What is REST?

Quick Answer

REST is a way of producing and consuming web APIs. It tends to be simpler and easier to work with than other API methodologies (such as SOAP). It seeks improved scalability, performance, and client-server independence.

Digging Deeper

REST is an acronym for *Representational State Transfer*. This Wikipedia article offers a good primer on the subject but for those who want to go right to the source material, read Chapter 5 of Dr. Roy Fielding's dissertation. There are numerous web articles and vlogs that discuss various REST topics. This article will focus on what REST means to the SRP HTTP Framework.

At a high-level, REST is no different from other web API methodologies (generally known as *RPC* or *remote procedure call*) such as SOAP or XML-RPC. They all communicate with web servers using URLs and HTTP. Contrary to conventional thinking, REST is *not* a standard but it does point to other standards (e.g., HTTP). REST attempts to describe a philosophy of building web APIs using six different constraints:

- Client-Server Architecture
- Statelessness
- Cacheability
- Layered System
- Code on Demand
- Uniform Interface

Client-Server Architecture is assumed when working with web APIs, so we won't explore this. The *Cacheability* and *Layered System* constraints are generally handled through intermediary devices between the client and the server as a way of improving performance and reliability, so we won't explore these either. Finally, *Code on Demand* is an optional constraint that is only useful in environments where the client and the server are tightly controlled. Therefore we'll skip over this as well. This leaves us to explore *Statelessness* and *Uniform Interface*.

Statelessness

A stateless system is one where the server is unaware of the state of the client. That is, the server makes no assumptions about what data the client already has or what options are available to the client. In systems where the client state is managed (i.e., a stateful design), this is often handled through session managers, which are server-side systems that track the activity of each client. REST maintains that stateful designs will eventually become over burdened and will hinder scalability.

The SRP HTTP Framework does not enforce statelessness *nor* does it offer any tools to make the system stateful. Statelessness is really a matter of self-governance in the design of the API. Developers can move toward statelessness by avoiding, or minimizing, database locks. Resources can also be returned along with metadata (see *Uniform Interface* below) that instruct the client how it can request a state change. Stateless APIs are not difficult to implement unless they share resources (i.e., database tables) with a desktop OpenInsight environment. In these cases, OpenInsight clients typically maintain pessimistic locks on database rows. Stateless APIs typically use optimistic locks (or no locks at all), both of which require a way to resolve conflicts if an OpenInsight client is retaining a lock for an extended period of time.

Uniform Interface

REST, as it is argued, attempts to use HTTP more faithfully. This is the primary basis for a uniform interface. That is, by adhering to the published HTTP standards, API producers and consumers can better anticipate how to interface with each other. It also provides for greater decoupling, allowing independent evolution between the client and the server.

One key element is that the URL is a reference to a *resource* on the server rather than a reference to *function* (or remote procedure) on the server. REST is known for embracing all of the documented HTTP methods so clients can convey a wide variety of intent with the resource. For instance:

API	Purpose
`POST` `/customers`	Create a new customer.
`GET` `/customers/{ID}`	Read a specified customer .
`PUT` `/customers/{ID}`	Update a specified customer.
`DELETE` `/customers/{ID}`	Delete a specified customer.

Application and database developers will recognize the above as the basic functions of CRUD. REST is far more than just an alternative way to implement CRUD, but it is a healthy start to understand how REST differs from RPC. One major takeaway is that REST uses four of the well defined HTTP methods to specify action (or intent) and only uses one URL to specify the resource. The `{ID}` used by some of the APIs technically means there are two URLs, but the resource itself, i.e., customers, is still represented by a single URL.

RPC methodologies tend to focus on just two HTTP methods, GET and POST, and rely upon the URL or a payload body to specify the action. Here is a very simple RPC example:

API	Purpose
POST `/newcustomer`	Create a new customer.
GET `/readcustomer`	Read a specified customer .
POST `/updatecustomer`	Update a specified customer.
POST `/deletecustomer`	Delete a specified customer.

We will observe that each CRUD action uses a different URL. SOAP differs from standard RPC in that the URL will tend to be singular but the entire action will be described in the payload.

HTTP provides a way of managing metadata through the use of *request headers* and *response headers*. Effective use of headers avoids the need to build proprietary messaging within the payload or URL query params. A good example of this is the *Accept* request header. Clients should use the Accept header to indicate their preferred data format (aka media type). Servers should look for the value specified in the Accept header and return the resource in this format if it is able.

HTTP also provides a set of response status codes that inform the client how the request was handled. For example: 200 means OK, 201 means a new resource was created, 404 means the resource does not exist, 405 means the HTTP method is not supported, etc. Servers should send back the most applicable status code and clients should attempt to handle any valid status code appropriately.

An important aspect of the Uniform Interface constraint is described as *Hypermedia As The Engine Of Application State*, or HATEOAS. We discuss the nature and value of HATEOAS in our *Why is HATEOAS important?* article, but we'll provide a simple explanation of it here. HATEOAS is a design feature where information about the state of a resource should be provided to the client through hyperlinks (aka hypermedia). This avoids the need for maintaining state and it avoids the need for the client to assume (or hardcode) how state change requests are made. Incorporating HATEOAS to the design of web APIs requires a fair amount of extra work and planning. This often drives many developers to opt out of HATEOAS. This is often referred to as *pragmatic REST* whereas *purist REST* includes and advocates for HATEOAS. Regardless of the ongoing internal debates, Dr. Fielding himself has expressed his feelings rather clearly, "...if the engine of application state (and hence the API) **is not being driven by hypertext**, then it **cannot be RESTful** and **cannot be a REST API**" (*emphasis added*).

We believe HATEOAS is a very important part of a well designed RESTful API. Therefore, the SRP HTTP Framework is designed to make this easy to implement.