What is an SLA for software testing?

An SLA – Service-Level Agreement – is a written contract agreement between the quality team (provider) and development team (customer) to ensure everyone understands the engagement, service of work, and expectations from the provider. If specific conditions are not met there are consequences and often the quality of the product suffers.

### Why do we need an SLA?

In the real world of software development, you will discover that the combination of quality, ownership, and accountability is challenging. The development teams depend on the quality services to meet their quality objectives. The best way to improve ownership and accountability is to provide an SLA that ensures 100 percent alignment of quality goals, engagement, services, and expected outcomes. The exact metrics for each SLA vary depending on the customer quality goals and key performance indicators (KPIs). The agreement aims to establish a mutual understanding of services, areas prioritized, responsibilities, guarantees, and warranties provided by the service provider. The communication of the SLA is critical to the execution and completion of agreements at hand. It builds trust and awareness about your SLA for software testing among internal and external stakeholders.

### How would an SLA help your QA team?

Think about what matters to the QA team – the stakeholders: does the application meet everyone’s expectations from a quality standpoint?

The QA team needs to 1) understand everyone’s expectations; and 2) determine the current state of the application health.

That is the beauty of a well-written SLA. It helps the QA team decide whether to invest time in making the application faster and reduce the cost of release, increase quality, and enhance resilience if, in fact, those elements are what’s expected. It provides guidance on where to focus the time such as reducing testing technical debt, automating the right things, and speeding up testing in the CI pipeline, standards and processes.

The purpose of the agreement is to identify metrics and target measurable quality results that indicate the health of the application. More importantly, the SLA is written to enhance the communication and relationship between the customer and the QA team.

### Summary

Testing is a critical component of the success of any software development project. The SLA is a well thought-out agreement between provider and customer to protect both parties in the event that disputes arise, and to avoid misunderstandings of quality ownership.The SLA can save a considerable amount of time and money for both the provider and the customer. It will pay off in the long-term and supports good communication and relationships between the two parties.

Before we go on to define the SLAs, it's important to define the Key Result Areas (KRAs). These are broad level areas where the SLAs will be measured and they could be in areas like governance, process, resources/staff and transition. Once these are defined, we can define the SLAs within each KRA. It's Important to choose SLAs which are relevant to the engagement (managed service, co-sourced, staff augmentation) or the type of testing (functional/automation/performance/SOA etc.).

In any new engagement where SLAs are defined for the first time, there will invariably be questions about the targets like how do we determine these targets? In such situations, it's always important to define a "Nursery Period". The purpose of this "Nursery Period" is to benchmark the targets for those SLAs where a period of demonstration is required before it can be set. At the end of this exercise, all SLAs should be specific, quantifiable and measurable.

To continue the relevance of SLAs, it's important that it is reviewed regularly. I think it should be reviewed every quarter. The SLA's that do not serve their purpose need to be eliminated. "Raise the bar" for SLAs which are being consistently met for consecutive periods.

Lastly, it is extremely important to create awareness about the SLAs amongst internal and external stakeholders and project participants. This is necessary so that everybody understands the SLAs and its objectives. Communication of the SLAs is very critical and essential to the successful execution and completion of the testing assignment at hand.

|  |  |  |
| --- | --- | --- |
| **No.** | **SOAP** | **REST** |
| 1) | SOAP is a **protocol**. | REST is an **architectural style**. |
| 2) | SOAP stands for **Simple Object Access Protocol**. | REST stands for **REpresentational State Transfer**. |
| 3) | SOAP **can't use REST** because it is a protocol. | REST **can use SOAP** web services because it is a concept and can use any protocol like HTTP, SOAP. |
| 4) | SOAP **uses services interfaces to expose the business logic**. | REST **uses URI to expose business logic**. |
| 5) | **JAX-WS** is the java API for SOAP web services. | **JAX-RS** is the java API for RESTful web services. |
| 6) | SOAP **defines standards**to be strictly followed. | REST does not define too much standards like SOAP. |
| 7) | SOAP **requires more bandwidth** and resource than REST. | REST **requires less bandwidth** and resource than SOAP. |
| 8) | SOAP **defines its own security**. | RESTful web services **inherits security measures** from the underlying transport. |
| 9) | SOAP **permits XML** data format only. | REST **permits different** data format such as Plain text, HTML, XML, JSON etc. |
| 10) | SOAP is **less preferred** than REST. | REST **more preferred** than SOAP. |

<https://rapidapi.com/blog/api-vs-web-service/>

Web Services and API:

What is a Web Service?

A Web service is a way for two machines to communicate with each other over a network.

A web server running on a computer listens for requests from other computers. When a request from another computer is received, over a network, the Web service returns the requested resources. This resource could be JSON, XML, an HTML file, Images, Audio Files, etc.

It’s important to note the requirement of the request being made over a network.

What is an API?

An API, or Application Programming Interface, is a set of definitions and protocols that allow one application to communicate with another application.

In general, when we speak about APIs, we are likely speaking about web APIs [APIs that are accessible over the internet]. This is not always the case though. APIs can be exposed through local files (such as a JAR file in a Java program, .H file in C/C++ programs, etc.) to allow two local applications to communicate with each other. This doesn’t require a network as the two applications are communicating within a single device.

So What’s the Difference?

You might be wondering to yourself, APIs and Web services sound like the same thing. It’s a way for two computers to communicate with each other over the internet, right? Well, not quite.

As we mentioned in the section about “What is an API?,” not all APIs are accessible over the internet(a network), while Web Services must always be accessed through a network. That’s the difference right there.

*All Web Services are APIs, but not all APIs are Web services.*

Is a REST API a Web Service?

The short answer? Yes, REST APIs are a type of Web Service APIs.

A REST API is a standardized architecture style for creating a Web Service API.

One of the requirements to be a REST API is the utilization of HTTP methods to make a request over a network.

REST was officially defined by computer scientist Roy Fielding in 2000 during his Ph.D. dissertation. It essentially changed the way applications are built. The implementation of the frontend “client” can be built completely independently from the backend “server.”

A REST request from the client to the server usually consists of the following components:

1. URL Path [https://api.example.com/user]
2. HTTP Method [GET, PUT, POST, PATCH, DELETE]
3. Header – (optional) additional information that the client needs to pass along in the request such as Authorization credentials, Content-Type of the body, User-Agent to define what type of application is making the request, and more]
4. Parameters – (optional) variable fields that alter how the resource will be returned.
5. Body – (optional) contains data that needs to be sent to the server.

Here’s How the REST API Works:

Let’s say that you want to see what your best friend posted on Instagram. To do this, you need to go on the app and open up your friends Instagram page.

In this example, your Instagram app [the client], would make a request to Instagram’s server [the server] to request your friend’s Instagram profile. This request would be a GET request to the /users endpoint and in the parameters of the request your friend’s account ID would be included.

HTTP Method: GET

URL: https://api.instagram.com/v1/users/

Parameters: user={best\_friends\_user\_id}

[Disclaimer: This is a very simplified example of a request to the Instagram API]

In the same way that you use a GET request to retrieve data, a POST request would be used to create data on a platform. So let’s use the example of posting an image to Instagram. This request would be a POST request to the /media endpoint with a body of the image and parameters with your caption.

HTTP Method: POST

URL: https://api.instagram.com/v1/media/

Parameters: caption={my\_great\_caption}&user={my\_user\_id}

Body: Image to upload

[Disclaimer: This is not an accurate example request to the Instagram API. The post media endpoint is a private endpoint.]

Benefits of REST APIs

The reason REST is so great is that it offers a standardized methodology for making requests to an API. Once you learn one REST API, other REST APIs are going to function in a similar way.

If an API is available over the internet, there is no need to install additional software within your application. You can access the data from any application that is connected to the same network as the API.

With separated development on the client and server, the client code can be updated without affecting the server, and the server code can be updated without affecting the server. This is assuming the changes are developed in a backward-compatible way.

The Core Differences

If you got a little lost reading about APIs and web services, here is a quick summary of what we covered above.

APIs are application interfaces, meaning that one application is able to interact with another application in a standardized way.

Web services are a type of API, which must be accessed through a network connection.

REST APIs are a standardized architecture for building web APIs using HTTP methods.

## **What is a WSDL?**

WSDL, or Web Service Description Language, is an XML based definition language. It’s used for describing the functionality of a SOAP based web service.

WSDL files are central to testing SOAP-based services. SoapUI uses WSDL files to generate test requests, assertions and mock services.

WSDL files define various aspects of SOAP messages:

* Whether any element or attribute is allowed to appear multiple times
* The required or optional elements and attributes
* A specific order of elements, if it is required

You may consider a WSDL file as a contract between the provider and the consumer of the service. SoapUI supports 1.1 version of the WSDL specification and corresponding bindings for SOAP versions 1.1 and 1.2.

A WSDL can contain any number of services (the bindings). A binding exposes an interface for the specified protocol. In the example above, the WSDL file exposes two bindings: one for SOAP 1.1 (“CurrencyConverterSoap”) and one for SOAP 1.2 (“CurrencyConverterSoap12”).

**Tip:** SoapUI saves the WSDL file to a cache to avoid unnecessary network requests when you work with the project. If you want SoapUI to always use a remote WSDL file, set the Cache Definition project property to False.

Quick Tip: WSDL file should be read from bottom to top

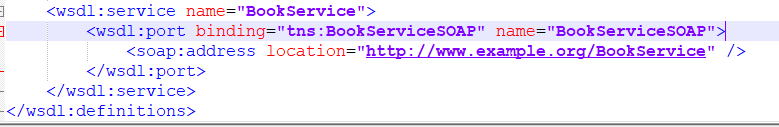
A few points to note before we look at a WSDL file:

* XML message structure can be defined inside a WSDL itself
* We can have an XSD (*XSD* is XML itself. Its purpose is to define the structure of Req and Response message) as a separate file and import into a WSDL
* Predominantly used for SOAP web services

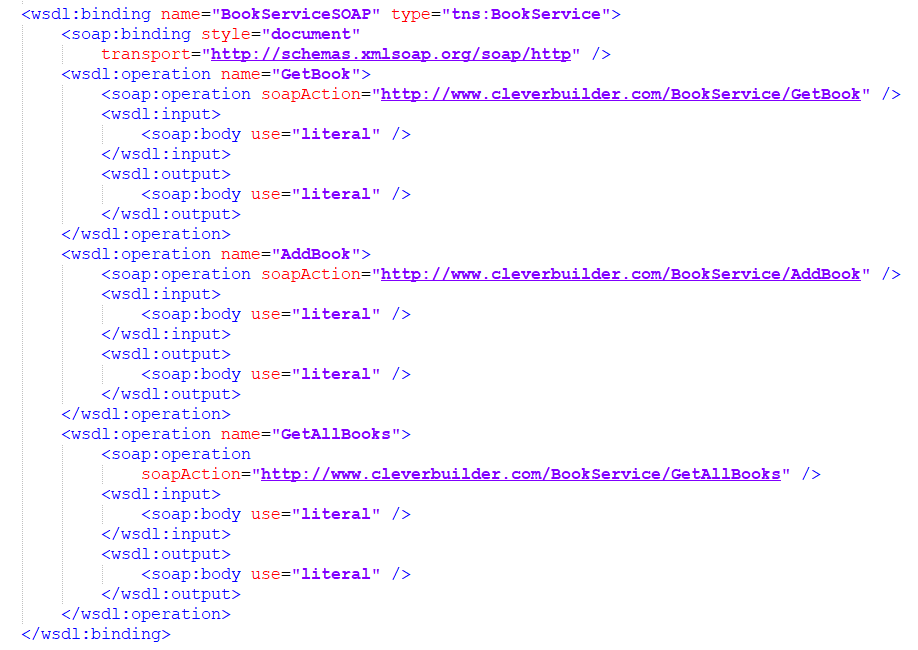
**note that a WSDL file should be read from bottom to top (I learned from experience) as from element “Service > binding > PortType > operation > input(req) / Output (resp) > Message > types (fields) details of input and output**“.



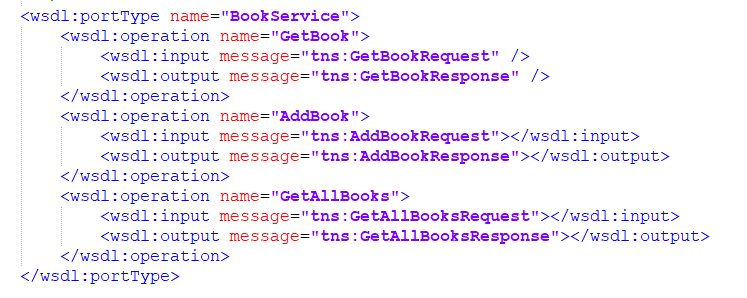
1. Let's start with the most bottom element: “Service”. The service element shows the service name and where it can be accessed from — in other words, its endpoint



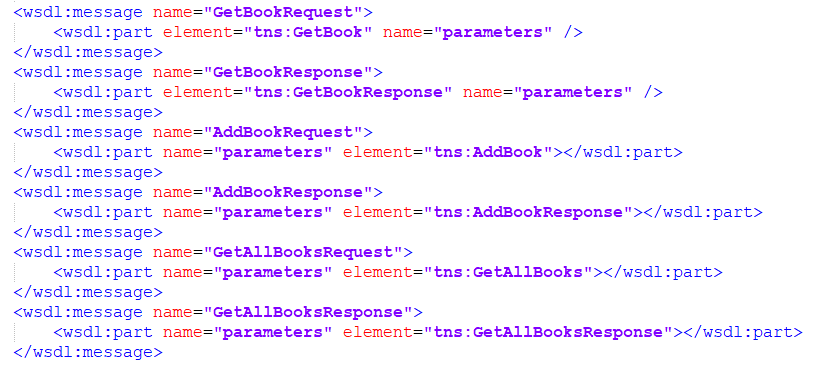
2. The binding element defines exactly how each operation will take place over the network. In this case, we are using SOAP.



3. A WSDL portType is used to combine multiple messages (e.g. input, output) into a single operation. Here, we define three synchronous (input/output) operations  
and the messages that must be used for each.



4. message element is used to define a message exchanged between a web service, consisting of zero or more parts.



5.The types element defines the data types (XML elements) that are used by the web service.



**Info for beginners in IBM Integration bus:**By default, a WSDL file is mandatory if you are using SOAP input-node/Req Node, but if you don’t want to validate input req or receive generic input from multiple front-ends/send to multiple backends on the same flow, you can use SOAP nodes without WSDL by changing “Operation mode” to Gateway mode.

A SOAP webservice can be implemented in IIB with both SOAP and HTTP nodes.

HTTP Methods:

POST:

The POST verb is most-often utilized to \*\*create\*\* new resources. In particular, it's used to create subordinate resources. That is, subordinate to some other (e.g. parent) resource. In other words, when creating a new resource, POST to the parent and the service takes care of associating the new resource with the parent, assigning an ID (new resource URI), etc.

On successful creation, return HTTP status 201, returning a Location header with a link to the newly-created resource with the 201 HTTP status.

POST is neither safe nor idempotent. It is therefore recommended for non-idempotent resource requests. Making two identical POST requests will most-likely result in two resources containing the same information.

GET:

The HTTP GET method is used to \*\*read\*\* (or retrieve) a representation of a resource. In the “happy” (or non-error) path, GET returns a representation in XML or JSON and an HTTP response code of 200 (OK). In an error case, it most often returns a 404 (NOT FOUND) or 400 (BAD REQUEST).

According to the design of the HTTP specification, GET (along with HEAD) requests are used only to read data and not change it. Therefore, when used this way, they are considered safe. That is, they can be called without risk of data modification or corruption—calling it once has the same effect as calling it 10 times, or none at all. Additionally, GET (and HEAD) is idempotent, which means that making multiple identical requests ends up having the same result as a single request.

Do not expose unsafe operations via GET—it should never modify any resources on the server.

PUT:

PUT is most-often utilized for \*\*update\*\* capabilities, PUT-ing to a known resource URI with the request body containing the newly-updated representation of the original resource.

However, PUT can also be used to create a resource in the case where the resource ID is chosen by the client instead of by the server. In other words, if the PUT is to a URI that contains the value of a non-existent resource ID. Again, the request body contains a resource representation. Many feel this is convoluted and confusing. Consequently, this method of creation should be used sparingly, if at all.

Alternatively, use POST to create new resources and provide the client-defined ID in the body representation—presumably to a URI that doesn't include the ID of the resource (see POST below).

On successful update, return 200 (or 204 if not returning any content in the body) from a PUT. If using PUT for create, return HTTP status 201 on successful creation. A body in the response is optional—providing one consumes more bandwidth. It is not necessary to return a link via a Location header in the creation case since the client already set the resource ID.

PUT is not a safe operation, in that it modifies (or creates) state on the server, but it is idempotent. In other words, if you create or update a resource using PUT and then make that same call again, the resource is still there and still has the same state as it did with the first call.

If, for instance, calling PUT on a resource increments a counter within the resource, the call is no longer idempotent. Sometimes that happens and it may be enough to document that the call is not idempotent. However, it's recommended to keep PUT requests idempotent. It is strongly recommended to use POST for non-idempotent requests.

* [PATCH](https://www.restapitutorial.com/lessons/httpmethods.html#patch)PATCH is used for \*\*modify\*\* capabilities. The PATCH request only needs to contain the changes to the resource, not the complete resource.

This resembles PUT, but the body contains a set of instructions describing how a resource currently residing on the server should be modified to produce a new version. This means that the PATCH body should not just be a modified part of the resource, but in some kind of patch language like JSON Patch or XML Patch.

PATCH is neither safe nor idempotent. However, a PATCH request can be issued in such a way as to be idempotent, which also helps prevent bad outcomes from collisions between two PATCH requests on the same resource in a similar time frame. Collisions from multiple PATCH requests may be more dangerous than PUT collisions because some patch formats need to operate from a known base-point or else they will corrupt the resource. Clients using this kind of patch application should use a conditional request such that the request will fail if the resource has been updated since the client last accessed the resource. For example, the client can use a strong ETag in an If-Match header on the PATCH request.

DELETE:

Delete is pretty easy to understand. It is used to \*\*delete\*\* a resource identified by a URI.

On successful deletion, return HTTP status 200 (OK) along with a response body, perhaps the representation of the deleted item (often demands too much bandwidth), or a wrapped response (see Return Values below). Either that or return HTTP status 204 (NO CONTENT) with no response body. In other words, a 204 status with no body, or the JSEND-style response and HTTP status 200 are the recommended responses.

HTTP-spec-wise, DELETE operations are idempotent. If you DELETE a resource, it's removed. Repeatedly calling DELETE on that resource ends up the same: the resource is gone. If calling DELETE say, decrements a counter (within the resource), the DELETE call is no longer idempotent. As mentioned previously, usage statistics and measurements may be updated while still considering the service idempotent as long as no resource data is changed. Using POST for non-idempotent resource requests is recommended.

There is a caveat about DELETE idempotence, however. Calling DELETE on a resource a second time will often return a 404 (NOT FOUND) since it was already removed and therefore is no longer findable. This, by some opinions, makes DELETE operations no longer idempotent, however, the end-state of the resource is the same. Returning a 404 is acceptable and communicates accurately the status of the call.

[**HEAD**](https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/HEAD)

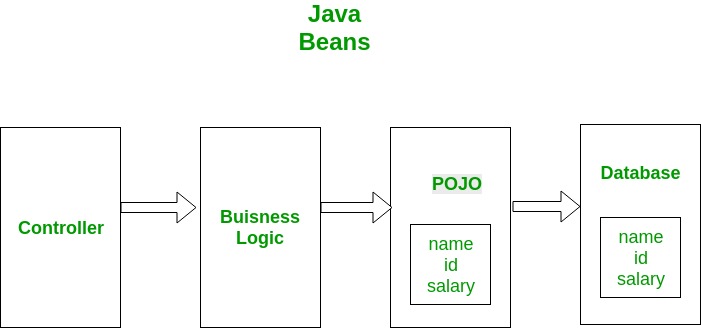
The HEAD method asks for a response identical to that of a GET request, but without the response body.

It's worth pointing out that not every endpoint that supports GET will support HEAD - it completely depends on the API you're testing.

## **OPTIONS**

Last but not least we have OPTIONS requests. OPTIONS requests are one of my favorites, though not as widely used as the other HTTP methods. In a nutshell, an OPTIONS request should **return data describing what other methods and operations the server supports** at the given URL.

OPTIONS requests are more loosely defined and used than the others, making them a good candidate to **test for fatal API errors**. If an API isn't expecting an OPTIONS request, it's good to put a test case in place that verifies failing behavior.



POJO is an object which encapsulates Business Logic. Following image shows a working example of POJO class. Controllers interact with your business logic which in turn interact with POJO to access the database. In this example a database entity is represented by POJO. This POJO has the same members as database entity.

Beans are special type of Pojos. There are some restrictions on POJO to be a bean.

* All JavaBeans are POJOs but not all POJOs are JavaBeans.
* Serializable i.e. they should implement Serializable interface. Still, some POJOs who don’t implement Serializable interface are called POJOs because Serializable is a marker interface and therefore not of much burden.
* Fields should be private. This is to provide the complete control on fields.
* Fields should have getters or setters or both.
* A no-arg constructor should be there in a bean.
* Fields are accessed only by constructor or getter setters.

|  |  |
| --- | --- |
| POJO | Java Bean |
| It doesn’t have special restrictions other than those forced by Java language. | It is a special POJO which have some restrictions. |
| It doesn’t provide much control on members. | It provides complete control on members. |
| It can implement Serializable interface. | It should implement serializable interface. |
| Fields can be accessed by their names. | Fields are accessed only by getters and setters. |
| Fields can have any visiblity. | Fields have only private visiblity. |
| There may/may-not be a no-arg constructor. | It must have a no-arg constructor. |
| It is used when you don’t want to give restriction on your members and give user complete access of your entity | It is used when you want to provide user your entity but only some part of your entity. |

* **Conclusion**
* POJO classes and Beans both are used to define java objects to increase their readability and reusability. POJOs don’t have other restrictions while beans are special POJOs with some restrictions.

# Request header

A **request header** is an [HTTP header](https://developer.mozilla.org/en-US/docs/Glossary/HTTP_header) that can be used in an HTTP request to provide information about the request context, so that the server can tailor the response. For example, the [Accept-\*](https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Accept) headers indicate the allowed and preferred formats of the response. Other headers can be used to supply authentication credentials (e.g.  [Authorization](https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Authorization)), to control caching, or to get information about the user agent or referrer, etc.

Not all headers that can appear in a request are referred to as request headers by the specification. For example, the [Content-Type](https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Type) header is referred to as a [representation header](https://developer.mozilla.org/en-US/docs/Glossary/Representation_header).

In addition, [CORS](https://developer.mozilla.org/en-US/docs/Glossary/CORS) defines a subset of request headers as [simple headers](https://developer.mozilla.org/en-US/docs/Glossary/Simple_header), request headers that are always considered authorized and are not explicitly listed in responses to [preflight](https://developer.mozilla.org/en-US/docs/Glossary/Preflight_request) requests.

The HTTP message below shows a few request headers after a [GET](https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/GET) request:

GET /home.html HTTP/1.1

Host: developer.mozilla.org

User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.9; rv:50.0) Gecko/20100101 Firefox/50.0

Accept: text/html,application/xhtml+xml,application/xml;q=0.9,\*/\*;q=0.8

Accept-Language: en-US,en;q=0.5

Accept-Encoding: gzip, deflate, br

Referer: https://developer.mozilla.org/testpage.html

Connection: keep-alive

Upgrade-Insecure-Requests: 1

If-Modified-Since: Mon, 18 Jul 2016 02:36:04 GMT

If-None-Match: "c561c68d0ba92bbeb8b0fff2a9199f722e3a621a"

Cache-Control: max-age=0

# Response header

A **response header** is an [HTTP header](https://developer.mozilla.org/en-US/docs/Glossary/HTTP_header) that can be used in an HTTP response and that doesn't relate to the content of the message. Response headers, like [Age](https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Age), [Location](https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Location) or [Server](https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Server) are used to give a more detailed context of the response.

Not all headers appearing in a response are categorized as response headers by the specification. For example, the [Content-Type](https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Type) header is a [representation header](https://developer.mozilla.org/en-US/docs/Glossary/Representation_header) indicating the original type of data in the body of the response message (prior to the encoding in the [Content-Encoding](https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Encoding) representation header being applied). However, "conversationally" all headers are usually referred to as response headers in a response message.

The following shows a few response and representation headers after a [GET](https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/GET) request.

200 OK

Access-Control-Allow-Origin: \*

Connection: Keep-Alive

Content-Encoding: gzip

Content-Type: text/html; charset=utf-8

Date: Mon, 18 Jul 2016 16:06:00 GMT

Etag: "c561c68d0ba92bbeb8b0f612a9199f722e3a621a"

Keep-Alive: timeout=5, max=997

Last-Modified: Mon, 18 Jul 2016 02:36:04 GMT

Server: Apache

Set-Cookie: mykey=myvalue; expires=Mon, 17-Jul-2017 16:06:00 GMT; Max-Age=31449600; Path=/; secure

Transfer-Encoding: chunked

Vary: Cookie, Accept-Encoding

X-Backend-Server: developer2.webapp.scl3.mozilla.com

X-Cache-Info: not cacheable; meta data too large

X-kuma-revision: 1085259

x-frame-options: DENY

# Representation header

A **representation header** is an [HTTP header](https://developer.mozilla.org/en-US/docs/Glossary/HTTP_header) that describes the particular representation of the resource sent in an HTTP message body.

Representations are different versions of a particular resource that might be returned from a request. For example, the same data resource might be formatted as XML or JSON, and that resource might then be encoded in one or more compressed formats for sending. Clients specify the formats that they prefer during content negotiation (using Accept-\* headers), and the representation headers tell the client the format of the representation they actually received.

Representation headers may be present in both HTTP request and response messages. If sent as a response to a HEAD request, they describe the body content that would be sent if the resource was actually requested.

Representation headers include: [Content-Type](https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Type), [Content-Encoding](https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Encoding), [Content-Language](https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Language), and [Content-Location](https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Location).

# Fetch metadata request header

A **fetch metadata request header** is an [HTTP request header](https://developer.mozilla.org/en-US/docs/Glossary/Request_header) that provides additional information about the context from which the request originated. This allows the server to make decisions about whether a request should be allowed based on where the request came from and how the resource will be used.

With this information a server can implement a resource isolation policy, allowing external sites to request only those resources that are intended for sharing, and that are used appropriately. This approach can help mitigate common cross-site web vulnerabilities such as [CSRF](https://developer.mozilla.org/en-US/docs/Glossary/CSRF), [Cross-site scripting ('XSSI') attacks](https://developer.mozilla.org/en-US/docs/Glossary/Cross-site_scripting), timing attacks, and cross-origin information leaks.

These headers are prefixed with Sec-, and hence have [forbidden header names](https://developer.mozilla.org/en-US/docs/Glossary/Forbidden_header_name). As such, they cannot be modified from JavaScript.

The fetch metadata request headers are:

* [Sec-Fetch-Site](https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Sec-Fetch-Site)
* [Sec-Fetch-Mode](https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Sec-Fetch-Mode)
* [Sec-Fetch-User](https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Sec-Fetch-User)
* [Sec-Fetch-Dest](https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Sec-Fetch-Dest)

<https://www.okta.com/identity-101/authentication-vs-authorization/#:~:text=Authentication%20and%20authorization%20might%20sound,permission%20to%20access%20a%20resource>

**Authentication** confirms that users are who they say they are. **Authorization** gives those users permission to access a resource.

## **What Is Authentication?**

Authentication is the act of validating that users are whom they claim to be. This is the first step in any security process.

Complete an authentication process with:

* **Passwords.**Usernames and passwordsare the most common [authentication factors](https://www.okta.com/products/adaptive-multi-factor-authentication/). If a user enters the correct data, the system assumes the identity is valid and grants access.
* [**One-time pins**](https://www.okta.com/blog/2020/06/what-is-a-one-time-password-otp/)**.** Grant access for only one session or transaction.
* **Authentication apps.**Generate security codes via an outside party that grants access.
* [**Biometrics**](https://www.okta.com/blog/2020/07/biometric-authentication/)**.**A user presents a fingerprint or eye scan to gain access to the system.

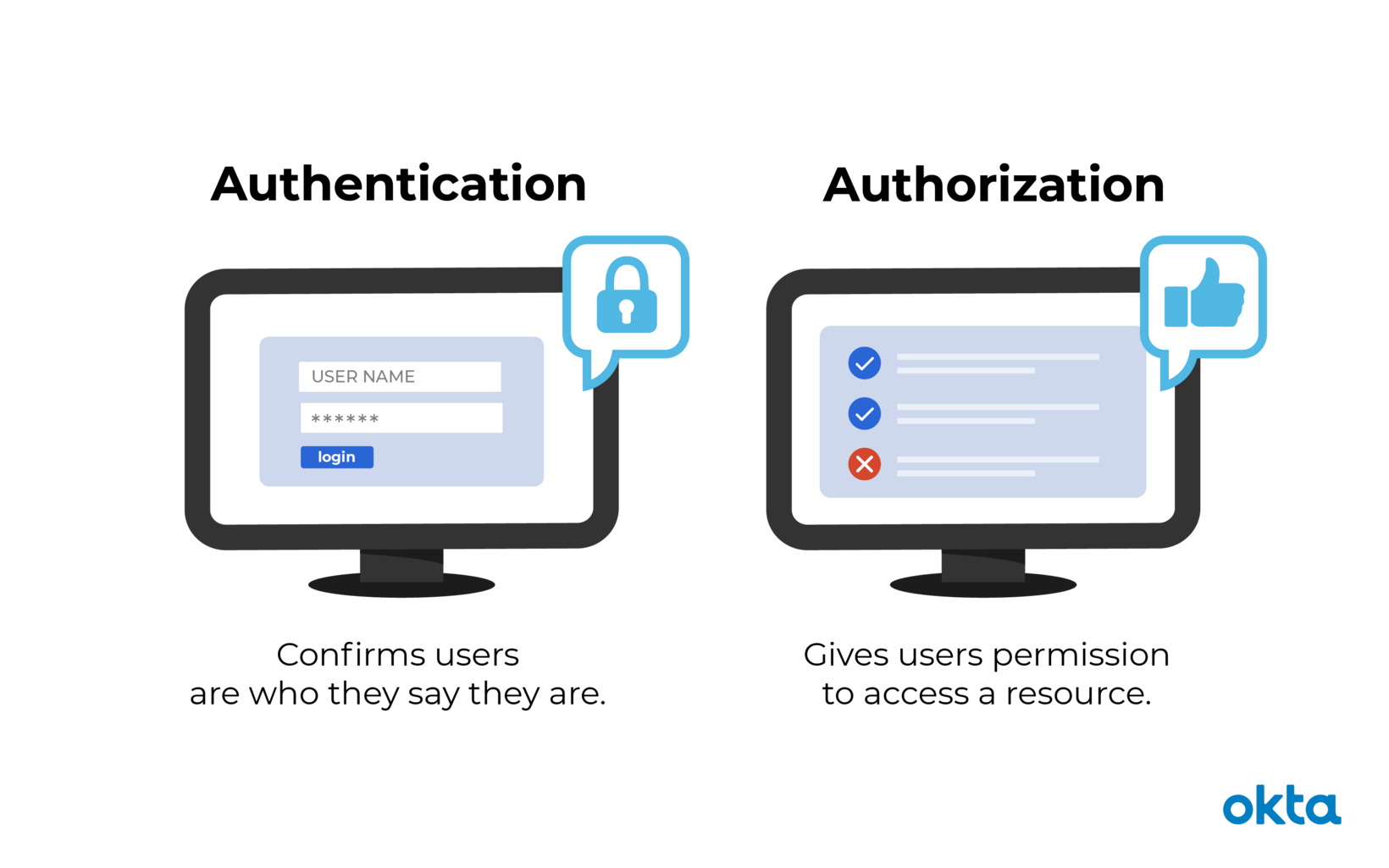
In some instances, systems require the successful verification of more than one factor before granting access. This multi-factor authentication (MFA) requirement is often deployed to increase security beyond what passwords alone can provide.

## **What Is Authorization?**

Authorization in system security is the process of giving the user permission to access a specific resource or function. This term is often used interchangeably with access control or client privilege.

Giving someone permission to download a particular file on a server or providing individual users with administrative access to an application are good examples of authorization.

In secure environments, authorization must always follow authentication. Users should first prove that their identities are genuine before an organization’s administrators grant them access to the requested resources.



## **Authentication vs. Authorization**

* **Authentication**, in the form of a key. The lock on the door only grants access to someone with the correct key in much the same way that a system only grants access to users who have the correct credentials.
* **Authorization,**in the form of permissions. Once inside, the person has the authorization to access the kitchen and open the cupboard that holds the pet food. The person may not have permission to go into the bedroom for a quick nap.

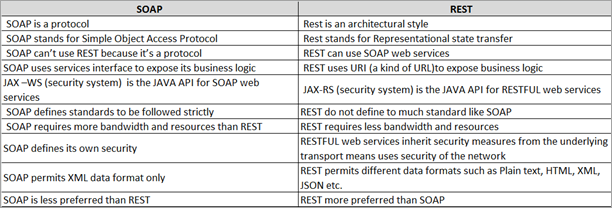
|  |  |
| --- | --- |
| **Authentication** | **Authorization** |
| **What does it do?** | Verifies credentials | Grants or denies permissions |
| **How does it work?** | Through passwords, biometrics, one-time pins, or apps | Through settings maintained by security teams |
| **Is it visible to the user?** | Yes | No |
| **It is changeable by the user?** | Partially | No |
| **How does data move?** | Through ID tokens | Through access tokens |

<https://www.linkedin.com/pulse/rest-api-automation-using-http-client-amit-tayade/>

<https://www.softwaretestinghelp.com/web-services-testing-using-apache-http-client/>

### 202 (Accepted)

A 202 response is typically used for actions that take a long while to process. It indicates that request has been accepted for processing, but the processing has not been completed. The request might or might not be eventually acted upon, or even may be disallowed when processing occurs.



JSON (JavaScript Object Notation) is a lightweight data-interchange format and it completely language independent. It is based on the JavaScript programming language and easy to understand and generate.

Example :

{"Geeks":[

{ "firstName":"Vivek", "lastName":"Kothari" },

{ "firstName":"Suraj", "lastName":"Kumar" },

{ "firstName":"John", "lastName":"Smith" },

{ "firstName":"Peter", "lastName":"Gregory" }

]}

XML (Extensible markup language) was designed to carry data, not to display data. It is a W3C recommendation. Extensible Markup Language (XML) is a markup language that defines a set of rules for encoding documents in a format that is both human-readable and machine-readable. The design goals of XML focus on simplicity, generality, and usability across the Internet. It is a textual data format with strong support via Unicode for different human languages. Although the design of XML focuses on documents, the language is widely used for the representation of arbitrary data structures such as those used in web services.

Example :

<Geeks>

<Geek>

<firstName>Vivek</firstName> <lastName>Kothari</lastName>

</Geek>

<Geek>

<firstName>Suraj</firstName> <lastName>Kumar</lastName>

</Geek>

<Geek>

<firstName>John</firstName> <lastName>Smith</lastName>

</Geek>

<Geek>

<firstName>Peter</firstName> <lastName>Gregory</lastName>

</Geek>

</Geeks>



Namespace:

Name conflicts in XML can easily be avoided using a name prefix.

API Security:

Below, one will find some recommendations to consider prior starting to develop a RESTful API.

* Data Protection
* Transport Layer Security

<https://dzone.com/articles/top-5-rest-api-security-guidelines>

Principles of RESTful API Security Testing

There are only four core principles to performing security tests on RESTful APIs. As is often the case however, these principles can be difficult to put into practice.

The simple principles are as follows, and can be implemented trivially into a web server:

Inputs of an incorrect type must be rejected.

a. Corollary: Inputs that are null (empty), when a null is unacceptable, must be rejected.

2. Inputs of an incorrect size must be rejected.

The more difficult principles require an intimate understanding in the range of acceptable values and users, which can be hard to infer without understanding how a REST API will be consumed.

3. For a given input value, the API must provide the expected output.

This can be easy to test when the input domain and the output range are simple (e.g integers or phone numbers). This becomes extremely difficult when building permissive RESTful APIs that enable users to submit their own content (e.g in a chat application).

4. Input values outside the expected domain must be rejected.

Once again, this is easy when the domain is simple (e.g input values should be integers above zero), but becomes complex when users can supply content (e.g a file upload endpoint could present a significant challenge to secure).

5. For a given user, the API must provide only the data that they are authorized to access.

If permissions are already defined and are resources stratified in accordance with their permission level, this can be easy to implement. In practice however, authorization is a hard problem — with several multi-billion dollar companies (like Okta) around to solve it.

Most APIs aren’t properly tested to ensure they meet this criteria. Because of this, breaches occur frequently and entire industries exist to offer a protection layer on top of APIs.

A well designed APIs should present the first-line of defense against attack, and so effective testing should be a top priority.

<https://medium.com/@SphericalDefense/api-security-testing-34c979687762>

REST API Architectural Constraints

* Difficulty Level : [Basic](https://www.geeksforgeeks.org/basic/)
* Last Updated : 01 Jul, 2020

REST stands for REpresentational State Transfer and API stands for Application Program Interface. REST is a software architectural style that defines the set of rules to be used for creating web services. Web services which follow the REST architectural style are known as RESTful web services. It allows requesting systems to access and manipulate web resources by using a uniform and predefined set of rules. Interaction in REST based systems happen through Internet’s Hypertext Transfer Protocol (HTTP).

A Restful system consists of a:

* client who requests for the resources.
* server who has the resources.

It is important to create REST API according to industry standards which results in ease of development and increase client adoption.

**Architectural Constraints of RESTful API:** There are six architectural constraints which makes any web service are listed below:

* Uniform Interface
* Stateless
* Cacheable
* Client-Server
* Layered System
* Code on Demand

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.

**Uniform Interface:** It is a key constraint that differentiate between a REST API and Non-REST API. It suggests that there should be an uniform way of interacting with a given server irrespective of device or type of application (website, mobile app).  
There are four guidelines principle of Uniform Interface are:

* **Resource-Based:** Individual resources are identified in requests. For example: API/users.
* **Manipulation of Resources Through Representations:** Client has representation of resource and it contains enough information to modify or delete the resource on the server, provided it has permission to do so. Example: Usually user get a user id when user request for a list of users and then use that id to delete or modify that particular user.
* **Self-descriptive Messages:** Each message includes enough information to describe how to process the message so that server can easily analyses the request.
* **Hypermedia as the Engine of Application State (HATEOAS):** It need to include links for each response so that client can discover other resources easily.

**Stateless:** It means that the necessary state to handle the request is contained within the request itself and server would not store anything related to the session. In REST, the client must include all information for the server to fulfill the request whether as a part of query params, headers or URI. Statelessness enables greater availability since the server does not have to maintain, update or communicate that session state. There is a drawback when the client need to send too much data to the server so it reduces the scope of network optimization and requires more bandwidth.

**Cacheable:** Every response should include whether the response is cacheable or not and for how much duration responses can be cached at the client side. Client will return the data from its cache for any subsequent request and there would be no need to send the request again to the server. A well-managed caching partially or completely eliminates some client–server interactions, further improving availability and performance. But sometime there are chances that user may receive stale data.

**Client-Server:** REST application should have a client-server architecture. A Client is someone who is requesting resources and are not concerned with data storage, which remains internal to each server, and server is someone who holds the resources and are not concerned with the user interface or user state. They can evolve independently. Client doesn’t need to know anything about business logic and server doesn’t need to know anything about frontend UI.

**Layered system:** An application architecture needs to be composed of multiple layers. Each layer doesn’t know any thing about any layer other than that of immediate layer and there can be lot of intermediate servers between client and the end server. Intermediary servers may improve system availability by enabling load-balancing and by providing shared caches.

**Code on demand:** It is an optional feature. According to this, servers can also provide executable code to the client. The examples of code on demand may include the compiled components such as Java applets and client-side scripts such as JavaScript.

**Rules of REST API:** There are certain rules which should be kept in mind while creating REST API endpoints.

* REST is based on the resource or noun instead of action or verb based. It means that a URI of a REST API should always end with a noun. Example: /api/users is a good example, but /api?type=users is a bad example of creating a REST API.
* HTTP verbs are used to identify the action. Some of the HTTP verbs are – GET, PUT, POST, DELETE, UPDATE, PATCH.
* A web application should be organized into resources like users and then uses HTTP verbs like – GET, PUT, POST, DELETE to modify those resources. And as a developer it should be clear that what needs to be done just by looking at the endpoint and HTTP method used.

| URI | HTTP verb | Description |
| --- | --- | --- |
| api/users | GET | Get all users |
| api/users/new | GET | Show form for adding new user |
| api/users | POST | Add a user |
| api/users/1 | PUT | Update a user with id = 1 |
| api/users/1/edit | GET | Show edit form for user with id = 1 |
| api/users/1 | DELETE | Delete a user with id = 1 |
| api/users/1 | GET | Get a user with id = 1 |

* Always use plurals in URL to keep an API URI consistent throughout the application.
* Send a proper HTTP code to indicate a success or error status.

**Note :**You can easily use GET and POST but in order to use PUT and DELETE you will need to install method override. You can do this by following below code :

npm install method-override --save

This simply require this package in your code by writing :

var methodOverride = require("method-override");

Now you can easily use PUT and DELETE routes :

app.use(methodOverride("\_method"));

**HTTP verbs:** Some of the common HTTP methods/verbs are described below:

* **GET:** Retrieves one or more resources identified by the request URI and it can cache the information receive.
* **POST:** Create a resource from the submission of a request and response is not cacheable in this case. This method is unsafe if no security is applied to the endpoint as it would allow anyone to create a random resource by submission.
* **PUT:** Update an existing resource on the server specified by the request URI.
* **DELETE:** Delete an existing resource on the server specified by the request URI. It always return an appropriate HTTP status for every request.
* GET, PUT, DELETE methods are also known as Idempotent methods. Applying an operation once or applying it multiple times has the same effect. Example: Delete any resource from the server and it succeeds with 200 OK and then try again to delete that resource than it will display an error message 410 GONE.

REST API Architectural Constraints

* Difficulty Level : [Basic](https://www.geeksforgeeks.org/basic/)
* Last Updated : 01 Jul, 2020

REST stands for REpresentational State Transfer and API stands for Application Program Interface. REST is a software architectural style that defines the set of rules to be used for creating web services. Web services which follow the REST architectural style are known as RESTful web services. It allows requesting systems to access and manipulate web resources by using a uniform and predefined set of rules. Interaction in REST based systems happen through Internet’s Hypertext Transfer Protocol (HTTP).

A Restful system consists of a:

* client who requests for the resources.
* server who has the resources.

It is important to create REST API according to industry standards which results in ease of development and increase client adoption.

**Architectural Constraints of RESTful API:** There are six architectural constraints which makes any web service are listed below:

* Uniform Interface
* Stateless
* Cacheable
* Client-Server
* Layered System
* Code on Demand

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.

**Uniform Interface:** It is a key constraint that differentiate between a REST API and Non-REST API. It suggests that there should be an uniform way of interacting with a given server irrespective of device or type of application (website, mobile app).  
There are four guidelines principle of Uniform Interface are:

* **Resource-Based:** Individual resources are identified in requests. For example: API/users.
* **Manipulation of Resources Through Representations:** Client has representation of resource and it contains enough information to modify or delete the resource on the server, provided it has permission to do so. Example: Usually user get a user id when user request for a list of users and then use that id to delete or modify that particular user.
* **Self-descriptive Messages:** Each message includes enough information to describe how to process the message so that server can easily analyses the request.
* **Hypermedia as the Engine of Application State (HATEOAS):** It need to include links for each response so that client can discover other resources easily.

**Stateless:** It means that the necessary state to handle the request is contained within the request itself and server would not store anything related to the session. In REST, the client must include all information for the server to fulfill the request whether as a part of query params, headers or URI. Statelessness enables greater availability since the server does not have to maintain, update or communicate that session state. There is a drawback when the client need to send too much data to the server so it reduces the scope of network optimization and requires more bandwidth.

**Cacheable:** Every response should include whether the response is cacheable or not and for how much duration responses can be cached at the client side. Client will return the data from its cache for any subsequent request and there would be no need to send the request again to the server. A well-managed caching partially or completely eliminates some client–server interactions, further improving availability and performance. But sometime there are chances that user may receive stale data.

**Client-Server:** REST application should have a client-server architecture. A Client is someone who is requesting resources and are not concerned with data storage, which remains internal to each server, and server is someone who holds the resources and are not concerned with the user interface or user state. They can evolve independently. Client doesn’t need to know anything about business logic and server doesn’t need to know anything about frontend UI.

**Layered system:** An application architecture needs to be composed of multiple layers. Each layer doesn’t know any thing about any layer other than that of immediate layer and there can be lot of intermediate servers between client and the end server. Intermediary servers may improve system availability by enabling load-balancing and by providing shared caches.

**Code on demand:** It is an optional feature. According to this, servers can also provide executable code to the client. The examples of code on demand may include the compiled components such as Java applets and client-side scripts such as JavaScript.

**Rules of REST API:** There are certain rules which should be kept in mind while creating REST API endpoints.

* REST is based on the resource or noun instead of action or verb based. It means that a URI of a REST API should always end with a noun. Example: /api/users is a good example, but /api?type=users is a bad example of creating a REST API.
* HTTP verbs are used to identify the action. Some of the HTTP verbs are – GET, PUT, POST, DELETE, UPDATE, PATCH.
* A web application should be organized into resources like users and then uses HTTP verbs like – GET, PUT, POST, DELETE to modify those resources. And as a developer it should be clear that what needs to be done just by looking at the endpoint and HTTP method used.

| URI | HTTP verb | Description |
| --- | --- | --- |
| api/users | GET | Get all users |
| api/users/new | GET | Show form for adding new user |
| api/users | POST | Add a user |
| api/users/1 | PUT | Update a user with id = 1 |
| api/users/1/edit | GET | Show edit form for user with id = 1 |
| api/users/1 | DELETE | Delete a user with id = 1 |
| api/users/1 | GET | Get a user with id = 1 |

* Always use plurals in URL to keep an API URI consistent throughout the application.
* Send a proper HTTP code to indicate a success or error status.

**Note :**You can easily use GET and POST but in order to use PUT and DELETE you will need to install method override. You can do this by following below code :

npm install method-override --save

This simply require this package in your code by writing :

var methodOverride = require("method-override");

Now you can easily use PUT and DELETE routes :

app.use(methodOverride("\_method"));

**HTTP verbs:** Some of the common HTTP methods/verbs are described below:

* **GET:** Retrieves one or more resources identified by the request URI and it can cache the information receive.
* **POST:** Create a resource from the submission of a request and response is not cacheable in this case. This method is unsafe if no security is applied to the endpoint as it would allow anyone to create a random resource by submission.
* **PUT:** Update an existing resource on the server specified by the request URI.
* **DELETE:** Delete an existing resource on the server specified by the request URI. It always return an appropriate HTTP status for every request.
* GET, PUT, DELETE methods are also known as Idempotent methods. Applying an operation once or applying it multiple times has the same effect. Example: Delete any resource from the server and it succeeds with 200 OK and then try again to delete that resource than it will display an error message 410 GONE.

<https://www.geeksforgeeks.org/rest-api-architectural-constraints/>

What is OAuth?

OAuth is an open-standard authorization protocol or framework that provides applications the ability for “secure designated access.” For example, you can tell Facebook that it’s OK for ESPN.com to access your profile or post updates to your timeline without having to give ESPN your Facebook password. This minimizes risk in a major way: In the event ESPN suffers a breach, your Facebook password remains safe.

OAuth doesn’t share password data but instead uses authorization tokens to prove an identity between consumers and service providers. OAuth is an authentication protocol that allows you to approve one application interacting with another on your behalf without giving away your password.

SAML vs. OAuth

SAML (Security Assertion Markup Language) is an alternative federated authentication standard that many enterprises use for Single-Sign On (SSO). SAML enables enterprises to monitor who has access to corporate resources.

There are many differences between SAML and OAuth. SAML uses XML to pass messages, and OAuth uses JSON. OAuth provides a simpler mobile experience, while SAML is geared towards enterprise security. That last point is a key differentiator: OAuth uses API calls extensively, which is why mobile applications, modern web applications, game consoles, and Internet of Things (IoT) devices find OAuth a better experience for the user. SAML, on the other hand, drops a session cookie in a browser that allows a user to access certain web pages – great for short-lived work days, but not so great when have to log into your thermostat every day.

OAuth 1.0 vs. OAuth 2.0

OAuth 2.0 is a complete redesign from OAuth 1.0, and the two are not compatible. If you create a new application today, use OAuth 2.0. This blog only applies to OAuth 2.0, since OAuth 1.0 is deprecated.

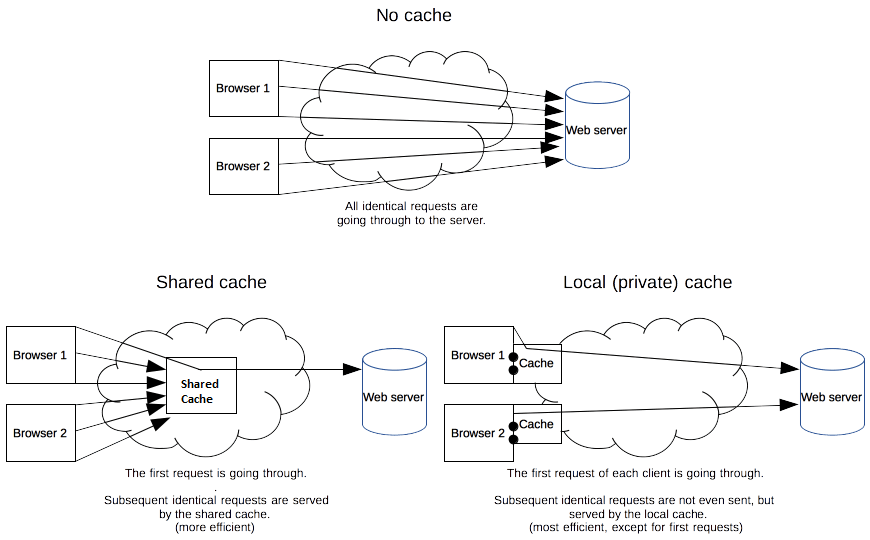
OAuth 2.0 is faster and easier to implement. OAuth 1.0 used complicated cryptographic requirements, only supported three flows, and did not scale.

OAuth 2.0, on the other hand, has six flows for different types of applications and requirements, and enables signed secrets over HTTPS. OAuth tokens no longer need to be encrypted on the endpoints in 2.0 since they are encrypted in transit.

**HTTP Caching:**

The performance of web sites and applications can be significantly improved by reusing previously fetched resources. Web caches reduce latency and network traffic and thus lessen the time needed to display a representation of a resource. By making use of HTTP caching, Web sites become more responsive.

* 1. Private Cache: - is dedicated to a single user
  2. Shared Cache: - is a cache that stores responses for reuse by more than one user



Private browser caches

A private cache is dedicated to a single user. You might have seen "caching" in your browser's settings already. A browser cache holds all documents downloaded via HTTP by the user. This cache is used to make visited documents available for back/forward navigation, saving, viewing-as-source, etc. without requiring an additional trip to the server. It likewise improves offline browsing of cached content.

Shared proxy caches

A shared cache is a cache that stores responses to be reused by more than one user. For example, an ISP or your company might have set up a web proxy as part of its local network infrastructure to serve many users so that popular resources are reused a number of times, reducing network traffic and latency.

**Controlling caching**

The Cache-Control header

The Cache-Control HTTP/1.1 general-header field is used to specify directives for caching mechanisms in both requests and responses. Use this header to define your caching policies with the variety of directives it provides.

**No caching**

The cache should not store anything about the client request or server response. A request is sent to the server and a full response is downloaded each and every time.

***Cache-Control: no-store***

**Cache but revalidate**

A cache will send the request to the origin server for validation before releasing a cached copy.

***Cache-Control: no-cache***

**Private and public caches**

The "public" directive indicates that the response may be cached by any cache. This can be useful if pages with HTTP authentication, or response status codes that aren't normally cacheable, should now be cached.

On the other hand, "private" indicates that the response is intended for a single user only and must not be stored by a shared cache. A private browser cache may store the response in this case.

***Cache-Control: private***

***Cache-Control: public***

**Expiration**

The most important directive here is max-age=<seconds>, which is the maximum amount of time in which a resource will be considered fresh. This directive is relative to the time of the request, and overrides the Expires header (if set). For the files in the application that will not change, you can normally use aggressive caching. This includes static files such as images, CSS files, and JavaScript files, for example.

***Cache-Control: max-age=31536000***

**Validation:**

When using the "must-revalidate" directive, the cache must verify the status of the stale resources before using it and expired ones should not be used. For more details, see the Validation section below.

***Cache-Control: must-revalidate***

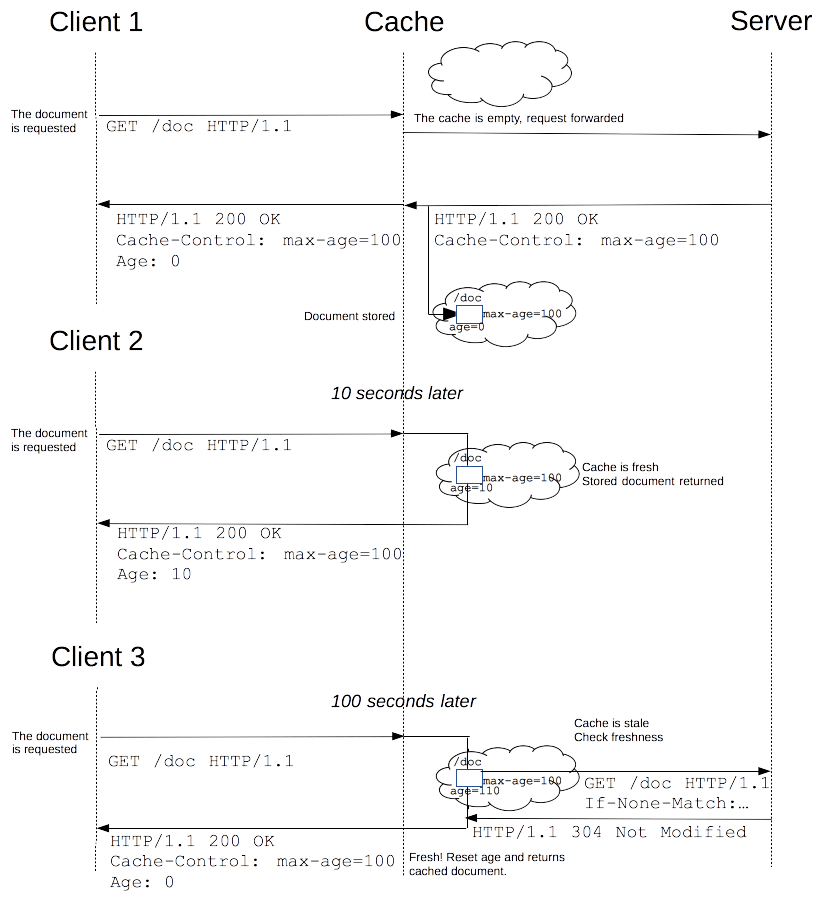
**The Pragma header**

Pragma is an HTTP/1.0 header. Pragma: no-cache is like Cache-Control: no-cache in that it forces caches to submit the request to the origin server for validation, before releasing a cached copy. However, Pragma is not specified for HTTP responses and is therefore not a reliable replacement for the general HTTP/1.1 Cache-Control header.

Pragma should only be used for backwards compatibility with HTTP/1.0 caches where the Cache-Control HTTP/1.1 header is not yet present.

**Freshness**

Once a resource is stored in a cache, it could theoretically be served by the cache forever. Caches have finite storage so items are periodically removed from storage. This process is called cache eviction. On the other side, some resources may change on the server so the cache should be updated. As HTTP is a client-server protocol, servers can't contact caches and clients when a resource changes; they have to communicate an expiration time for the resource. Before this expiration time, the resource is fresh; after the expiration time, the resource is stale. Eviction algorithms often privilege fresh resources over stale resources. Note that a stale resource is not evicted or ignored; when the cache receives a request for a stale resource, it forwards this request with a If-None-Match to check if it is in fact still fresh. If so, the server returns a 304 (Not Modified) header without sending the body of the requested resource, saving some bandwidth.



The freshness lifetime is calculated based on several headers. If a "Cache-Control: max-age=N" header is specified, then the freshness lifetime is equal to N. If this header is not present, which is very often the case, it is checked if an Expires header is present. If an Expires header exists, then its value minus the value of the Date header determines the freshness lifetime.

**Heuristic freshness checking**

If an origin server does not explicitly specify freshness (e.g. using Cache-Control or Expires header) then a heuristic approach may be used.

In this case look for a Last-Modified header. If this header is present, then the cache's freshness lifetime is equal to the value of the Date header minus the value of the Last-modified header divided by 10. The expiration time is computed as follows:

***expirationTime = responseTime + freshnessLifetime - currentAge***

where responseTime is the time at which the response was received according to the browser.

**Using HTTP cookies**

An HTTP cookie (web cookie, browser cookie) is a small piece of data that a server sends to the user's web browser. The browser may store it and send it back with later requests to the same server. Typically, it's used to tell if two requests came from the same browser — keeping a user logged-in, for example. It remembers stateful information for the stateless HTTP protocol.

Cookies are mainly used for three purposes:

1. Session management

Logins, shopping carts, game scores, or anything else the server should remember

1. Personalization

User preferences, themes, and other settings

1. Tracking

Recording and analyzing user behavior

**Creating cookies:**

After receiving an HTTP request, a server can send one or more Set-Cookie headers with the response. The cookie is usually stored by the browser, and then the cookie is sent with requests made to the same server inside a Cookie HTTP header. An expiration date or duration can be specified, after which the cookie is no longer sent. Additional restrictions to a specific domain and path can be set, limiting where the cookie is sent.

**The Set-Cookie and Cookie headers**

The Set-Cookie HTTP response header sends cookies from the server to the user agent. A simple cookie is set like this:

***Set-Cookie: <cookie-name>=<cookie-value>***

This shows the server sending headers to tell the client to store a pair of cookies:

HTTP/2.0 200 OK

Content-Type: text/html

Set-Cookie: yummy\_cookie=choco

Set-Cookie: tasty\_cookie=strawberry

[page content]

Then, with every subsequent request to the server, the browser sends back all previously stored cookies to the server using the Cookie header.

GET /sample\_page.html HTTP/2.0

Host: www.example.org

Cookie: yummy\_cookie=choco; tasty\_cookie=strawberry

Define the lifetime of a cookie

The lifetime of a cookie can be defined in two ways:

1. Session cookies are deleted when the current session ends. The browser defines when the "current session" ends, and some browsers use session restoring when restarting, which can cause session cookies to last indefinitely long.
2. Permanent cookies are deleted at a date specified by the Expires attribute, or after a period of time specified by the Max-Age attribute.

**Restrict access to cookies**

There are a couple of ways to ensure that cookies are sent securely and are not accessed by unintended parties or scripts: the Secure attribute and the HttpOnly attribute.

A cookie with the Secure attribute is sent to the server only with an encrypted request over the HTTPS protocol, never with unsecured HTTP (except on localhost), and therefore can't easily be accessed by a man-in-the-middle attacker. Insecure sites (with http: in the URL) can't set cookies with the Secure attribute. However, do not assume that Secure prevents all access to sensitive information in cookies; for example, it can be read and modified by someone with access to the client's hard disk (or JavaScript if the HttpOnly attribute is not set).

A cookie with the HttpOnly attribute is inaccessible to the JavaScript Document.cookie API; it is sent only to the server. For example, cookies that persist server-side sessions don't need to be available to JavaScript, and should have the HttpOnly attribute. This precaution helps mitigate cross-site scripting (XSS) attacks.

**Security**

Information should be stored in cookies with the understanding that all cookie values are visible to, and can be changed by, the end-user. Depending on the application, it may be desirable to use an opaque identifier which is looked-up by the server or to investigate alternative authentication/confidentiality mechanisms such as JSON Web Tokens.

Ways to mitigate attacks involving cookies:

1. Use the HttpOnly attribute to prevent access to cookie values via JavaScript.
2. Cookies that are used for sensitive information (such as indicating authentication) should have a short lifetime, with the SameSite attribute set to Strict or Lax. (See SameSite cookies, above.) In browsers that support SameSite, this has the effect of ensuring that the authentication cookie is not sent with cross-site requests, so such a request is effectively unauthenticated to the application server.

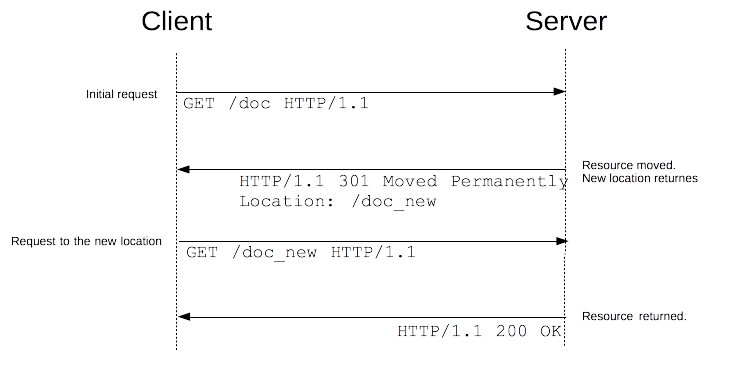
**Redirections in HTTP**

URL redirection, also known as URL forwarding, is a technique to give more than one URL address to a page, a form, or a whole Web site/application. HTTP has a special kind of response, called a HTTP redirect, for this operation.

Redirects accomplish numerous goals:

Temporary redirects during site maintenance or downtime

Permanent redirects to preserve existing links/bookmarks after changing the site's URLs, progress pages when uploading a file, etc.



There are several types of redirects, sorted into three categories:

1. Permanent redirections:

These redirections are meant to last forever. They imply that the original URL should no longer be used and replaced with the new one.

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Text** | **Method handling** | **Typical use case** |
| **301** | Moved Permanently | [GET methods unchanged.](https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/GET) | Reorganization of a Web site. |
| Others may or may not be changed to GET. (See [1]) |
| **308** | Permanent Redirect | Method and body not changed. | Reorganization of a Web site, with non-GET links/operations. |

The specification did not intend to allow method changes, but there are existing user agents that do change their method. 308 was created to remove the ambiguity of the behavior when using non-GET methods.

1. Temporary redirections:

Sometimes the requested resource can't be accessed from its canonical location, but it can be accessed from another place. In this case, a temporary redirect can be used.

Search engine robots and other crawlers don't memorize the new, temporary URL. Temporary redirections are also used when creating, updating, or deleting resources, to show temporary progress pages.

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Text** | **Method handling** | **Typical use case** |
| **302** | Found | [GET methods unchanged.](https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/GET) | The Web page is temporarily unavailable for unforeseen reasons. |
| Others may or may not be changed to GET. (See[2]) |
| **303** | See Other | [GET methods unchanged.](https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/GET) | Used to redirect after a PUT or a POST, so that refreshing the result page doesn't re-trigger the operation. |
| Others *changed* to GET (body lost). |
| **307** | Temporary Redirect | Method and body not changed | The Web page is temporarily unavailable for unforeseen reasons. Better than 302 when non-GET operations are available on the site. |

The specification did not intend to allow method changes, but there are existing user agents that do change their method. 307 was created to remove the ambiguity of the behavior when using non-GET methods.

1. Special redirections:

304 (Not Modified) redirects a page to the locally cached copy (that was stale), and 300 (Multiple Choice) is a manual redirection: the body, presented by the browser as a Web page, lists the possible redirections and the user clicks on one to select it.

|  |  |  |
| --- | --- | --- |
|  |  |  |
| **Code** | **Text** | **Typical use case** |
| **300** | Multiple Choice | [Not many: the choices are listed in an HTML page in the body. Machine-readable choices are encouraged to be sent as Link headers with rel=alternate.](https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Link) |
| **304** | Not Modified | Sent for revalidated conditional requests. Indicates that the cached response is still fresh and can be used. |

Alternative way of specifying redirections

HTTP redirects aren't the only way to define redirections. There are two others:

1. HTML redirections with the <meta> element
2. JavaScript redirections via the DOM

**HTTP range requests:**

HTTP range requests allow to send only a portion of an HTTP message from a server to a client. Partial requests are useful for large media or downloading files with pause and resume functions, for example.

Checking if a server supports partial requests

If the Accept-Ranges is present in HTTP responses (and its value isn't "none"), the server supports range requests. You can check this by issuing a HEAD request with cURL, for example.

curl -I http://i.imgur.com/z4d4kWk.jpg

HTTP/1.1 200 OK

...

Accept-Ranges: bytes

Content-Length: 146515

In this response, Accept-Ranges: bytes indicates that bytes can be used as unit to define a range. Here the Content-Length header is also useful as it indicates the full size of the image to retrieve.

If sites omit the Accept-Ranges header, they likely don't support partial requests. Some sites also explicitly send "none" as a value, indicating no support. In some apps, download managers disable their pause buttons in that case.

curl -I <https://www.youtube.com/watch?v=EwTZ2xpQwpA>

HTTP/1.1 200 OK

...

Accept-Ranges: none

Requesting a specific range from a server:

If the server supports range requests, you can issue such a request by using the Range header. It indicates the part(s) of a document that the server should return.

### [Single part ranges](https://developer.mozilla.org/en-US/docs/Web/HTTP/Range_requests#single_part_ranges)

We can request a single range from a resource. Again, we can test a request by using cURL. The "-H" option will append a header line to the request, which in this case is the Range header requesting the first 1024 bytes.

curl http://i.imgur.com/z4d4kWk.jpg -i -H "Range: bytes=0-1023"

The issued request looks like this:

GET /z4d4kWk.jpg HTTP/1.1

Host: i.imgur.com

Range: bytes=0-1023

The server responses with the [206](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/206) Partial Content status:

HTTP/1.1 206 Partial Content

Content-Range: bytes 0-1023/146515

Content-Length: 1024

...

(binary content)

The [Content-Length](https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Length) header now indicates the size of the requested range (and not the full size of the image). The [Content-Range](https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Range) response header indicates where in the full resource this partial message belongs.

### [Multipart ranges](https://developer.mozilla.org/en-US/docs/Web/HTTP/Range_requests#multipart_ranges)

The [Range](https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Range) header also allows you to get multiple ranges at once in a multipart document. The ranges are separated by a comma.

curl http://www.example.com -i -H "Range: bytes=0-50, 100-150"

The server responses with the [206](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/206) Partial Content status and a [Content-Type](https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Type): multipart/byteranges; boundary=3d6b6a416f9b5 header, indicating that a multipart byterange follows. Each part contains its own Content-Type and Content-Range fields and the required boundary parameter specifies the boundary string used to separate each body-part.

HTTP/1.1 206 Partial Content

Content-Type: multipart/byteranges; boundary=3d6b6a416f9b5

Content-Length: 282

--3d6b6a416f9b5

Content-Type: text/html

Content-Range: bytes 0-50/1270

<!doctype html>

<html>

<head>

<title>Example Do

--3d6b6a416f9b5

Content-Type: text/html

Content-Range: bytes 100-150/1270

eta http-equiv="Content-type" content="text/html; c

--3d6b6a416f9b5--

### [Conditional range requests](https://developer.mozilla.org/en-US/docs/Web/HTTP/Range_requests#conditional_range_requests)

When resuming to request more parts of a resource, you need to guarantee that the stored resource has not been modified since the last fragment has been received.

The [If-Range](https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/If-Range) HTTP request header makes a range request conditional: if the condition is fulfilled, the range request will be issued and the server sends back a [206](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/206) Partial Content answer with the appropriate body. If the condition is not fulfilled, the full resource is sent back, with a [200](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/200) OK status. This header can be used either with a [Last-Modified](https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Last-Modified) validator, or with an [ETag](https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/ETag), but not with both.

If-Range: Wed, 21 Oct 2015 07:28:00 GMT

## [Partial request responses](https://developer.mozilla.org/en-US/docs/Web/HTTP/Range_requests#partial_request_responses)

There are three relevant statuses, when working with range requests:

* In case of a successful range request, the [206](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/206) Partial Content status is sent back from a server.
* In case of a range request that is out of bounds (none of the range values overlap the extent of the resource, i.e first-byte-pos of all ranges is greater than the resource length), the server responds with a [416](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/416) Requested Range Not Satisfiable status.
* In case of no support of range requests, the [200](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/200) OK status is sent back from a server.

Identifying resources on the Web:

The target of an HTTP request is called a "resource", whose nature isn't defined further; it can be a document, a photo, or anything else. Each resource is identified by a Uniform Resource Identifier (URI) used throughout HTTP for identifying resources.

### [URLs](https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/Identifying_resources_on_the_Web#urls)

The most common form of URI is the Uniform Resource Locator ([URL](https://developer.mozilla.org/en-US/docs/Glossary/URL)), which is known as the web address.

https://developer.mozilla.org

https://developer.mozilla.org/en-US/docs/Learn/

https://developer.mozilla.org/en-US/search?q=URL

Any of those URLs can be typed into your browser's address bar to tell it to load the associated page (resource).

A URL is composed of different parts, some mandatory and others are optional. A more complex example might look like this:

http://www.example.com:80/path/to/myfile.html?key1=value1&key2=value2#SomewhereInTheDocument

### [URNs](https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/Identifying_resources_on_the_Web#urns)

A Uniform Resource Name (URN) is a URI that identifies a resource by name in a particular namespace.

urn:isbn:9780141036144

urn:ietf:rfc:7230

The two URNs correspond to

* the book Nineteen Eighty-Four by George Orwell,
* the IETF specification 7230, Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing.

## [Syntax of Uniform Resource Identifiers (URIs)](https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/Identifying_resources_on_the_Web#syntax_of_uniform_resource_identifiers_uris)

### [Scheme or protocol](https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/Identifying_resources_on_the_Web#scheme_or_protocol)

****

http:// is the protocol. It indicates which protocol the browser must use. Usually it is the HTTP protocol or its secured version, HTTPS. The Web requires one of these two, but browsers also know how to handle other protocols such as mailto: (to open a mail client) or ftp: to handle file transfer, so don't be surprised if you see such protocols. Common schemes are:

| **Scheme** | **Description** |
| --- | --- |
| data | [Data URIs](https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/Data_URIs) |
| file | Host-specific file names |
| ftp | [File Transfer Protocol](https://developer.mozilla.org/en-US/docs/Glossary/FTP) |
| http/https | [Hyper text transfer protocol (Secure)](https://developer.mozilla.org/en-US/docs/Glossary/HTTP) |
| javascript | URL-embedded JavaScript code |
| mailto | Electronic mail address |
| ssh | Secure shell |
| tel | telephone |
| urn | Uniform Resource Names |
| view-source | Source code of the resource |
| ws/wss | [WebSocket connections (Secure)](https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API) |

### [Authority](https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/Identifying_resources_on_the_Web#authority)

****

www.example.com is the domain name or authority that governs the namespace. It indicates which Web server is being requested. Alternatively, it is possible to directly use an [IP address](https://developer.mozilla.org/en-US/docs/Glossary/IP_Address), but because it is less convenient, it is not often used on the Web.

### [Port](https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/Identifying_resources_on_the_Web#port)

****

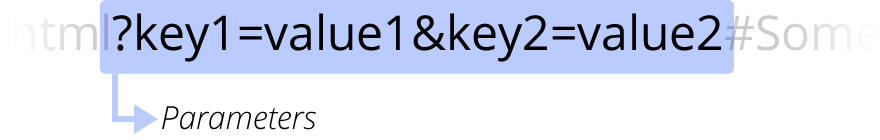
:80 is the port in this instance. It indicates the technical "gate" used to access the resources on the web server. It is usually omitted if the web server uses the standard ports of the HTTP protocol (80 for HTTP and 443 for HTTPS) to grant access to its resources. Otherwise it is mandatory.

### [Path](https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/Identifying_resources_on_the_Web#path)

****

/path/to/myfile.html is the path to the resource on the Web server. In the early days of the Web, a path like this represented a physical file location on the Web server. Nowadays, it is mostly an abstraction handled by Web servers without any physical reality.

### [Query](https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/Identifying_resources_on_the_Web#query)

****

?key1=value1&key2=value2 are extra parameters provided to the Web server. Those parameters are a list of key/value pairs separated with the & symbol. The Web server can use those parameters to do extra stuff before returning the resource to the user. Each Web server has its own rules regarding parameters, and the only reliable way to know how a specific Web server is handling parameters is by asking the Web server owner.

### [Fragment](https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/Identifying_resources_on_the_Web#fragment)

****

#SomewhereInTheDocument is an anchor to another part of the resource itself. An anchor represents a sort of "bookmark" inside the resource, giving the browser the directions to show the content located at that "bookmarked" spot. On an HTML document, for example, the browser will scroll to the point where the anchor is defined; on a video or audio document, the browser will try to go to the time the anchor represents. It is worth noting that the part after the #, also known as fragment identifier, is never sent to the server with the request.

## [Usage notes](https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/Identifying_resources_on_the_Web#usage_notes)

When using URLs in [HTML](https://developer.mozilla.org/en-US/docs/Glossary/HTML) content, you should generally only use a few of these URL schemes. When referring to subresources — that is, files that are being loaded as part of a larger document — you should only use the HTTP and HTTPS schemes. Increasingly, browsers are removing support for using FTP to load subresources, for security reasons.

FTP is still acceptable at the top level (such as typed directly into the browser's URL bar, or the target of a link), although some browsers may delegate loading FTP content to another application.

## [Examples](https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/Identifying_resources_on_the_Web#examples)

https://developer.mozilla.org/en-US/docs/Learn

tel:+1-816-555-1212

git@github.com:mdn/browser-compat-data.git

ftp://example.org/resource.txt

urn:isbn:9780141036144

mailto:help@supercyberhelpdesk.info