# REST

REST is acronym for **Re**presentational **S**tate **T**ransfer. It is architectural style for **distributed hypermedia systems** and was first presented by Roy Fielding in 2000 in his famous [dissertation](https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm).

Like any other architectural style, REST also does have it’s own [6 guiding constraints](https://restfulapi.net/rest-architectural-constraints/) which must be satisfied if an interface needs to be referred as **RESTful**. These principles are listed below.

## Guiding Principles of REST

1. **Client–server** – By separating the user interface concerns from the data storage concerns, we improve the portability of the user interface across multiple platforms and improve scalability by simplifying the server components.
2. **Stateless** – Each request from client to server must contain all of the information necessary to understand the request, and cannot take advantage of any stored context on the server. Session state is therefore kept entirely on the client.
3. **Cacheable** – Cache constraints require that the data within a response to a request be implicitly or explicitly labeled as cacheable or non-cacheable. If a response is cacheable, then a client cache is given the right to reuse that response data for later, equivalent requests.
4. **Uniform interface** – By applying the software engineering principle of generality to the component interface, the overall system architecture is simplified and the visibility of interactions is improved. In order to obtain a uniform interface, multiple architectural constraints are needed to guide the behavior of components. REST is defined by four interface constraints: identification of resources; manipulation of resources through representations; self-descriptive messages; and, hypermedia as the engine of application state.
5. **Layered system** – The layered system style allows an architecture to be composed of hierarchical layers by constraining component behavior such that each component cannot “see” beyond the immediate layer with which they are interacting.
6. **Code on demand (optional)** – REST allows client functionality to be extended by downloading and executing code in the form of applets or scripts. This simplifies clients by reducing the number of features required to be pre-implemented.

For more clarity, let’s divide the **resource archetypes** into four categories (document, collection, store and controller) and then **you should always target to put a resource into one archetype and then use it’s naming convention consistently**. For uniformity’s sake, resist the temptation to design resources that are hybrids of more than one archetype.

1. A document resource is a singular concept that is akin to an object instance or database record
2. http://api.example.com/device-management/managed-devices/{device-id}
3. http://api.example.com/user-management/users/{id}
4. http://api.example.com/user-management/users/admin

2. A collection resource is a server-managed directory of resources. Clients may propose new resources to be added to a collection.

http://api.example.com/device-management/managed-devices

http://api.example.com/user-management/users

http://api.example.com/user-management/users/{id}/accounts

3. A store is a client-managed resource repository. A store resource lets an API client put resources in, get them back out, and decide when to delete them. A store never generates new URIs. Instead, each stored resource has a URI. The URI was chosen by a client when it was initially put into the store.

1. [http://api.example.com/song-management/users/{id}/playlists](http://api.example.com/song-management/users/%7bid%7d/playlists)

#### controller

A controller resource models a procedural concept. Controller resources are like executable functions, with parameters and return values; inputs and outputs.

Use “verb” to denote controller archetype.

http://api.example.com/cart-management/users/{id}/cart/checkout

http://api.example.com/song-management/users/{id}/playlist/play

In the context of REST APIs, when making multiple identical requests has the same effect as making a single request – then that REST API is called **idempotent**.

If you follow REST principles in designing API, you will have automatically **idempotent REST APIs** for GET, PUT, DELETE, HEAD, OPTIONS and TRACE HTTP methods. Only POST APIs will not be idempotent.

1. POST is NOT idempotent.
2. GET, PUT, DELETE, HEAD, OPTIONS and TRACE are idempotent.

Below given points may serve as a checklist for designing the security mechanism for REST APIs.

#### Keep it Simple

Secure an API/System – just how secure it needs to be. Every time you make the solution more complex “unnecessarily,” you are also likely to leave a hole.

#### Always Use HTTPS

By always using [SSL](https://www.digicert.com/ssl/), the authentication credentials can be simplified to a randomly generated access token. The token is delivered in the username field of HTTP Basic Auth. It’s relatively simple to use, and you get a lot of security features for free.

If you use [HTTP 2](https://http2.github.io/), to improve performance – you can even [send multiple requests over a single connection](https://en.wikipedia.org/wiki/HTTP_persistent_connection), that way you avoid the complete TCP and SSL handshake overhead on later requests.

#### Use Password Hash

Passwords must always be hashed to protect the system (or minimize the damage) even if it is compromised in some hacking attempts. There are many such [hashing algorithms](https://howtodoinjava.com/security/how-to-generate-secure-password-hash-md5-sha-pbkdf2-bcrypt-examples/) which can prove really effective for password security e.g. PBKDF2, bcrypt and scrypt algorithms.

#### Never expose information on URLs

Usernames, passwords, session tokens, and API keys should not appear in the URL, as this can be captured in web server logs, which makes them easily exploitable.

https://api.domain.com/user-management/users/{id}/someAction?apiKey=abcd123456789 //Very BAD !!

The above URL exposes the API key. So, never use this form of security.

#### Consider OAuth

Though [basic auth](https://en.wikipedia.org/wiki/Basic_access_authentication) is good enough for most of the APIs and if implemented correctly, it’s secure as well – yet you may want to consider [OAuth](https://tools.ietf.org/html/rfc6749) as well. The OAuth 2.0 authorization framework enables a third-party application to obtain limited access to an HTTP service, either on behalf of a resource owner by orchestrating an approval interaction between the resource owner and the HTTP service, or by allowing the third-party application to obtain access on its behalf.

#### Consider Adding Timestamp in Request

Along with other request parameters, you may add a request timestamp as an HTTP custom header in API requests. The server will compare the current timestamp to the request timestamp and only accepts the request if it is within a reasonable timeframe (1-2 minutes, perhaps).

This will prevent very basic [replay attacks](https://en.wikipedia.org/wiki/Replay_attack) from people who are trying to [brute force](https://en.wikipedia.org/wiki/Brute-force_attack) your system without changing this timestamp.

#### Input Parameter Validation

Validate request parameters on the very first step, before it reaches to application logic. Put strong validation checks and reject the request immediately if validation fails. In API response, send relevant error messages and example of correct input format to improve user experience.

APIs only need to be up-versioned when a breaking change is made. Breaking changes include:

* a change in the format of the response data for one or more calls
* a change in the response type (i.e. changing an integer to a float)
* removing any part of the API.

http://api.example.com/v1

Accept-version: v1

Accept: application/vnd.example+json;version=1.0

REST APIs can return the resource representations in several formats such as XML, JSON, HTML, or even plain text. All such forms can be compressed to a lesser number of bytes to save bandwidth over the network. **Different protocols use different techniques to enable compression** and notify the clients about the compression scheme – so that the client can decompress it before consuming the representations.

*Compression, like encryption, is something that happens to a representation in transit and must be undone before the client can use the representation.*

HTTP is most widely used protocol for REST – so I am taking example of **HTTP specific response compression**.

## Compression Related Request/Response Headers

### Accept-Encoding

While requesting resource representations – along with an HTTP request, the client sends an Accept-Encoding header that says what kind of compression algorithms the client understands.

The two standard values for Accept-Encoding are **compress** and **gzip**.

A sample request with accept-encoding header looks like this :

GET /employees HTTP/1.1

Host: www.domain.com

Accept: text/html

Accept-Encoding: gzip, compress

Other possible usage of accept-encoding may be:

Accept-Encoding: compress, gzip

Accept-Encoding:

Accept-Encoding: \*

Accept-Encoding: compress;q=0.5, gzip;q=1.0

Accept-Encoding: gzip;q=1.0, identity; q=0.5, \*;q=0

If an Accept-Encoding field is present in a request, and if the server cannot send a response which is acceptable according to the Accept-Encoding header, then the server SHOULD send an error response with the **406 (Not Acceptable)** status code.

### Content-Encoding

If the server understands one of the compression algorithms from Accept-Encoding, it can use that algorithm to compress the representation before serving it. When successfully compressed, server lets know the client of encoding scheme by another HTTP header i.e. Content-Encoding.

200 OK

Content-Type: text/html

Content-Encoding: gzip

If the content-coding of an entity in a request message is not acceptable to the origin server, the server SHOULD respond with a status code of **415 (Unsupported Media Type)**. If multiple encodings have been applied to an entity, the content encodings MUST be listed in the order in which they were applied.

## Server-driven Vs Agent-driven Content Negotiation

If the selection of the best representation for a response is made by an algorithm located at the server, it is called server-driven negotiation. If that selection is made at agent or client side, its called agent-driven content negotiation

So, most REST API implementations rely on agent driven content negotiations. Agent driven content negotiation rely on usage of HTTP request headers or resource URI patterns.

REST APIs use the **Status-Line** part of an HTTP response message to inform clients of their request’s overarching result. [RFC 2616](https://www.ietf.org/rfc/rfc2616.txt) defines the [Status-Line syntax](https://www.w3.org/Protocols/rfc2616/rfc2616-sec6.html#sec6.1) as shown below:

|  |  |
| --- | --- |
| **CATEGORY** | **DESCRIPTION** |
| **1xx: Informational** | Communicates transfer protocol-level information. |
| **2xx: Success** | Indicates that the client’s request was accepted successfully. |
| **3xx: Redirection** | Indicates that the client must take some additional action in order to complete their request. |
| **4xx: Client Error** | This category of error status codes points the finger at clients. |
| **5xx: Server Error** | The server takes responsibility for these error status codes. |

200-success

201- created

202- received

301 – moved permanently

400. – bad req

401 – unauthorized

403 – forbidden

404 – not found

500 – Internal server error

Sometimes client may want to set their preferences when requesting multiple MIME types. To set this preference, q parameter (*relative quality factor*) is used.

**Value of q parameter can be from 0 to 1**. 0 is lowest value (i.e. least preferred) and 1 is highest (i.e. most preferred).

A sample usage can be:

|  |
| --- |
| Accept : text/html, application/xml;q=0.9, \*/\*;q=0.8 |

**HTTP2:**

Server push

Multiplexing multiple messages in one TCP connection

Compression of request headers

Binary protocol

* Protocol negotiation mechanism — protocol electing, eg. HTTP/1.1, HTTP/2 or other.
* High-level compatibility with HTTP/1.1 — methods, status codes, URIs and header fields.
* Page load speed improvements trough:
* Compression of request headers
* Binary protocol
* HTTP/2 Server Push
* Request multiplexing over a single TCP connection
* Request pipelining
* HOL blocking (Head-of-line) — Package blocking

In the context of an [HTTP](https://en.wikipedia.org/wiki/HTTP) transaction, **basic access authentication** is a method for an [HTTP user agent](https://en.wikipedia.org/wiki/User_agent) (e.g. a web browser) to provide a [user name](https://en.wikipedia.org/wiki/User_name) and [password](https://en.wikipedia.org/wiki/Password) when making a request. In basic HTTP authentication, a request contains a header field in the form of Authorization: Basic <credentials>, where credentials is the [Base64](https://en.wikipedia.org/wiki/Base64) encoding of ID and password joined by a single colon :.s

The *WWW-Authenticate* field for basic authentication is constructed as following:

WWW-Authenticate: Basic realm="User Visible Realm"

The server may choose to include the *charset* parameter from [RFC](https://en.wikipedia.org/wiki/RFC_(identifier)) [7617](https://tools.ietf.org/html/rfc7617) [[2]](https://en.wikipedia.org/wiki/Basic_access_authentication#cite_note-:0-2):

When the user agent wants to send authentication credentials to the server, it may use the *Authorization* field.

The *Authorization* field is constructed as follows:[[6]](https://en.wikipedia.org/wiki/Basic_access_authentication#cite_note-RFC7617-6)

1. The username and password are combined with a single colon (:). This means that the username itself cannot contain a colon.
2. The resulting string is encoded into an octet sequence. The character set to use for this encoding is by default unspecified, as long as it is compatible with US-ASCII, but the server may suggest use of UTF-8 by sending the *charset* parameter.[[6]](https://en.wikipedia.org/wiki/Basic_access_authentication#cite_note-RFC7617-6)
3. The resulting string is encoded using a variant of Base64.
4. The authorization method and a space (e.g. "Basic ") is then prepended to the encoded string.

For example, if the browser uses Aladdin as the username and OpenSesame as the password, then the field's value is the Base64 encoding of Aladdin:OpenSesame, or QWxhZGRpbjpPcGVuU2VzYW1l. Then the Authorization header will appear as:

Authorization: Basic QWxhZGRpbjpPcGVuU2VzYW1l

A client may avoid a login prompt when accessing a basic access authentication by prepending *username*:*password*@ to the hostname in the URL. For example, the following would access the page *index.html* at the web site *www.example.com* with the secure HTTPS protocol and provide the username *Aladdin* and the password *OpenSesame* credentials via basic authorization:

## **What is OAuth?**

OAuth is an open standard for authorization, commonly used as a way for Internet users to log in to third party websites using their Microsoft, Google, Facebook, Twitter, One Network etc. accounts without exposing their password.

you can implement your own OAuth server, here I am explaining about social auth. so the term OAuth here after refers to social auth with OAuth.

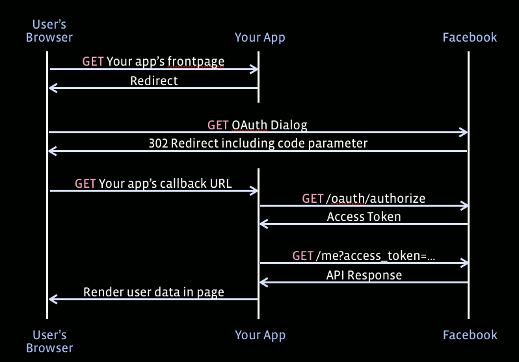
In layman's terms, OAuth lets users login to your web service with accounts(Facebook, Google etc).

## **Terminology:**

* **client**: The user of your API.
* **Resource Owner (api server):** Your API
* **Authorization Server (auth server):** Facebook/Google etc auth server.
* **Authorization grant:** the method by which you authorize a user. we are using Authorization code here.
* **Authorization code:** A code that the auth server returns to the client which can be exchanged for an access token at the api server.
* **Access Token:** A string that identifies a user, usually comes with an expiry period.
* **Consumer Key or APP\_ID:** a public key used by auth server to identify your application.
* **Consumer Secret or APP\_SECRET:** a private key which should be kept confidential.

the below terms has nothing to do with OAuth but are used with OAuth to make it more secure.

* **Timestamp:** a string that tells date and time.
* **Nonce:** a number or string which can be used only once.

[](https://i.stack.imgur.com/CbBw4.png)  
source: <http://smerity.com/>

I will explain the diagram with Facebook login as an example.

background. consider you have done the below, before explaining the diagram.

1. You register an app with Facebook developers portal.
2. Facebook provides you two codes, 1) a secret\_key and 2) an app\_id
3. You designed a button which says Login with Facebook.

now the diagram.

1. Client requests the API server.
2. API server redirects to login page saying. To access the data: please login with facebook to access the page
3. User clicks on the login with Facebook button, a new popup OAuth dialog opens. asking for facebook username and password.
4. User enters his username and password, then allow access to your app. **auth server** redirects the user to your website with a code as parameter in URL.
5. **API Server** is called on the **step 4**, **API server** captures code from URL.
6. **API server** call **auth server** with the provided client\_secret
7. **Auth server** returns to the access token for the user to the API Server.
8. **API server** asks **auth server** for user information for the given access token.
9. **Auth Server** returns details about user, profile pic, email etc.
10. **API server** identifies the user, sends him the response along with access token.
11. **client** sends the access token to the **api server** on next request.
12. **API server** checks if access token is valid and respond.
13. When access token is expired, **client** is asked to login again.

Now, **How does this secure your api?**

Make the portions which need security as login required to access them. if the client who makes the request is not logged in to your api, send him to step 2 of the diagram.

**So what is nonce? timestamp?**

If someone steal an access token, he can get access to **API server** as long as the access token expires. So when the user requests a page, server sends him back a nonce which is stored in the server. the client signs the request with the recieved nonce and complete the request. as the nonce is only used once, server deletes the nonce. when an attacker grabs the nonce, and make a fake request to the server, server rejects the request as the one time number is invalid as its used already.

TimeStamp is used identify the time the token or nonce is created which is used to expire the token or nonce in a limited time frame (1-2seconds), the time needed for a request to complete.

**OkHttp Implementation**

|  |
| --- |
| OkHttp has HTTP/2, a built-in response cache, web sockets, and a simpler API.  It’s got better defaults and is easier to use efficiently.  It’s got a better URL model, a better cookie model, a better headers model and a better call model.  OkHttp makes canceling calls easy.  OkHttp has carefully managed TLS defaults that are secure and widely compatible.  Okhttp works with Retrofit, which is a brilliant API for REST. It also works with Okio, which is a great library for data streams.  OkHttp is a small library with one small dependency (Okio) and is less code to learn. OkHttp is more widely deployed, with a billion Android 4.4+ devices using it internally. |

OkHttp can be customized for every request easily — like timeout customization, etc. for each request.

OkHttp perseveres when the network is troublesome: it will silently recover from common connection problems.

If your service has multiple IP addresses OkHttp will attempt alternate addresses if the first connect fails.

Complete analytics of any request can be obtained. You can know bytes sent, bytes received, and the time taken on any request. These analytics are important so that you can find the data usage of your application and the time taken for each request, so you can identify slow requests.

Using OkHttp is easy. Its request/response API is designed with fluent builders and immutability.

It supports both synchronous blocking calls and async calls with callbacks.

OkHttp supports Android 2.3 and above. For Java, the minimum requirement is 1.7.

Top of Form

Bottom of Form

To send a synchronous GET request we need to build a Request object based on a URL and make a Call. After its execution we get back an instance of Response:

@Test

public void whenGetRequest\_thenCorrect() throws IOException {

Request request = new Request.Builder()

.url(BASE\_URL + "/date")

.build();

Call call = client.newCall(request);

Response response = call.execute();

assertThat(response.code(), equalTo(200));

}

Use *Call.cancel()* to stop an ongoing call immediately. If a thread is currently writing a request or reading a response, an *IOException* will be thrown.

Use this to conserve the network when a call is no longer necessary; for example when your user navigates away from an application:

* the **Connection Timeout** (http.connection.timeout) – the time to establish the connection with the remote host
* the **Socket Timeout** (http.socket.timeout) – the time waiting for data – after establishing the connection; maximum time of inactivity between two data packets

The first two parameters – the connection and socket timeouts – are the most important. However, setting a timeout for obtaining a connection is definitely important in high load scenarios, which is why the third parameter shouldn't be ignored.

**Types of API**

OS API – win32 API to get file system, network devices, UI elements

Library API – users API – logging API, logging library exposes logging Api

Remote API – user component->order proxy->order component, uses proprietry protocol like DCOM,.NET and Java RMI

**Web API** - my webApp->weather App,uses standard protocol like http etc

Platform agnostic - .net can interact with pythin api

Stanard protoclols like http

Usually request/response

Diff by request format,contents,response format and contents

* graphQL – needs dev work,needs optimization
* grpc – uses http/2,protobuf,supports bi-directional and streaming message

Eg:

service HelloService{

Rpc sayHello(HelloRequest) returns (HelloResponse);}

message HelloRequest

{

String greeting = 1;

}

message HelloResponse

{

String reply = 1;

}

Quite performant,requires special libraries at both ends

* Beeceptor to mock api endpoints
* Fully-qualified url should have .api/version/entity/id of entity/subentity
* id of entity – good for get /put/delete, but not post
* Eg: api/v1/order/18/items

**Query parameters**

* /api/v1/orders?fromDate=12/12/2018&toDate=2/2/2019
* /api/v1/order/17/items?startsWith=a
* /api/v1/order/17/items?user=john&date=12/12/2018
* Id,subentity and query parms are optional,I,e 17,items and qp are optional here
* WebDAV extends the standard for document editing
* 207 – multi status – used for cases where multiple entiites are handled

**API Documentation**

OpenAPI(also called as swagger) is default for documentation

Pettstire.swagger.io

Inspector.swagger.io

**Version** – always add version support policy

Including version in api in 3 ways

* /api/v1/order/17/data/v20/query
* Put version in header(most correct form)

Accept:application/vnd.name.v1+json

* In QP like example.com/users/1234?version=4 – don’t use it

**Authentication(AuthN)** – who is the user

* Authorization(AuthZ) - what is he allowed to do
* Auth server shld be familiar with resource server, so resource server should register itself with auth server(app registration).
* Auth server tells the Resource server,,who the user is and what is he allowed to do using specific callback url,
* Client-id and client-secret must be passed to auth-server while asking for authorizations,so auth-server will know which app is requesting user-authorization

A screenshot of a computer

AI-generated content may be incorrect.

* Here jwt is only the access token, which contains data to authenticate the user,
* Jwt – header,payload and signature
* { “alg”:”HS256”,

“typ”:”JWT”

}

{“sub”:”123456789”,

“name”:”JohnDoe”,

“admin”: true

}

All 3 are encoded with base64 and cocn wid dot(.)

* Resouece server checks sign and reads payload,JWT(access token) should be sent with *Authorization:bearer* header

JWT can be sent in body/req parm(not recomm)

**Performance**

* Async functions is popular way to do
* Caching is other way to do ,set expire and invalidation
* Rate limit – limit max concurrent requests the API handles,
* Quota – limits no of request a specific client can make

**Monitoring**

Req/sec

#no of failures

Latency

CPU

#no of users

# no of sessions

Geo distribution

RAM  
**Patterns**

HATEOAS – hyper media as engine of Application state.

* Each REST request returns related resources
* Clients should not have prior knowledge about other resources
* Client should call route endpoint and all other resources shld be accessed from the route endpoint

API discovery and gateway

* Solves loose coupling challenge of services
* Discovery can be done using tools like Consul
* Gateway is done using public clocud,but on-premise is challenging and provides authn/authz,monitoring,routing,policies,LB