behind a server or a client can change without causing any adverse effects. Such a design is useful on the World Wide Web, where standards and technologies change and evolve rather quickly. The idea of client-server separation is actually not new and has been around much longer than REST. However, REST mandates it as a core principle because it is a proven concept with numerous advantages.

Another important REST concept is the concept of statelessness. Statelessness is important due to the uncertain nature of communicating over an electronic network; connections can be dropped and data can be lost. REST addresses this issue by mandating that each message be a self-contained representation of state[4]. Hence, servers do not need to keep track of the conversational state for each client.

REST, as an architectural style, is one that focuses on the development of software systems over a long-term time-scale. As such, it aims to “promote software longevity and independent evolution”[5]. Hence, the emphasis is long-term software design versus short-term software design, and as stated by Fielding himself, “REST is intended for long-lived network-based applications that span multiple organizations”[5]. This idea is essentially described by the concept known as “Anarchic Scalability”[4]. Usually, software systems are created with the assumption that they are under the control of one entity, or at least under the control of separate entities who share a common goal. However, this assumption is not tenable, given the decentralized and distributed nature of the Internet. Any system created on the basis of such an assumption cannot run safely on the Internet. A system that *is* anarchically stable, on the other hand, can run safely on

the Internet. Furthermore, such a system is scalable since it does not make any assumptions as to the clients that interact with it; the system's interaction with the outside world is defined through an abstract interface.

Fielding's design of REST hearkens back to the simple concepts that made the world-wide web so successful, and applies these concepts towards application design. REST built on those existing ideas and concepts, which were shown to be popular and scalable. REST simply defines a few rules and capabilities so that instead of simply rendering pages or transferring documents, resource endpoints end up being functional and so can actually “do something”.

To see what REST can do, consider an example: Let's say we have an online retailer like Amazon. Assume that we have a person who is selling his or her products on this site. Let us also assume that this site provides a web-based user-interface for the customer. From this interface, the user can keep track of orders for their products and also finalize orders and ship them to their customers. If a particular item is running low, they can order additional items to ensure that those items remain in stock.

After a while, this online retailer realizes that they are getting feature requests from their customers to allow automation of certain tasks, like finalizing and shipping, and reordering items when they fall below a certain threshold. The retailer has also realized that people would like to access the site from their mobile phones as well. To satisfy their customers, they decide to embark on a project to implement these features. Now this can be reasonably easy, or tremendously difficult; the difference is in how they decided to implement their service. If they didn't adhere to REST principles, there would most probably be a tight coupling between the code that performs the business-logic, and the user-interface code. The business-logic may have some inbuilt assumptions that it is being accessed by a web-based user-interface. Instead of providing a generic interface into the service, they created an application that was designed to be used by via a web-based user-interface. Now when the developers for this retailer want to implement the new features, they realize that they have to make a lot of changes to the application code. Since there are a lot of changes to be made, it is entirely possible that they end up creating a lot of software-defects as well. Essentially, by not designing their application to have a generic and scalable interface, they locked themselves into a particular paradigm, which has now made it difficult for them to scale their software solution.

Now assume that they designed their application using REST principles (**Figure 1**). This would mean that their application has a uniform interface. The application itself makes no assumption as to the kinds of clients that access it. Therefore it remains generic enough to be accessed by any sort of client since all it exposes is a single, generic interface. It would now be trivial for developers to implement a mobile-client, and also create a Software Development Kit for the service, to enable customers to perform automated operations. Furthermore, all of

these new changes can be implemented without having to modify the original application's code. Using REST principles leads to a functional application with a generic interface that is extremely scalable, modular, maintainable, and one that can be easily used with other software components.

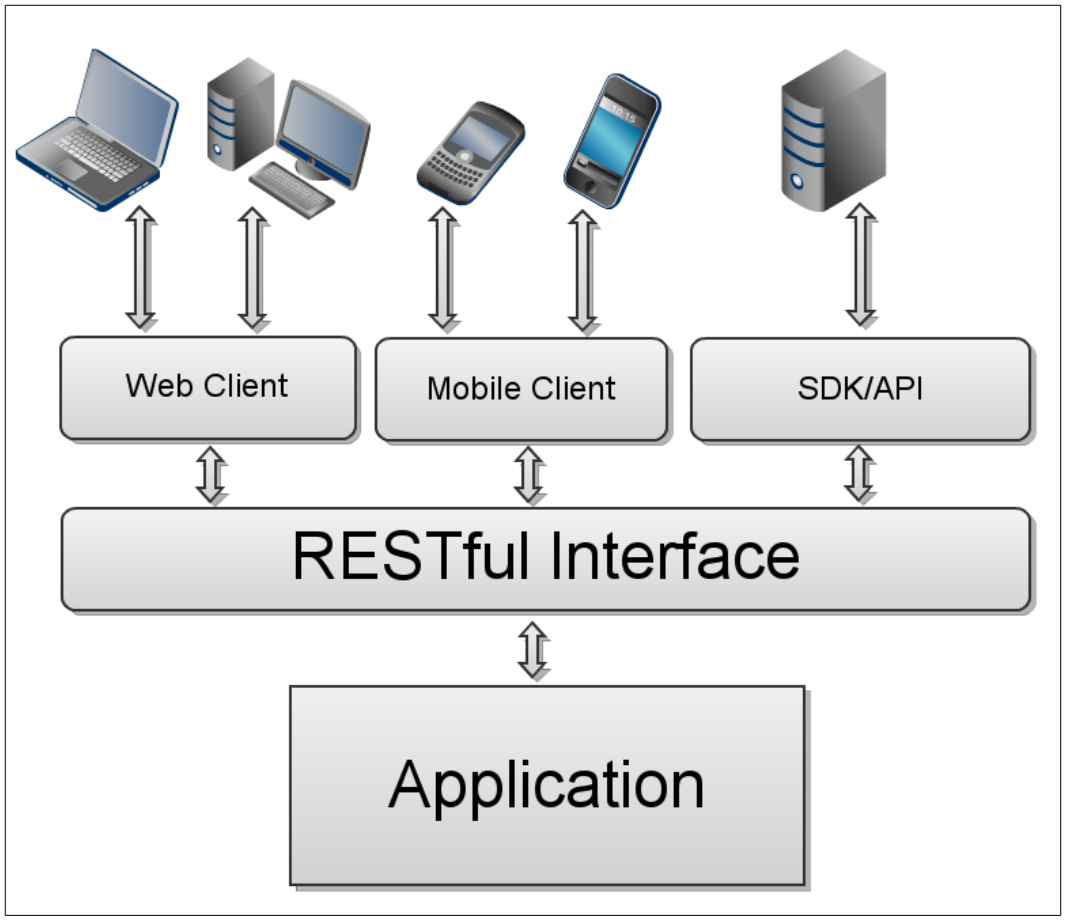


**Figure 1: Example of a RESTful architecture**

**REST as a basis for web-based applications**

Modern web-applications lend themselves quite easily to the REST architectural style. There are a few reasons for this. This first reason is that there is already an implicit client-server separation: the client is the user's browser and the server is the web-application itself. The second is that REST is mainly used through HTTP, and so a web-application will have no issues working with that protocol. Another reason is the stateless and generic nature of RESTful applications. Since a RESTful application does not maintain state, and only exposes and describes itself through a generic and abstract interface, it is theoretically possible to write multiple clients for one RESTful application. For example, one could even write a command-line application to communicate with a RESTful application.

However, the appeal of RESTful applications in the context of the web is that one can easily write a rich, web-based client for a RESTful web-application by simply

using HTML, Javascript, and CSS. This is because there is a clean separation between the consumer of the RESTful web-application (the client, or in this case the HTML/Javascript/CSS web-client) and the RESTful web-application itself (i.e., the server). Furthermore, the client can be easily updated to change the look and feel, and this has no impact on the web-application itself. Essentially, the server architecture can evolve and change independent of the client.

Creating a web-application using a REST architecture is also beneficial because the interface to the application is now implicitly public, uniform, and abstract. This means that the application can function as a platform, which means that it is now possible for your web-application to be easily consumed, or integrated into other systems. For example, an airline may decide to architect its reservation system using a REST architectural style. With this system in place, it is trivial to write multiple clients to service different types of users; a web-based client or a mobile-client can be useful for customers, whereas an internal, proprietary client can be useful for airline employees. Without a REST-based architecture, the system may have been tightly coupled to a legacy interface and implementation. This would have made it difficult to adapt the system to changing technologies. In some cases, the system is so tightly coupled to a specific implementation that the only option would be a rewrite. It seems pretty obvious then, that a system that has been designed using REST-principles would require only minimal or no changes to have it work with a new interface. This is because only the clients need to be updated to match changing technologies or standards, and not the system itself. In an environment like the Web, where technologies and standards can change with the matter of a year or two, we can easily see how a REST-based architecture is advantageous.

**REST and Service Oriented Architecture**

Service-oriented architecture, or SOA, is a way to architect systems such that application functionality is exposed through interoperable services. SOA allows applications to easily communicate and exchange data with each other. A service-oriented architecture enforces loose-coupling and separation of concern. Application functionality is exposed using an abstract interface that exposes different services.

On the face of it, REST and SOA seem to be similar in the sense that they aspire to the same goals: highly scalable and interoperable systems that enforce separation-of-concern, and are exposed via with generic interfaces. However, the similarity stops there because REST and SOA go about achieving these goals in two entirely-different ways[6].

The primary unit of data in REST, as mentioned before, is the *resource*. There is no such concept in an application built using SOA. Instead, a SOA-based application exposes *services*. A *service* is an abstract notion of a certain task that can be performed by the application. This task encapsulates some sort of

functionality that can is provided by the application. So a SOA-based application essentially exposes a set of different functionalities. In contrast, a *resource* maps to a singular entity, and there is no concept of providing specific “functionality”; the only functionality that is provided by REST is through the standardized HTTP operations of GET, PUT, POST, and DELETE. In layman's terms, SOA deals with *verbs* (services) whereas REST deals with *nouns* (resources).

Another difference between REST and SOA deals with the way business-logic will be implemented. In SOA, the business logic is an essential part of the functionality that is provided by the application, and as such resides within the services themselves. In an architecture based on REST principles (resource-oriented architecture, or ROA), the business logic will not be an essential part of the resource, and it shouldn't since resources are essentially just entities, and as such shouldn't contain any business logic since that would violate separation-of-concern. Instead, an ROA created using REST principles will implement business logic using triggers[7][8]. Essentially REST abstracts entities out to *resources*, and a change to the state of one entity needs to be propagated to other entities within the system so that the entire system is in a consistent state. For example, imagine that a car insurance provider has exposed the customer as a resource. Assume that a change is made to the customer's age so that the customer is now over the age of 25. This usually means that the customer's insurance rates will go down. If the application didn't implement a trigger on the customer resource, then the customer's rate will not be change. Hence, in an application that implements an ROA using REST, there will be a trigger on the customer resource that automatically goes and updates an “insurance plan” resource.

Compared to SOA, REST is more of a resource-oriented architecture, and so there is nothing called a Service-Oriented REST Architecture, because those are two different and incompatible concepts[6]. In simple terms, SOA places more importance on the functionality offered by the system, compared to the components or entities that make up the system. REST, on the other hand, does the reverse by placing more importance on the components that make up the system (and the relationships between them) compared to the functionality offered by the system. In essence, the functionality is implicitly defined through the explicit definition of the relationships between different components. Therefore, in a SOA-based application that exposes an API, programmers usually call services. While those services may provide a reference to components, changes to those components are not effected directly: instead the component will be passed into another service and that service will take care of updating the state of the component. Contrast that with an application that implements ROA through REST. In an API exposed by such an application, programmers will work directly with the entities (or more accurately, their abstract representations via resources) while the application will take care of maintaining the state of the application through triggers.

While both SOA and REST have evolved from the concept of “distributed

objects", SOA focuses on the design of the application with the distribution aspect being a secondary concern whereas REST, by contrast, focuses mainly on ensuring that distributed systems and objects can scale and perform well together[9]. Both REST and SOA are acceptable ways of architecting a system, and while both methods have the same end-goals in mind (scalability, separation-of-concern, and interoperability), the semantics of their approach to realizing these end-goals are different.

**REST and SOAP**

There have been numerous comparisons between REST and SOAP (Simple Object Access Protocol), but a lot of these comparisons are based on the flawed assumption that REST is a protocol; it is not. REST uses the HTTP protocol predominantly, but HTTP is not mandatory; REST can use any other protocol. SOAP, on the other hand, evolved from a desire to make remote-procedure calls accessible over the Internet. SOAP uses HTTP as a transport and uses URIs to describe endpoints, but other than that, SOAP has its own XML protocol that sits on top of HTTP (or sometimes TCP/IP). SOAP's XML and RPC heritage comes from the fact that it is a successor to XML-RPC.

Another major difference between SOAP and XML is that SOAP is a rather heavy protocol because it describes functions and types of data. In contrast, REST is rather lightweight because it can use HTTP methods to perform operations.

The primary difference between REST and SOAP is that REST has multiple, unique URI's that map to a resource, whereas SOAP defines a single, heavy endpoint that offers its own namespace. There is no way to uniquely identify a single resource in SOAP through URIs.

REST is also more suited to being implemented with ROA, whereas SOAP is more suited to being implemented with SOA. This is because we basically want service calls (which are procedures) to be exposed over the Internet, and SOAP's XML-RPC heritage is a perfect fit for that.

REST-based API's and SOAP API's are equally valid ways of exposing functionality and usually a service will offer both. However, SOAP has wider support in the way of languages and frameworks that support it.

**Criticisms of REST**

One of the main criticisms of REST is that there is a lack of frameworks that enforce a REST-style of architecture or even its basic design-principle. Since REST is not a protocol or a specification, there is not an accepted set of compliance tests that can determine if a system is actually "RESTful"[10].

Criticisms of REST have also arisen due to the misunderstanding of Fielding's

original concept. This has led to the improper conflation of REST with the idea of a "REST protocol", and has also led to improper definitions of what a RESTful API or service actually is (on an interesting note, perhaps the only example of a truly RESTful service is the world-wide web itself)[11].

Since REST is intended to be an architectural-style that is intended to be generic enough, scalable over a long time-period, and span multiple organizations, it imposes very little other than the simple definition of what a RESTful architecture is. An expected consequence of REST is that it is not completely efficient due to its generality. For example, a truly RESTful API is not expected to have a published specification. Rather, they need to be "discovered" by accessing the root *resource*. In essence, the response from a truly RESTful API behaves just like a hypertext document; it provides data for the current request and links to other, related data. The client is expected to follow the links to other data to retrieve the relevant information. For example, suppose a client accessed a *resource* at *http://example.org/users*, the server should not respond with a list of representations of *user* objects. Rather, it should respond with a list of links to individual *user* objects, for example *http://example.org/user/1*, *http://example.org/user/2* and so on. Furthermore, the media type of the response conveys the semantics of what needs to be displayed. Also, each state-representation provides links to other, available state transitions. The best way to visualize this is to see how a browser works: a browser has no idea whether it is rendering a news site or a banking site; all it knows is how to render content based on the media type. The "state changes" are performed by users when they click a link to read a different news article, or to transfer money from the account. To quote Roy Fielding:

*"A REST API should be entered with no prior knowledge beyond the initial URI (bookmark) and a set of standardized media types that are appropriate for the intended audience (i.e., expected to be understood by any client that might use the API). From that point on, all application state transitions must be driven by client selection of server-provided choices that are present in the received representations or implied by the user's manipulation of those representations."*[5]

The end result of such an API is that all interactions are extremely "chatty". That is, a client may need to make multiple requests to access a piece of data. In this way, interaction with a RESTful service or API is "learned" rather than described. The problem with a chatty API is that it is somewhat inefficient. Information that could be transmitted in a single request in instead spread out over multiple requests.

A concern with the "learned" aspect of a REST API might be that it would be difficult to code against a "learned" API, since there are no explicit definitions for other resources, other than the primary endpoint. However, this is not necessarily so. For example, assume you have an endpoint at http://example.org/user/11,

which identifies “User 11”. When you access that URL with GET, you are able to retrieve information for that user. But if you access that URL with POST and some data, then you are able to update information for that user. Performing that update may require other state transitions to be made by the calling code, and that information can be conveyed in the media-type definition and documentation for the response returned by a POST to that URL. The media-type may contain a link to the next-available state-transition. There could also be a parameter that says that whoever consumes this data, should automatically transition to that available media-type. Basically, the state representation that comes back can contain all required information in-band, that is necessary for an automated consumer of the service, to decide what to do next.

Another often-heard criticism of REST is that there is no standardized representation of the state returned by a REST call. However, again, this criticism is unwarranted because there already is a standardized representation, and that representation is hypermedia. The documentation or description of the semantics of the hypermedia is done through documenting media-types and associated hypermedia controls (such as links). With this in mind, perhaps a legitimate criticism of a truly RESTful approach might be that it is “too hard” or “overkill”[11]. This is because it might be too much work for API authors to define and support custom media-types as well as to transmit the complex modeled-relationships in-band.

**Future of REST**

Since REST was first described by Fielding almost a decade ago, it has gained widespread acceptance. However, this acceptance has been fraught with inconsistencies and confusion as to what REST truly is, which is unusual for a concept that is based on simple concepts. This may be attributed to a lack of concrete examples from Fielding as to what constitutes a truly RESTful architecture, as opposed to just an abstract description of the concept. Although, as mentioned before, one could argue that we already have an example of a concrete RESTful architecture in the World Wide Web.

The future direction for REST would first require a more concerted effort to convey what REST actually is, and how one can actually get to a truly RESTful architecture. One example of such an effort is the “Richardson Maturity Model” of REST, developed by Leonard Richardson[12]. The model describes different levels of progressive enhancements that one can take to make an application truly RESTful. These levels are:

- Level 0: The Swamp of POX
- Level 1: Resources
- Level 2: HTTP Verbs
- Level 3: Hypermedia Controls

It is important to note that while each level is a stepping stone, it is not until Level 3 (Hypermedia Controls) that one achieves true RESTfulness.

The Richardson Maturity Model is interesting in another way because it seems to parallel the general understanding of REST over the years since Fielding's dissertation. Early API's that claimed to be RESTful seemed to be at around Level 0 (The Swamp of POX). Calls to such services usually returned XML (hence the acronym POX for Plain Old XML). There are also modern versions of such Level 0 APIs, which return JSON instead. In these Level 0 API's there is no notion of a "resource". Instead, these API endpoints are simply abstractions of remote-procedure calls for applications built using SOA.

Level 1 (Resources) is an enhancement over Level 0, because it explicitly identifies resources. There are numerous APIs at this level that claim to be RESTful and the basis for that claim is that they use "resources". Unfortunately simply having URLs that expose "resources" is not enough to make an application RESTful. This is also the reason for the idea that by simply having an API that has RESTful URL's an application is automatically RESTful. API's or applications that are at Level 1 may be exposing their entities via resources, but the application as a whole is usually still built using SOA and the resources are simply abstractions to service calls or remote-procedure calls.

Level 2 (HTTP Verbs) seems to be the level of REST where we are at today. Everyone seems to have realized that to at least start being truly RESTful, the semantics of HTTP requests methods need to be honored. Hence, requests for data are sent using GET, the adding of new data is sent using PUT, the modification of existing data is performed using POST (or the less-common PATCH), and the deletion of existing data is performed using DELETE. It is important to note that this does *not* mean that REST is inextricably linked to HTTP; it is just that the most commonly-used protocol for REST today, is HTTP. REST can be used with any protocol as long as that protocol presents a general interface to each resource. In light of this, a better name for this level would probably be "Uniformity and Generality of Interfaces" or "Uniform Verbs".

Level 3 (Hypermedia Controls) is the level at which one can claim to have a truly RESTful architecture. At this level, the semantics of the state is conveyed using media-types. Level 3 is usually expressed through the acronym HATEOAS, which stands for Hypermedia As The Engine Of Application State. Options for transitions into different states are provided using hypermedia controls like links. Level 3 is the future direction of REST and there is a push towards developing API's, frameworks, and applications in this manner. For example, the Spring application-development framework already has a module called "Spring HATEOAS" which makes it easy to create REST representations that follow the HATEOAS principle[13].

HATEOAS is not a new concept; it was originally mentioned within Fielding's

thesis. However, it would seem that it is only now, over a decade later, that we are really beginning to understand what Fielding meant, when he said that representation of state should be described using hypermedia. Hence the future direction of REST doesn't lie with REST itself, but rather, the software-engineering community's understanding of what REST actually is, and how best to implement those principles. Essentially, REST has a future as long as the world-wide web exists, since REST itself has been built on those same principles that has made the web so successful.

**Conclusion**

REST is an important and powerful architectural-style that aids the design of long-term, robust, and maintainable web-applications. While REST is a simple concept, there has been confusion surrounding its definition and application. Some of this confusion is related to a lack of knowledge about the trade-offs that REST makes, in favor of a generalized and scalable architecture. In addition, since its inception there has been a desire to jump on the "REST bandwagon", which has also led to confusion surrounding REST and what it actually means. This is ironic, because REST describes a very simple architecture with very few constraints. As with all software-design principles, architectural styles, and patterns, one needs to evaluate and understand what REST provides to see if it is an applicable solution to one's problem.

**References**


[1] R. Fielding, "Architectural Styles and the Design of Network-based Software Architectures," University of California, Irvine, Irvine, CA, 2000.

[2] R. Fielding, "No REST in CMIS," *Untangled musings of Roy T. Fielding*. 29-Sep-2008.

[3] M. zur Muehlen, J. Nickerson V., and K. Swenson D., "Developing Web Services Choreography Standards: the Case of REST vs. SOAP," *Decision Support Systems - Special Issue: Web services and process management*, vol. 40, no. 1, pp. 9–29, Jul. 2005.

[4] R. Fielding and R. Taylor, "Principled Design of the Modern Web Architecture," *ACM Transactions on Internet Technology*, vol. 2, no. 2, pp. 115–150, May 2002.

[5] R. Fielding, "Rest APIs must be hypertext-driven," *Untangled musings of Roy T. Fielding*. 20-Oct-2008.

[6] D. Nene, "Service Oriented REST Architecture Is an Oxymoron," */var/log/mind*. 01-Oct-2009.

[7] D. Nene, "REST Is the DBMS of the Internet," */var/log/mind*. 09-Jun-2009.

[8] D. Nene, "Design Characteristics of REST/Resource Oriented Server Frameworks and Clients," */var/log/mind*. 10-Jun-2009.

[9] S. Vinoski, "REST Eye for the SOA Guy," *Internet Computing, IEEE*, vol. 11, no. 1, pp. 82–84, Feb. 2007.

[10] B. Henry C., "RESTFul Services - Applying the REST Architectural

Style," Regis University, 2011.
[11] M. Bleigh, "REST isn't what you this it is, and that's OK," *Intridea Blog: Technology, Design, Business*. 29-Apr-2010.
[12] M. Fowler, "Richardson Maturity Model (steps toward the glory of REST)," *Martin Fowler*. 18-Mar-2010.
[13] O. Gierke, *Spring HATEOAS*. Springsource, 2012.