REST

REST (**Representational state transfer**) is a software architectural style that was created to guide the design and development of the architecture for the World Wide Web. REST defines a set of constraints for how the architecture of a distributed, Internet-scale hypermedia system, such as the Web, should behave. The REST architectural style emphasises uniform interfaces, independent

deployment of components, the scalability of interactions between them, and creating a layered architecture to promote caching to reduce user-perceived latency, enforce security, and encapsulate legacy systems.[1]

REST has been employed throughout the software industry to create stateless, reliable Web-based applications. An application that obeys the REST constraints may be informally described as *RESTful*, although this term is more commonly associated with the design of HTTP-based APIs and what are widely considered best practices regarding the

"verbs" (HTTP methods) a resource responds to while having little to do with REST as originally formulated—and is often even at odds with the concept.[2]

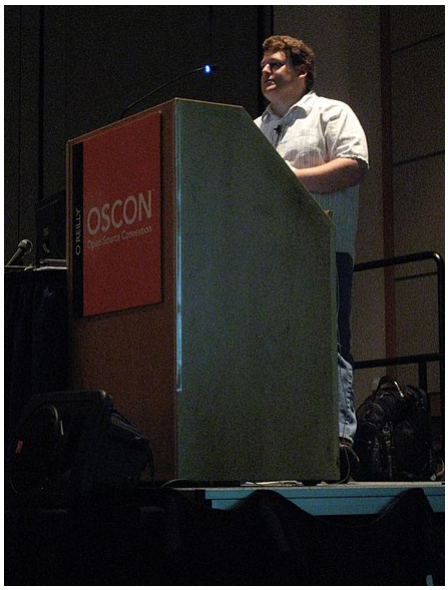
Principle

The term *representational state transfer* was introduced and defined in 2000 by computer scientist Roy Fielding in his doctoral dissertation. It means that a server will respond with the representation of a resource (today, it will most often be an HTML, XML or JSON document) and that resource will contain hypermedia links that can be followed to make the state of

the system change. Any such request will in turn receive the representation of a resource, and so on.

An important consequence is that the only identifier that needs to be known is the identifier of the first resource requested, and all other identifiers will be discovered. This means that those identifiers can change without the need to inform the client beforehand and that there can be only loose coupling between client and server.

History



Roy Fielding speaking at OSCON 2008

The Web began to enter everyday use in 1993–1994, when websites for general use started to become available.[3] At the time, there was only a fragmented description of the Web's architecture, and there was pressure in the industry to agree on some standard for the Web interface protocols. For instance, several

experimental extensions had been added to the communication protocol (HTTP) to support proxies, and more extensions were being proposed, but there was a need for a formal Web architecture with which to evaluate the impact of these changes.[4]

The W3C and IETF working groups together started work on creating formal descriptions of the Web's three primary standards: URI, HTTP, and HTML. Roy Fielding was involved in the creation of these standards (specifically HTTP 1.0 and 1.1, and URI), and during the next six years he created the REST architectural style, testing its constraints on the Web's

protocol standards and using it as a means to define architectural improvements — and to identify architectural mismatches. Fielding defined REST in his 2000 PhD dissertation "Architectural Styles and the Design of Network-based Software Architectures"[1][5] at UC Irvine.

To create the REST architectural style, Fielding identified the requirements that apply when creating a world-wide network-based application, such as the need for a low entry barrier to enable global adoption. He also surveyed many existing architectural styles for network-based

applications, identifying which features are shared with other styles, such as caching and client–server features, and those which are unique to REST, such as the concept of resources. Fielding was trying to both categorise the existing architecture of the current implementation and identify which aspects should be considered central to the behavioural and performance requirements of the Web.

By their nature, architectural styles are independent of any specific implementation, and while REST was created as part of the development of the Web standards, the implementation of the

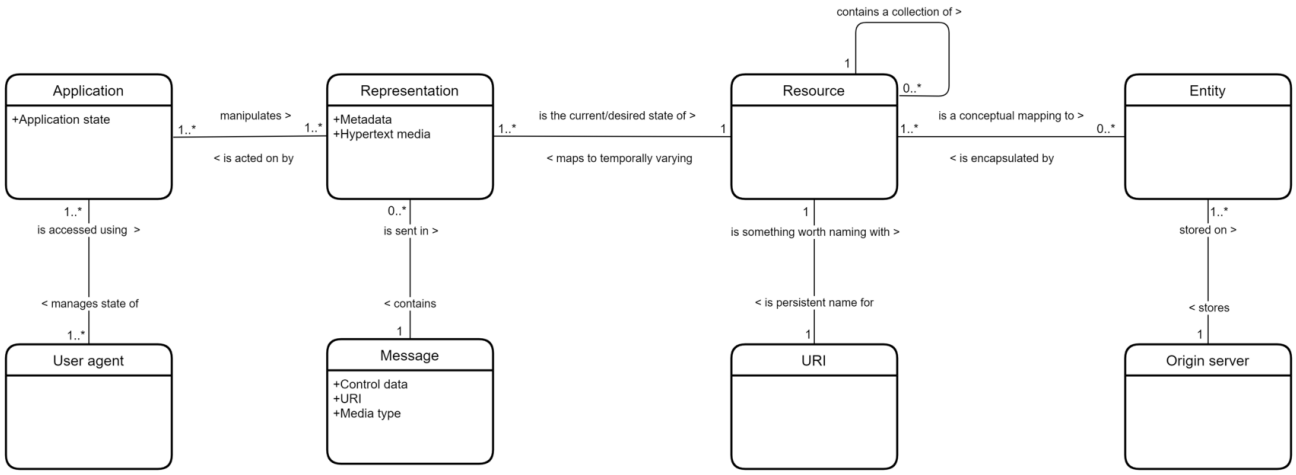
Web does not obey every constraint in the REST architectural style. Mismatches can occur due to ignorance or oversight, but the existence of the REST architectural style means that they can be identified before they become standardised. For example, Fielding identified the embedding of session information in URIs as a violation of the constraints of REST which can negatively affect shared caching and server scalability. HTTP cookies also violated REST constraints because they can become out of sync with the browser's application state, making them unreliable; they also contain opaque data that can be a concern for privacy and security.

Architectural properties

The REST architectural style is designed for network-based applications, specifically client-server applications. But more than that, it is designed for Internet-scale usage, so the coupling between the *user agent* (client) and the *origin server* must be as loose as possible to facilitate large-scale adoption.

The strong decoupling of client and server together with the text-based transfer of information using a uniform addressing protocol provided the basis for meeting the requirements of the Web: robustness

(anarchic scalability), independent deployment of components, large-grain data transfer, and a low entry-barrier for content readers, content authors and developers alike.



An entity-relationship model of the concepts expressed in the REST architectural style

The constraints of the REST architectural style affect the following architectural properties:[1][6]

- Performance in component interactions, which can be the dominant factor in user-perceived performance and network efficiency;[7]
- Scalability allowing the support of large numbers of components and interactions among components;
- Simplicity of a uniform interface;
- Modifiability of components to meet changing needs (even while the application is running);
- Visibility of communication between components by service agents;
- Portability of components by moving program code with the data;

- Reliability in the resistance to failure at the system level in the presence of failures within components, connectors, or data.[7]

Architectural constraints

The REST architectural style defines six guiding constraints.[6][8] When these constraints are applied to the system architecture, it gains desirable non-functional properties, such as performance, scalability, simplicity, modifiability, visibility, portability, and reliability.[1]

The formal REST constraints are as follows:

Client–server architecture

The client-server design pattern enforces the principle of separation of concerns: separating the user interface concerns from the data storage concerns. Portability of the user interface is thus improved. In the case of the Web, a plethora of web browsers have been developed for most platforms without the need for knowledge of any server implementations. Separation also simplifies the server components,

improving scalability, but more importantly it allows components to evolve independently (anarchic scalability), which is necessary in an Internet-scale environment that involves multiple organisational domains.

Statelessness

In computing, a stateless protocol is a communications protocol in which no session information is retained by the receiver, usually a server. Relevant session data is sent to the receiver by the client in such a way that every packet of information transferred can be understood

in isolation, without context information from previous packets in the session. This property of stateless protocols makes them ideal in high volume applications, increasing performance by removing server load caused by retention of session information.

Cacheability

As on the World Wide Web, clients and intermediaries can cache responses. Responses must, implicitly or explicitly, define themselves as either cacheable or non-cacheable to prevent clients from providing stale or inappropriate data in

response to further requests. Well-managed caching partially or completely eliminates some client–server interactions, further improving scalability and performance. The cache can be performed at the client machine in memory or browser cache storage. Additionally cache can be stored in a Content Delivery Network (CDN).

Layered system

A client cannot ordinarily tell whether it is connected directly to the end server or to an intermediary along the way. If a proxy or load balancer is placed between the client

and server, it won't affect their communications, and there won't be a need to update the client or server code. Intermediary servers can improve system scalability by enabling load balancing and by providing shared caches. Also, security can be added as a layer on top of the web services, separating business logic from security logic.[9] Adding security as a separate layer enforces security policies. Finally, intermediary servers can call multiple other servers to generate a response to the client.

Code on demand (optional)

Servers can temporarily extend or customize the functionality of a client by transferring executable code: for example, compiled components such as Java applets, or client-side scripts such as JavaScript.

Uniform interface

The uniform interface constraint is fundamental to the design of any RESTful system.[1] It simplifies and decouples the architecture, which enables each part to

evolve independently. The four constraints for this uniform interface are:

- Resource identification in requests: Individual resources are identified in requests, for example using URIs in RESTful Web services. The resources themselves are conceptually separate from the representations that are returned to the client. For example, the server could send data from its database as HTML, XML or as JSON—none of which are the server's internal representation.
- Resource manipulation through representations: When a client holds a

representation of a resource, including any metadata attached, it has enough information to modify or delete the resource's state.

- Self-descriptive messages: Each message includes enough information to describe how to process the message. For example, which parser to invoke can be specified by a media type.[1]
- Hypermedia as the engine of application state (HATEOAS) - Having accessed an initial URI for the REST application—analogous to a human Web user accessing the home page of a website—

a REST client should then be able to use server-provided links dynamically to discover all the available resources it needs. As access proceeds, the server responds with text that includes hyperlinks to other resources that are currently available. There is no need for the client to be hard-coded with information regarding the structure of the server.[10]

Classification models

Several models have been developed to help classify REST APIs according to their

adherence to various principles of REST design, such as

- the Richardson Maturity Model
- the Classification of HTTP-based APIs[11]
- the W S^3 maturity model[12]

Applied to web services

Web service APIs that adhere to the REST architectural constraints are called RESTful APIs.[13] HTTP-based RESTful APIs are defined with the following aspects:[14]

- the resource identifier (URI) of one or several resources used as starting points, sometimes called endpoints or entry points
- the encoding of all possible resource representations (which will include representation of the data and of the hypermedia links for state transitions)
- the possible state transitions and where they can occur

Unlike SOAP-based web services, there is no "official" standard for RESTful web APIs. This is because REST is an architectural style, while SOAP is a protocol. REST is not a standard in itself,

but RESTful implementations make use of standards, such as HTTP, URI, JSON, and XML. Many developers describe their APIs as being RESTful, even though these APIs do not fulfill all of the architectural constraints described above (especially the uniform interface constraint).[2] Most APIs claiming to be RESTful actually only satisfy the level 2 of the Richardson Maturity Model.

See also

- Clean URL – URL intended to improve the usability of a website

- Content delivery network – Layer in the internet ecosystem addressing bottlenecks
- Domain Application Protocol (DAP)
- List of URI schemes – Namespace identifier assigned by IANA
- Microservices – Collection of loosely coupled services used to build computer applications
- Overview of RESTful API Description Languages
- Resource-oriented architecture (ROA)
- Resource-oriented computing (ROC)
- Service-oriented architecture (SOA)

- Web-oriented architecture (WOA)

References

1. *Fielding, Roy Thomas (2000). "Chapter 5: Representational State Transfer (REST)" (http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm) . Architectural Styles and the Design of Network-based Software Architectures (Ph.D.). University of California, Irvine. Archived (https://web.archive.org/web/20210513160155/https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm) from the original on 2021-05-13. Retrieved 2004-08-17.*

2. *Fielding, Roy T. (2008-10-20). "REST APIs must be hypertext driven" (http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven) . roy.gbiv.com. Archived (https://web.archive.org/web/20100318060707/http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven) from the original on 2010-03-18. Retrieved 2016-07-06.*
3. *Couldry, Nick (2012). Media, Society, World: Social Theory and Digital Media Practice (https://books.google.com/books?id=AcHvP9trbkAC&pg=PA2) . London: Polity Press. p. 2. ISBN 9780745639208.*

4. *Fielding, Roy Thomas (2000). "Chapter 6: Experience and Evaluation" (https://www.ics.uci.edu/~fielding/pubs/dissertation/evaluation.htm) . Architectural Styles and the Design of Network-based Software Architectures (Ph.D.). University of California, Irvine. Archived (https://web.archive.org/web/20230326022001/https://www.ics.uci.edu/~fielding/pubs/dissertation/evaluation.htm) from the original on 2023-03-26. Retrieved 2023-06-21.*

5. *"Fielding discussing the definition of the REST term" (https://web.archive.org/web/20151105014201/https://groups.yahoo.com/neo/groups/rest-discuss/conversations/topics/6735) . groups.yahoo.com. Archived from the original (https://groups.yahoo.com/neo/groups/rest-discuss/conversations/topics/6735) on November 5, 2015. Retrieved 2017-08-08.*
6. *Erl, Thomas; Carlyle, Benjamin; Pautasso, Cesare; Balasubramanian, Raj (2012). "5.1". SOA with REST: Principles, Patterns & Constraints for Building Enterprise Solutions with REST. Upper Saddle River, New Jersey: Prentice Hall. ISBN 978-0-13-701251-0.*

7. *Fielding, Roy Thomas (2000). "Chapter 2: Network-based Application Architectures" (http://www.ics.uci.edu/~fielding/pubs/dissertation/net_app_arch.htm) . Architectural Styles and the Design of Network-based Software Architectures (Ph.D.). University of California, Irvine. Archived (https://web.archive.org/web/20141216114322/http://www.ics.uci.edu/~fielding/pubs/dissertation/net_app_arch.htm) from the original on 2014-12-16. Retrieved 2014-04-12.*
8. *Richardson, Leonard; Ruby, Sam (2007). RESTful Web Services (https://archive.org/details/restfulwebservic00rich_0) . Sebastopol, California: O'Reilly Media. ISBN 978-0-596-52926-0.*

9. *Lange, Kenneth (2016). The Little Book on REST Services (https://www.kennethlange.com/the-little-book-on-rest-services/) . Copenhagen. p. 19. Archived (https://web.archive.org/web/20190818061914/https://www.kennethlange.com/the-little-book-on-rest-services/) from the original on 18 August 2019. Retrieved 18 August 2019.*
10. *Gupta, Lokesh (2 June 2018). "REST HATEOAS" (http://restfulapi.net/hateoas/) . REST API Tutorial. RESTfulAPI.net. Archived (https://web.archive.org/web/20190407073345/https://restfulapi.net/hateoas/) from the original on 7 April 2019. Retrieved March 10, 2019.*

11. *"Classification of HTTP APIs" (http://algermissen.io/classification_of_http_apis.html) . algermissen.io. Archived (https://web.archive.org/web/20230129022641/http://algermissen.io/classification_of_http_apis.html) from the original on 2023-01-29. Retrieved 2023-01-29.*
12. *Ivan Salvadori, Frank Siqueira (June 2015). "A Maturity Model for Semantic RESTful Web APIs" (https://www.researchgate.net/publication/281287283) . Conference: Web Services (ICWS), 2015 IEEE International Conference OnAt. New York – via Researchgate.*

13. *Gupta, Lokesh. "What is REST API" (http://restfulapi.net/) . RESTful API Tutorial. Archived (https://web.archive.org/web/20161002224340/http://restfulapi.net/) from the original on 2 October 2016. Retrieved 29 September 2016.*
14. *Richardson, Leonard; Amundsen, Mike (2013), RESTful Web APIs, O'Reilly Media, ISBN 978-1-449-35806-8*

Further reading

- Pautasso, Cesare; Wilde, Erik; Alarcon, Rosa (2014), *REST: Advanced Research Topics and Practical Applications* (https://www.springer.com/engineering/sign

als/book/978-1-4614-9298-6) , Springer, ISBN 9781461492986

- Pautasso, Cesare; Zimmermann, Olaf; Leymann, Frank (April 2008), "Restful web services vs. "big"' web services", *Proceedings of the 17th international conference on World Wide Web* (http://www.jopera.org/docs/publications/2008/restws) , pp. 805–814, doi:10.1145/1367497.1367606 (https://doi.org/10.1145%2F1367497.1367606) , ISBN 9781605580852, S2CID 207167438 (https://api.semanticscholar.org/CorpusID:207167438)

- Ferreira, Otavio (Nov 2009), *Semantic Web Services: A RESTful Approach* (https://otaviofff.github.io/restful-grounding/) , IADIS, ISBN 978-972-8924-93-5
- Fowler, Martin (2010-03-18). "Richardson Maturity Model: steps towards the glory of REST" (https://martinfowler.com/articles/richardsonMaturityModel.html) . *martinfowler.com*. Retrieved 2017-06-26.
