The REST architectural style describes six constraints. These constraints, applied to the architecture, were originally communicated by Roy Fielding in his doctoral dissertation (see [http://www.ics.uci.edu/~fielding/pubs/dissertation/rest\_arch\_style.htm](http://www.ics.uci.edu/%7Efielding/pubs/dissertation/rest_arch_style.htm)) and defines the basis of RESTful-style.

**The six constraints are: (click the constraint to read more)**

* [Uniform Interface](http://www.restapitutorial.com/lessons/whatisrest.html)

The uniform interface constraint defines the interface between clients and servers. It simplifies and decouples the architecture, which enables each part to evolve independently. The four guiding principles of the uniform interface are:

**Resource-Based**

Individual resources are identified in requests using URIs as resource identifiers. The resources themselves are conceptually separate from the representations that are returned to the client. For example, the server does not send its database, but rather, some HTML, XML or JSON that represents some database records expressed, for instance, in Finnish and encoded in UTF-8, depending on the details of the request and the server implementation.

**Manipulation of Resources 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 on the server, provided it has permission to do so.

**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 cache-ability.

**Hypermedia as the Engine of Application State (HATEOAS)**

Clients deliver state via body contents, query-string parameters, request headers and the requested URI (the resource name). Services deliver state to clients via body content, response codes, and response headers. This is technically referred-to as hypermedia (or hyperlinks within hypertext).

Aside from the description above, HATEOS also means that, where necessary, links are contained in the returned body (or headers) to supply the URI for retrieval of the object itself or related objects. We'll talk about this in more detail later.

The uniform interface that any REST services must provide is fundamental to its design.

* [Stateless](http://www.restapitutorial.com/lessons/whatisrest.html)

As REST is an acronym for REpresentational State Transfer, statelessness is key. Essentially, what this means is that the necessary state to handle the request is contained within the request itself, whether as part of the URI, query-string parameters, body, or headers. The URI uniquely identifies the resource and the body contains the state (or state change) of that resource. Then after the server does it's processing, the appropriate state, or the piece(s) of state that matter, are communicated back to the client via headers, status and response body.

Most of us who have been in the industry for a while are accustomed to programming within a container which provides us with the concept of “session” which maintains state across multiple HTTP requests. In REST, the client must include all information for the server to fulfill the request, resending state as necessary if that state must span multiple requests. Statelessness enables greater scalability since the server does not have to maintain, update or communicate that session state. Additionally, load balancers don't have to worry about session affinity for stateless systems.

So what's the difference between state and a resource? State, or application state, is that which the server cares about to fulfill a request—data necessary for the current session or request. A resource, or resource state, is the data that defines the resource representation—the data stored in the database, for instance. Consider application state to be data that could vary by client, and per request. Resource state, on the other hand, is constant across every client who requests it.

Ever had back-button issues with a web application where it went AWOL at a certain point because it expected you to do things in a certain order? That's because it violated the statelessness principle. There are cases that don't honor the statelessness principle, such as three-legged OAuth, API call rate limiting, etc. However, make every effort to ensure that application state doesn't span multiple requests of your service(s).

* [Cacheable](http://www.restapitutorial.com/lessons/whatisrest.html)

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

* [Client-Server](http://www.restapitutorial.com/lessons/whatisrest.html)

The 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 is not altered.

* [Layered System](http://www.restapitutorial.com/lessons/whatisrest.html)

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. Layers may also enforce security policies.

* [Code on Demand (optional)](http://www.restapitutorial.com/lessons/whatisrest.html)

Servers are able to temporarily extend or customize the functionality of a client by transferring logic to it that it can execute. Examples of this may include compiled components such as Java applets and client-side scripts such as JavaScript.

Complying with these constraints, and thus conforming to the REST architectural style, will enable any kind of distributed hypermedia system to have desirable emergent properties, such as performance, scalability, simplicity, modifiability, visibility, portability and reliability.

**NOTE:** The only optional constraint of REST architecture is code on demand. If a service violates any other constraint, it cannot strictly be referred to as RESTful.