A concrete implementation of a REST Web service follows four basic design principles:

* Use HTTP methods explicitly
* Be stateless
* Expose directory structure-like URIs
* Transfer XML, JSON, or both

Rest asks developers to use HTTP methods explicitly and in a way that’s consistent with the protocol definition. This basic REST design principle establishes a one-to-one mapping between create, read, update, and delete (CRUD) operations and HTTP methods. According to this mapping:

* To create a resource on the server, use POST
* To retrieve a resource, use GET
* To change the state of a resource or to update it, use PUT
* To remove or delete a resource, use DELETE

GET is an operation that should be free of side effects, a property also known as *idempotence*.

As a general design principle, it helps to follow REST guidelines for using HTTP methods explicitly by using nouns in URIs instead of verbs. In a RESTful web service, the verbs – POST, GET, PUT, and DELETE, are already defined by the protocol. And ideally, to keep the interface generalized and to allow clients to be explicit about the operations they invoke, the Web service should not define more verbs or remote procedures, such as /adduser or /updateuser. This general design principle also applies to the body of an HTTP request, which is intended to be used to transfer resource state, not to carry the name of a remote method or remote procedure to be invoked.

**Be stateless**

Using intermediary servers to improve scale requires REST Web service clients to send complete, independent requests; that is, to send request that include all data needed to be fulfilled so that the components in the intermediary servers may forward, route, and load-balance without any state being held locally in between requests.

A complete, independent request doesn’t require the server, while processing a request, to retrieve any kind of application context or state. A REST Web service application (or client) includes within the HTTP heads and body of a request all of the parameters, context, and data needed by the server-side component to generate a response. Statelessness in this sense improves Web service performance and simplifies the design and implementation of server-side components because the absence of state on the server removes the need to synchronize session data with an external application.