Representational state transfer

From Wikipedia, the free encyclopedia

"REST" redirects here. For other uses, see Rest.

In computing, **Representational State Transfer** (**REST**) is a software architecture style for building scalable web services.[1][2] REST gives a coordinated set of constraints to the design of components in a distributed hypermedia system that can lead to a higher performing and more maintainable architecture.[3]

RESTful systems typically, but not always, communicate over the Hypertext Transfer Protocol with the same HTTP verbs (GET, POST, PUT, DELETE, etc.) which web browsers use to retrieve web pages and to send data to remote servers.[3] REST interfaces usually involve collections of resources with identifiers, for example `/people/paul`, which can be operated upon using standard verbs, such as `DELETE /people/paul`.

The W3C Technical Architecture Group (TAG) developed the REST architectural style in parallel with HTTP 1.1 of 1996-1999, based on the existing design of HTTP 1.0[4] of 1996. The World Wide Web itself represents the largest implementation of a system conforming to the REST architectural style.

Contents

Architectural properties

The architectural properties affected by the constraints of the REST architectural style are:[3][5]

- Performance - component interactions can be the dominant factor in user-perceived performance and network efficiency.[6]
- 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 thus:

> 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.[3]

- Simplicity of interfaces
- 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[6]

Architectural constraints

The architectural properties of REST are realized by applying specific interaction constraints to components, connectors, and data elements.[3][5] One can characterise applications conforming to the REST constraints described in this section as "RESTful".[2] If a service violates any of the required constraints, it cannot be considered RESTful. Complying with these constraints, and thus conforming to the REST architectural style, enables any kind of distributed hypermedia system to have desirable non-functional properties, such as performance, scalability, simplicity, modifiability, visibility, portability, and reliability.[3]

The formal REST constraints are:

Client–server

See also: Client–server model

A uniform interface separates clients from servers. This separation of concerns means that, for example, clients are not concerned with data storage, which remains internal to each server, so that the portability of client code is improved. Servers are not concerned with the user interface or user state, so that servers can be simpler and more scalable. Servers and clients may also be replaced and developed independently, as long as the interface between them is not altered.

Stateless

See also: Stateless protocol

The client–server communication is further 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.[7]

Cacheable

See also: Web cache

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

See also: 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)

See also: Client-side scripting

Servers can temporarily extend or customize the functionality of a client by the transfer of executable code. Examples of this may include compiled components such as Java applets and client-side scripts such as JavaScript. "Code on demand" is the only optional constraint of the REST architecture.

Uniform interface

The uniform interface constraint is fundamental to the design of any REST service.[3] The uniform interface simplifies and decouples the architecture, which enables each part to evolve independently. The four constraints for this uniform interface are:

Identification of resources
: 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.

Manipulation of resources through these 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). Responses also explicitly indicate their cacheability.[3]

Hypermedia as the engine of application state (HATEOAS)

Clients make state transitions only through actions that are dynamically identified within hypermedia by the server (e.g., by hyperlinks within hypertext). Except for simple fixed entry points to the application, a client does not assume that any particular action is available for any particular resources beyond those described in representations previously received from the server.

Applied to web services

Web service APIs that adhere to the REST architectural constraints are called RESTful APIs. HTTP-based RESTful APIs are defined with these aspects:

- base URI, such as `http://example.com/resources/`
- an Internet media type for the data. This is often JSON but can be any other valid Internet media type (e.g., XML, Atom, microformats, images, etc.)
- standard HTTP methods (e.g., GET, PUT, POST, or DELETE)
- hypertext links to reference state
- hypertext links to reference-related resources[8]

Example

The following table shows the HTTP methods that are typically used to implement a RESTful API:

RESTful API HTTP methods

Resource	GET	PUT	POST	DELETE
Collection URI, such as `http://api.example.com/v1/resources/`	**List** the URIs and perhaps other details of the collection's members.	**Replace** the entire collection with another collection.	**Create** a new entry in the collection. The new entry's URI is assigned automatically and is usually returned by the operation.[9]	**Delete** the entire collection.
Element URI, such as `http://api.example.com/v1/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.	Not generally used. Treat the addressed member as a collection in its own right and **create** a new entry in it.[9]	**Delete** the addressed member of the collection.

The PUT and DELETE methods are referred to as idempotent, meaning that the operation will produce the same result no matter how many times it is repeated. The GET method is a safe method (or *nullipotent*), meaning that calling it produces no side-effects. In other words, retrieving or accessing a record does not change it.

Unlike SOAP-based web services, there is no "official" standard for RESTful web APIs.[10] This is because REST is an architectural style, while SOAP is a protocol. Even though REST is not a standard *per se*, most RESTful implementations make use of standards such as HTTP, URI, JSON, and XML.[10]

See also

- Create, read, update and delete (CRUD)
- Domain Application Protocol (DAP)
- Hypermedia as the Engine of Application State
- OData – Protocol for REST APIs
- RAML (software)
- Resource-oriented architecture (ROA)
- Resource-oriented computing (ROC)
- RSDL (RESTful Service Description Language)
- Semantic URLs
- Service-oriented architecture (SOA)
- Swagger — specification for defining interfaces

Further reading

- Pautasso, Cesare; Wilde, Erik; Alarcon, Rosa (2014), *REST: Advanced Research Topics and Practical Applications* (http://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

References

1. Fielding, R. T.; Taylor, R. N. (2000). "Principled design of the modern Web architecture". pp. 407–416. doi:10.1145/337180.337228 (https://dx.doi.org/10.1145%2F337180.337228).
2. Richardson, Leonard; Sam Ruby (2007), *RESTful web service* (http://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."

3. 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."
4. "Fielding discusses the development of the REST style" (http://web.archive.org/web/20091111012314/http://tech.groups.yahoo.com/group/rest-discuss/message/6757). Tech.groups.yahoo.com. Retrieved 2014-09-14.
5. 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.
6. 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.
7. "Fielding talks about application states" (http://tech.groups.yahoo.com/group/rest-discuss/message/5841). Tech.groups.yahoo.com. Retrieved 2013-02-07.
8. 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 2013-02-07.
9. 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.

```
"POST /api/carts
Content-Type:application/vnd.example.coolapp.cart-v1+xml
Content-Length: 1032

<cart>
  <customerId>1343</customerId>
  <lineItems>
    <lineItem>
      <productId>12343</productId>
      <quantity>4</quantity>
    </lineItem>
    ...
  </lineItems>
</cart>

Response:
HTTP/1.1 201 Created
Location: /api/carts/323392
"
```

10. Elkstein, M (February 2008). "Learn REST: A Tutorial" (http://rest.elkstein.org/2008/02/what-is-rest.html). blogger.com. Retrieved 16 April 2015.

Categories: Cloud standards | Software architecture | Web 2.0 neologisms

---