WIKIPEDIA

Representational state transfer

Representational state transfer (**REST**) or **RESTful** web services are a way of providing interoperability between computer systems on the Internet. REST-compliant Web services allow requesting systems to access and manipulate textual representations of Web resources using a uniform and predefined set of stateless operations. Other forms of Web services exist, which expose their own arbitrary sets of operations such as WSDL and SOAP.[1]

"Web resources" were first defined on the World Wide Web as documents or files identified by their URLs, but today they have a much more generic and abstract definition encompassing every thing or entity that can be identified, named, addressed or handled, in any way whatsoever, on the Web. In a RESTful Web service, requests made to a resource's URI will elicit a response that may be in XML, HTML, JSON or some other defined format. The response may confirm that some alteration has been made to the stored resource, and it may provide hypertext links to other related resources or collections of resources. Using HTTP, as is most common, the kind of operations available include those predefined by the HTTP methods GET, POST, PUT, DELETE and so on.

By using a stateless protocol and standard operations, REST systems aim for fast performance, reliability, and the ability to grow, by re-using components that can be managed and updated without affecting the system as a whole, even while it is running.

The term *representational state transfer* was introduced and defined in 2000 by Roy Fielding in his doctoral dissertation.[2][3] Fielding's dissertation explained the REST principles were known as the "HTTP object model" beginning in 1994, and were used in designing the HTTP 1.1 and Uniform Resource Identifiers (URI) standards.[4][5][6] The term is intended to evoke an image of how a well-designed Web application behaves: it is a network of Web resources (a virtual state-machine) where the user progresses through the application by selecting links, such as `/user/tom`, and operations such as GET or DELETE (state transitions), resulting in the next resource (representing the next state of the application) being transferred to the user for their use.

Contents

History

Roy Fielding defined REST in his 2000 PhD dissertation "Architectural Styles and the Design of Network-based Software Architectures" at UC Irvine.[2] He developed the REST architectural style in parallel with HTTP 1.1 of 1996–1999, based on the existing design of HTTP 1.0[7] of 1996.

Roy Fielding speaking at OSCON 2008

In a retrospective look at the development of REST, Fielding said:

> Throughout the HTTP standardization process, I was called on to defend the design choices of the Web. That is an extremely difficult thing to do within a process that accepts proposals from anyone on a topic that was rapidly becoming the center of an entire industry. I had comments from well over 500 developers, many of whom were distinguished engineers with decades of experience, and I had to explain everything from the most abstract notions of Web interaction to the finest details of HTTP syntax. That process honed my model down to a core set of principles, properties, and constraints that are now called REST.[7]

Architectural properties

The constraints of the REST architectural style affect the following architectural properties:[2][8]

- Performance - component interactions can be the dominant factor in user-perceived performance and network efficiency[9]
- Scalability to support large numbers of components and interactions among components. Roy Fielding, one of the principal authors of the HTTP specification, describes REST's effect on scalability as follows:

  > REST's client–server separation of concerns simplifies component implementation, reduces the complexity of connector semantics, improves the effectiveness of performance tuning, and increases the scalability of pure server components. Layered system constraints allow intermediaries—proxies, gateways, and firewalls—to be introduced at various points in the communication without changing the interfaces between components, thus allowing them to assist in communication translation or improve performance via large-scale, shared caching. REST enables intermediate processing by constraining messages to be self-descriptive: interaction is stateless between requests, standard methods and media types are used to indicate semantics and exchange information, and responses explicitly indicate cacheability.[2]

- 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 is the resistance to failure at the system level in the presence of failures within components, connectors, or data[9]

Architectural constraints

Six guiding constraints define a RESTful system.[8][10] These constraints restrict the ways that the server may process and respond to client requests so that, by operating within these constraints, the service gains desirable non-functional properties, such as performance, scalability, simplicity, modifiability, visibility, portability, and reliability.[2] If a service violates any of the required constraints, it cannot be considered RESTful.

The formal REST constraints are as follows:

Client-server architecture

The first constraints added to the hybrid style are those of the client-server architectural style. The principle behind the client-server constraints is the separation of concerns. Separating the user interface concerns from the data storage concerns improves the portability of the user interface across multiple platforms. It also improves scalability by simplifying the server components. Perhaps most significant to the Web, however, is that the separation allows the components to evolve independently, thus supporting the Internet-scale requirement of multiple organizational domains.[2]

Statelessness

The client–server communication is constrained by no client context being stored on the server between requests. Each request from any client contains all the information necessary to service the request, and session state is held in the client. The session state can be transferred by the server to another service such as a database to maintain a persistent state for a period and allow authentication. The client begins sending requests when it is ready to make the transition to a new state. While one or more requests are outstanding, the client is considered to be *in transition*. The representation of each application state contains links that may be used the next time the client chooses to initiate a new state-transition.[11]

Cacheability

As on the World Wide Web, clients and intermediaries can cache responses. Responses must therefore, implicitly or explicitly, define themselves as cacheable or not to prevent clients from reusing 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.

Layered system

A client cannot ordinarily tell whether it is connected directly to the end server, or to an intermediary along the way. Intermediary servers may improve system scalability by enabling load balancing and by providing shared caches. They may also enforce security policies.

Code on demand (optional)

Servers can temporarily extend or customize the functionality of a client by transferring executable code. Examples of this may include compiled components such as Java applets and client-side scripts such as JavaScript.

Uniform interface

The uniform interface constraint is fundamental to the design of any REST service.[2] 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 Web-based REST systems. The resources themselves are conceptually separate from the representations that are returned to the client. For example, the server may send data from its database as HTML, XML or 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.

Self-descriptive messages

Each message includes enough information to describe how to process the message. For example, which parser to invoke may be specified by an Internet media type (previously known as a MIME type).[2]

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 actions and resources it needs. As access proceeds, the server responds with text that includes hyperlinks to other actions that are currently available. There is no need for the client to be hard-coded with information regarding the structure or dynamics of the REST service.[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]

- base URL, such as `http://api.example.com/resources`
- an internet media type that defines state transition data elements (e.g., Atom, microformats, application/vnd.collection+json,[14]:91–99 etc.) The current representation tells the client how to compose requests for transitions to all the next available application states. This could be as simple as a URL or as complex as a Java applet.[15]
- standard HTTP methods (e.g., OPTIONS, GET, PUT, POST, and DELETE)[16]

Relationship between URL and HTTP methods

The following table shows how HTTP methods are typically used in a RESTful API:

HTTP methods

Uniform Resource Locator (URL)	GET	PUT	PATCH	POST	DELETE
Collection, such as `https://api.example.com/resources`	**List** the URIs and perhaps other details of the collection's members.	**Replace** the entire collection with another collection.	Not generally used	**Create** a new entry in the collection. The new entry's URI is assigned automatically and is usually returned by the operation.[17]	**Delete** the entire collection.
Element, such as `https://api.example.com/resources/item17`	**Retrieve** a representation of the addressed member of the collection, expressed in an appropriate Internet media type.	**Replace** the addressed member of the collection, or if it does not exist, **create** it.	**Update** the addressed member of the collection.	Not generally used. Treat the addressed member as a collection in its own right and **create** a new entry within it.[17]	**Delete** the addressed member of the collection.

The GET method is a safe method (or *nullipotent*), meaning that calling it produces no side-effects: retrieving or accessing a record does not change it. The PUT and DELETE methods are idempotent, meaning that the state of the system exposed by the API is unchanged no matter how many times more than once the same request is repeated.

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 also describe their APIs as being RESTful, even though these APIs actually don't fulfill all of the architectural constraints described above (especially the uniform interface constraint).[15]

See also

- Atomicity, consistency, isolation, durability (ACID)
- Create, read, update and delete (CRUD)
- Domain Application Protocol (DAP)
- Microservices
- Overview of RESTful API Description Languages
  - OpenAPI Specification — specification for defining interfaces
  - OData – Protocol for REST APIs
  - RAML (software)
  - RSDL (RESTful Service Description Language)
- Resource-oriented architecture (ROA)
- Resource-oriented computing (ROC)
- Semantic URLs
- Service-oriented architecture (SOA)
- Web-oriented architecture (WOA)

References

1. "Web Services Architecture" (https://www.w3.org/TR/2004/NOTE-ws-arch-20040211/#relwwwrest). World Wide Web Consortium. 11 February 2004. 3.1.3 Relationship to the World Wide Web and REST Architectures. Retrieved 29 September 2016.
2. 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. "This chapter introduced the Representational State Transfer (REST) architectural style for distributed hypermedia systems. REST provides a set of architectural constraints that, when applied as a whole, emphasizes scalability of component interactions, generality of interfaces, independent deployment of components, and intermediary components to reduce interaction latency, enforce security, and encapsulate legacy systems."
3. "Fielding discussing the definition of the REST term" (https://groups.yahoo.com/neo/groups/rest-discuss/conversations/topics/6735). groups.yahoo.com. Retrieved 2017-08-08.
4. RFC 1945
5. RFC 2616
6. Fielding, Roy Thomas (2000). "Chapter 6: Experience and Evaluation" (http://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. "Since 1994, the REST architectural style has been used to guide the design and development of the architecture for the modern Web. This chapter describes the experience and lessons learned from applying REST while authoring the Internet standards for the Hypertext Transfer Protocol (HTTP) and Uniform Resource Identifiers (URI), the two specifications that define the generic interface used by all component interactions on the Web, as well as from the deployment of these technologies in the form of the libwww-perl client library, the Apache HTTP Server Project, and other implementations of the protocol standards."
7. "Fielding discusses the development of the REST style" (https://web.archive.org/web/20091111012314/http://tech.groups.yahoo.com/group/rest-discuss/message/6757). Tech.groups.yahoo.com. Archived from the original (http://tech.groups.yahoo.com/group/rest-discuss/message/6757) on November 11, 2009. Retrieved 2014-09-14.
8. Thomas Erl, Benjamin Carlyle, Cesare Pautasso, Raj Balasubramanian (2013). "5.1". In Thomas Erl. *SOA with REST*. Prentice Hall. ISBN 978-0-13-701251-0.
9. 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.

10. Richardson, Leonard; Ruby, Sam (2007), *RESTful Web service* (https://books.google.com/books?id=XUaErakHsoAC), O'Reilly Media, ISBN 978-0-596-52926-0, retrieved 18 January 2011, "The main topic of this book is the web service architectures which can be considered RESTful: those which get a good score when judged on the criteria set forth in Roy Fielding's dissertation."
11. "Fielding talks about application states" (http://tech.groups.yahoo.com/group/rest-discuss/message/5841). Tech.groups.yahoo.com. Retrieved 2013-02-07.
12. "REST HATEOAS" (http://restfulapi.net/hateoas/). RESTfulAPI.net.
13. "What is REST API" (http://restfulapi.net/). *RESTful API Tutorial*. Retrieved 29 September 2016.
14. Richardson, Leonard; Amundsen, Mike (2013), *RESTful Web APIs* (https://www.amazon.com/RESTful-Web-APIs-Leonard-Richardson/dp/1449358063/ref=sr_1_1?ie=UTF8&qid=1442372039&sr=8-1&keywords=restful+web+apis), O'Reilly Media, ISBN 978-1-449-35806-8, retrieved 15 September 2015
15. Roy T. Fielding (2008-10-20). "REST APIs must be hypertext driven" (http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven). roy.gbiv.com. Retrieved 2016-07-06.
16. Berners-Lee, Tim; Fielding, Roy T.; Nielsen, Henrik Frystyk. "Method Definitions" (https://tools.ietf.org/html/rfc1945#section-8). *Hypertext Transfer Protocol -- HTTP/1.0* (https://tools.ietf.org/html/rfc1945). IETF. pp. 30-32. sec. 8. RFC 1945. https://tools.ietf.org/html/rfc1945#section-8.
17. H, Jeremy (16 May 2012). "API Example Using REST" (http://thereisnorightway.blogspot.com/2012/05/api-example-using-rest.html). *There Is No Right Way*. Retrieved 31 July 2014.

Further reading

- Pautasso, Cesare; Wilde, Erik; Alarcon, Rosa (2014), *REST: Advanced Research Topics and Practical Applications* (https://www.springer.com/engineering/signals/book/978-1-4614-9298-6)
- Pautasso, Cesare; Zimmermann, Olaf; Leymann, Frank (April 2008), "RESTful Web Services vs. Big Web Services: Making the Right Architectural Decision" (http://www.jopera.org/docs/publications/2008/restws), *17th International World Wide Web Conference (WWW2008)*, Beijing, China
- Ferreira, Otavio (Nov 2009), *Semantic Web Services: A RESTful Approach* (http://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.