# REST Principles:

REST Stands for “**Representational State Transfer**”

It means when a RESTful API is called, the server will transfer to the client a representation of the state of the requested resource.

For example, when a developer calls Instagram API to fetch a specific user (the resource), the API will return the state of that user, including their name, the number of posts that user posted on Instagram so far, how many followers they have, and more.

The representation of the state can be in a JSON format, and probably for most APIs, this is indeed the case. It can also be in XML or HTML format.

REST doesn't add any specific functionality to HTTP but is an architectural style that was developed alongside HTTP and most commonly uses HTTP for its application layer protocol.

REST is a set of 'rules' (or 'constraints'). defining the characteristics of the optimal application for a distributed hypermedia system, like the World Wide Web. After all, Roy Thomas Fielding put those constraints together so people could improve the WWW and create applications that are scalable, performant, fault-tolerant and extendable. The more constraints you follow, the stronger your web API becomes and the harder it becomes to create.

Architectural Constraints

REST defines 6 architectural constraints which make any web service – a truly RESTful API.

* Uniform interface
* Client-server
* Stateless
* Cacheable
* Layered system
* Code on demand (optional)

**Uniform interface**

By REST you use the same concept to decouple the client from the implementation of the REST service. In order to define such an interface (contract between the client and the service), you have to use standards. Because if you want your APIs to be globally accepted, you have to enforce global concepts, like standards to make them understand each other.

**Identification of resources** - The REST style is centred around resources. This is unlike SOAP and other RPC styles that are modelled around procedures (or methods) and a resource is basically anything that can be named or resources are usually the entities from the business domain (i.e. customers, orders, products, etc.). Each resource is defined in requests using URIs (also named as endpoint) as unique resource identifiers So the first thing when you design a REST API is to design these URIs or endpoints so that your data is well structured and your API easy to use and understand. Normally, you would use the first URI below to access a collection of resources (i.e., several customers) and the second URI to access a specific resource inside that collection (i.e., a specific customer):

1) https://api.restapi.com/customers

2) https://api.restapi.com/customers/123456

This separation between the public interface, the collection of URIs of your API, and the business logic of your application hidden behind the URI schema make REST a solid way of structuring your API logic. This way, REST decouples the application architecture, allowing each endpoint (function or module) to evolve independently. This reduces application complexity and maintenance overhead.

**Manipulation of resources through these representations** - This means that the client does not interact directly with the server’s resource. For example, we do not allow the client to run SQL statements against our database tables. It just means that we show the resource’s data (i.e., state) in a neutral format. This is similar to how the data for a web page can be stored in a database but is always send to the browser in HTML. The most common format for RESTful web services is JSON, which is used in the body of the HTTP requests and responses:

{

"id":1,

"firstname":"Arpit",

"lastname":"Jain"

}

When a client wants to update the resource, it gets a representation of that resource from the server, updates the representation with the new data, sends the updated representation to the server, and ask the server to update its resource so it corresponds with the new representation. The benefit is that you avoid a strong coupling between the client and server (like with RMI in Java), so you can change the underlying implementation without affecting the clients. It also makes it easier for clients as they do not need to understand the underlying technology used by each server that they interact with. An identified resource can be returned in various formats such as HTML, XML, SVG, JSON, PNG, etc. These formats are representations of the identified resource. REST-ful applications may support more than one representation of the same resource at the same URI. And REST-ful applications allow clients to indicate which representation of a resource they wish to receive. This is accomplished via the Accept HTTP header that is passed by the client to the server with each request for a resource.

Resources can be updated (or added) by sending representations from the client to the server. REST-ful applications may support more than one representation for updates to the same resource. And REST-ful applications allow clients to indicate their preferred representation when sending data to the server. This is accomplished via the Content-Type HTTP header that is passed by the client to the server with each resource sent to the server.

**Self-descriptive messages**- The third constraint in the Uniform Interface is that each message (i.e. request/response) must include enough information for the receiver to understand it in isolation. Each message must have a media type (for example application/json or application/xml) that tells the receiver how the message should be parsed.

HTTP is not formally required for RESTful web services, but if you use the HTTP methods you should follow their formal meaning, so the user won’t rely on out of band information to understand them (i.e. don’t use **POST** to retrieve data or **GET** to save data).

So for the Customer URIs, which we defined earlier, we can expose the following methods for the client to use:

Create a new customer POST /customers

Delete an existing customer DELETE /customers/{id}

Get a specific customer GET /customers/{id}

Search for customers GET /customers

Update an existing customer PUT /customers/{id}

The benefit is that the four HTTP methods are clearly defined, so an API user who knows HTTP but doesn’t know our system can quickly guess what the service is doing by only looking at the HTTP method and URI path.

**Hypermedia as the engine of application state (A.K.A. HATEOAS)**- The fourth and final sub-constraint in the Uniform Interface is called Hypermedia as the Engine of Application State (HATEOAS). It sounds a bit overwhelming, but, it’s a simple concept.

A web page is an instance of application state, hypermedia is text with hyperlinks. The hypermedia drives the application state. In other words, we click on links to move to new pages (i.e., application states). So, when you are surfing the web, you are using hypermedia as the engine of application state!

This means that HATEOAS should guide a user through the interface by offering control alongside the data. A client would use these to navigate through the interface dynamically. The client should only know the root URI and can browse and use the entire interface by following links if it knows all the media-types that the server uses for his representations.

For example, inside a customer representation there could be a links section with links to the customer’s orders:

{

"id":1,

"firstname":"Arpit",

"lastname":"Jain",

"\_links": {

"self": {

"href":"https://api.restapi.com/customers/12"

},

"orders": {

"href":"https://api.restapi.com/customers/12/orders"

}

}

}

An enormous benefit is that the API user doesn’t need to look in the API documentation to see how to find the customer’s orders, so he or she can easily explore while developing without having to refer to out-of-band API documentation. It also means that the API user does not need to hardcode (and manually construct) the URIs that he or she wants to call.

**Client-server**

This essentially means that client application and server application must be able to evolve separately without any dependency on each other and I mean it should follow separation of concerns. Separation of concerns meaning the code on the client side can be changed at any time without affecting the conditions of the server, and the code on the server side can be changed without changing the conditions of the client. By separating concerns in this way, we can improve scalability and the flexibility of the interface across platforms.

A client should know only resource URIs and that’s all. As long as the interface between them is not altered, servers and clients may also be replaced and developed independently,

**Stateless**

Roy fielding got inspiration from HTTP, so it reflects in this constraint. Make all client-server interaction stateless. The server will not store anything about latest HTTP request client made. It will treat each request as new and it should not depend on any previous information shared between the two. No session, no history.

If client application needs to be a stateful application for the end user, where user logs in once and do other authorized operations thereafter, then each request from the client should contain all the information necessary to service the request – including authentication and authorization details.

No client context shall be stored on the server between requests. The client is responsible for managing the state of the application.

**Cacheable**

In today’s world, caching of data and responses is of utmost important wherever they are applicable/possible. Caching brings performance improvement for the client, and better scope for scalability for a server because the load has reduced.

In REST, each response can be labelled as cacheable or non-cacheable. The cached response can be used for the response of the request rather than asking the server so we can eliminate the communication between the server and client up to some degree. Though, the shortcoming of this feature is that if a server has some new information for a cached request the client might not get the updated information.

**Layered system**

Components can’t see beyond immediate layer. REST allows you to use a layered system architecture where you deploy the APIs on server A, and store data on server B and authenticate requests in Server C. These servers might provide a security layer, a caching layer, a load-balancing layer, or other functionality. Those layers should not affect the request or the response. The client is agnostic as to how many layers if any, there are between the client and the actual server responding to the request.

Code on demand (optional)

Well, this constraint is optional. Most of the time you will be sending the static representations of resources in the form of XML or JSON. But when you need to, you are free to return executable code to support a part of your application e.g. clients may call your API to get a UI widget rendering code. This simplifies things by reducing the number of features required to be pre-installed.

**Conclusion**

All the above constraints help you build a truly RESTful API and you should follow them. Still, at times you may find yourself violating one or two constraints. Do not worry, you are still making a RESTful API – but not “truly RESTful”.

Notice that all the above constraints are most closely related to WWW (the web). Using RESTful APIs, you can do the same thing with your web services what you do to web pages.

The term "REST API" is a new way to talk about HTTP based services and data access, even though well-designed HTTP based services have been around a long time. The way people previously used "non-REST APIs" (based on HTTP) is like how RESTful APIs are used today - an application uses an HTTP client to send requests and process responses. The application could be a desktop program, or it could be a server that uses third-party services via their HTTP API. The benefit of using RESTful services comes from consistency - every service everywhere uses the same status codes to mean the same thing (200 means 'success', 401 means 'unauthorized', etc), the same requests mean the same thing (GET is for fetching data, POST is for sending data, etc), common issues and concerns are standardized and documented (like caching, proxying, defining content formats, etc).