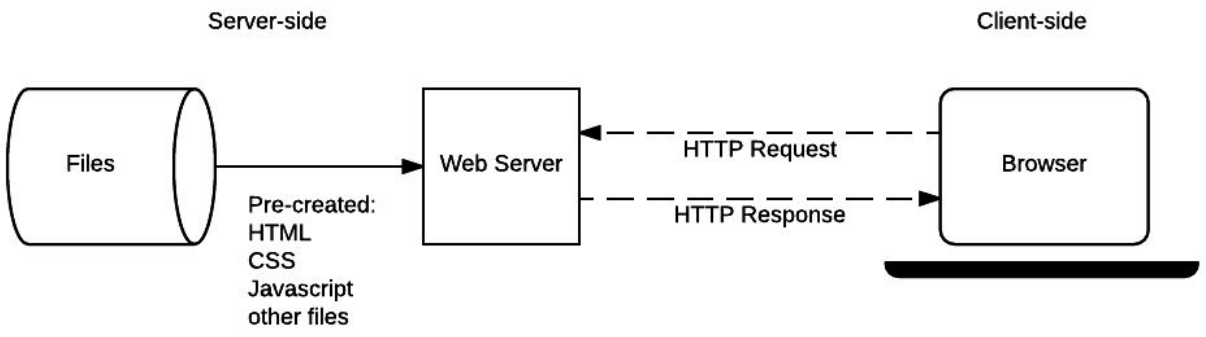
HTTP & HTTP Methods

HTTP (Hypertext Transfer Protocol) is a stateless Application Level transfer protocol for distributed information system. Here the stateless terms refers that the server need not to maintain any user status or information.

HTTP is the most basic foundation for data transfer over the internet since 1990s. HTTP is based on TCP/IP communication protocol. HTTP specifies how client request is sent to the server and how response will be.



HTTP Request & Response Architecture

1) So how does HTTP works?

HTTP is basically a request response protocol for communication between client and server. In most cases browser act as a client while Apache or IIS act as a server. The response comprises of an HTML file along with audio, video, etc and the request status is changed upon the response.

So what does an HTTP request response looks like?

<u>HTTP Request</u>

1. *GET /content/homepage.html HTTP/1.1*
2. *Host: weboniselab.com*

Here, *GET /content/homepage.html is* a required resource from the server.
Host: weboniselab.com shows host name where the request has been made.
Message body is optional.

<u>HTTP Response</u>

1. *HTTP/1.1 200 OK*
2. *Date: Wed, 10 Oct 2018 2:29:00 GMT*
3. *Server: Apache*

4. *Last-Modified: Wed, 10 Oct 2018 2:35:00 GMT*
5. *Transfer-Encoding: chunked*
6. *Content-Type: text/html; charset=UTF-8*
7. *Webpage Content*

Here, *HTTP/1.1 200 OK* is the Status Code.
Server: Apache is the server that responds to the request.

2) HTTP methods

HTTP methods also called as HTTP verbs indicates the desired action to be perform for specific resource. All of these HTTP methods corresponds to CRUD (Create, Read, Update, Delete) operations.
Following are the mostly used HTTP verbs;

HTTP Verbs
POST
GET
PUT
DELETE
PATCH

1) POST

POST is one of the most commonly used HTTP method. POST is used to send data to a server to create/update a resource.The data sent to the server with POST is stored in the request body of the HTTP request. Data such as customer information, file upload using the HTML form.

When executing a `POST` request, the client is actually submitting a new *document* to the remote host. So, a *query* string is not required. Which is why you don't have access to them in your application code.

The following example makes use of POST method to send a form data to the server, which will be processed by a process.cgi and finally a response will be returned:

```
POST /cgi-bin/process.cgi HTTP/1.1
User-Agent: Mozilla
Host: www.testexample.com
Content-Type: text/xml; charset=utf-8
Content-Length: 88
Accept-Language: en-us
Accept-Encoding: gzip, deflate
Connection: Keep-Alive
```

```
<?xml version="1.0" encoding="utf-8"?>
<string xmlns="http://testexample.com/">string</string>
```

The server side script process.cgi processes the passed data and sends the following response:

```
HTTP/1.1 200 OK
Date: Web, 10 Oct 2018 2:28:53 GMT
Server: Apache/2.2.14
Last-Modified: Web, 10 Oct 2018 2:28:53 GMT
Vary: Authorization,Accept
Accept-Ranges: bytes
Content-Length: 88
Content-Type: text/html
Connection: Closed
```

```
<html>
<body>
<h1>Request Processed Successfully</h1>
</body>
</html>
```

2) GET

GET is used to request data from a specified resource.GET is one of the most common HTTP method.When executing a GET request, you ask the server for one, or a set of entities. To allow the client to filter the result, it can use the so called "query string" of the URL. The query string is the part after the “?”.

From the point of view of your application code, you will need to inspect the URI query part to gain access to these values.GET requests should never be used to submit new information to the server. Especially not larger documents.
The following example makes use of GET method to fetch hello.html:

```
GET /hello.htm HTTP/1.1
User-Agent: Mozilla
Host: www.testexample.com
Accept-Language: en-us
Accept-Encoding: gzip, deflate
Connection: Keep-Alive
```

The server response against the above GET request will be as follows:

```
HTTP/1.1 200 OK
Date: Web, 10 Oct 2018 2:28:53 GMT
Server: Apache/2.2.14
Last-Modified: Web, 10 Oct 2018 2:28:53 GMT
Vary: Authorization,Accept
Accept-Ranges: bytes
Content-Length: 88
Content-Type: text/html
Connection: Closed
```

```
<html>
<body>
<h1>Hello, World!</h1>
</body>
</html>
```

2) PUT

Use PUT APIs primarily to update existing resource (if the resource does not exist then API may decide to create a new resource or not). If a new resource has been created by the PUT API, the origin server MUST inform the user agent via the HTTP response code 201 (Created) response and if an existing resource is modified, either the 200 (OK) or 204 (No Content) response codes SHOULD be sent to indicate successful completion of the request.

If the request passes through a cache and the Request-URI identifies one or more currently cached entities, those entries SHOULD be treated as stale. Responses to this method are not cacheable.

Example request URIs:-

HTTP PUT http://www.appdomain.com/users/123
HTTP PUT http://www.appdomain.com/users/123/accounts/456

4) DELETE

As the name applies, DELETE APIs are used to delete resources (identified by the Request-URI).

A successful response of DELETE requests SHOULD be HTTP response code 200 (OK) if the response includes an entity describing the status, 202 (Accepted) if the action has been queued, or 204 (No Content) if the action has been performed but the response does not include an entity.

DELETE operations are idempotent. If you DELETE a resource, it’s removed from the collection of resource. Repeatedly calling DELETE API on that resource will not change the outcome – however calling DELETE on a resource a second time will return a 404 (NOT FOUND) since it was already removed. Some may argue that it makes DELETE method non-idempotent. It’s a matter of discussion and personal opinion.

If the request passes through a cache and the Request-URI identifies one or more currently cached entities, those entries SHOULD be treated as stale. Responses to this method are not cacheable.

Example request URIs:-

HTTP DELETE http://www.appdomain.com/users/123
HTTP DELETE http://www.appdomain.com/users/123/accounts/456

5) PATCH

HTTP PATCH requests are to make partial update on a resource. If you see PUT requests also modify a resource entity so to make more clear – PATCH method is the correct choice for partially updating an existing resource and PUT should only be used if you’re replacing a resource in its entirety.

Please note that there are some challenges if you decide to use PATCH APIs in your application:

Support for PATCH in browsers, servers, and web application frameworks is not universal. IE8, PHP, Tomcat, Django, and lots of other software has missing or broken support for it. Request payload of PATCH request is not straightforward as it is for PUT request. e.g.

HTTP GET /users/1

produces below response:

{id: 1, username: 'admin', email: 'email@example.org'}

A sample patch request to update the email will be like this:

HTTP PATCH /users/1

```
[
{ “op”: “replace”, “path”: “/email”, “value”: “new.email@example.org” }
]
```

There may be following possible operations are per HTTP specification.

```
[
{ "op": "test", "path": "/a/b/c", "value": "foo" },
{ "op": "remove", "path": "/a/b/c" },
{ "op": "add", "path": "/a/b/c", "value": [ "foo", "bar" ] },
{ "op": "replace", "path": "/a/b/c", "value": 42 },
{ "op": "move", "from": "/a/b/c", "path": "/a/b/d" },
{ "op": "copy",  "from": "/a/b/d", "path": "/a/b/e" }
]
```

PATCH method is not a replacement for the POST or PUT methods. It applies a delta (diff) rather than replacing the entire resource.

a) Safe Methods
   As per HTTP specification, the GET and HEAD methods should be used only for retrieval of resource representations – and they do not update/delete the resource on the server. Both methods are said to be considered “safe“.

   This allows user agents to represent other methods, such as POST, PUT and DELETE, in a special way, so that the user is made aware of the fact that a possibly unsafe action is being requested – and they can update/delete the resource on server and so should be used carefully.

b) Idempotent Methods
   The term idempotent is used more comprehensively to describe an operation that will produce the same results if executed once or multiple times. This is a very useful property in many situations, as it means that an operation can be repeated or retried as often as necessary without causing unintended effects. With non-idempotent operations, the algorithm may have to keep track of whether the operation was already performed or not.

In HTTP specification, The methods GET, HEAD, PUT and DELETE are declared idempotent methods. Other methods OPTIONS and TRACE SHOULD NOT have side effects so both are also inherently idempotent.

HTTP Status Codes

Every HTTP Response message has a parameter called Status Line which is used by REST Api to inform clients of their request's overarching result.

Status-Line syntax:

```
Status-Line = HTTP-Version SP Status-Code SP Reason-Phrase CRLF
```

HTTP defines forty standard status codes that can be used to convey the results of a client's request. The status codes are divided into the five categories presented 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.

Following are the subset of these status codes

1. 200 (OK)

It indicates that the REST API successfully carried out whatever action the client requested, and that no more specific code in the 2xx series is appropriate.

Unlike the 204 status code, a 200 response should include a response body.The information returned with the response is dependent on the method used in the request, for example:

GET an entity corresponding to the requested resource is sent in the response;
HEAD the entity-header fields corresponding to the requested resource are sent in the response without any message-body;
POST an entity describing or containing the result of the action;
TRACE an entity containing the request message as received by the end server.

2. 201 (Created)

   A REST API responds with the 201 status code whenever a resource is created inside a collection. There may also be times when a new resource is created as a result of some controller action, in which case 201 would also be an appropriate response.

   The newly created resource can be referenced by the URI(s) returned in the entity of the response, with the most specific URI for the resource given by a Location header field.

   The origin server MUST create the resource before returning the 201 status code. If the action cannot be carried out immediately, the server SHOULD respond with 202 (Accepted) response instead.

3. 202 (Accepted)

   A 202 response is typically used for actions that take a long while to process. It indicates that the 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 maybe disallowed when processing occurs.

Its purpose is to allow a server to accept a request for some other process (perhaps a batch-oriented process that is only run once per day) without requiring that the user agent's connection to the server persist until the process is completed.

The entity returned with this response SHOULD include an indication of the request's current status and either a pointer to a status monitor (job queue location) or some estimate of when the user can expect the request to be fulfilled.

4. 204 (No Content)

   The 204 status code is usually sent out in response to a PUT, POST, or DELETE request when the REST API declines to send back any status message or representation in the response message's body.

   An API may also send 204 in conjunction with a GET request to indicate that the requested resource exists, but has no state representation to include in the body.

   If the client is a user agent, it SHOULD NOT change its document view from that which caused the request to be sent. This response is primarily intended to allow input for actions to take place without causing a change to the user agent's active document view, although any new or updated metainformation SHOULD be applied to the document currently in the user agent's active view.

   The 204 response MUST NOT include a message-body and thus is always terminated by the first empty line after the header fields.

5. 301 (Moved Permanently)
   The 301 status code indicates that the REST API's resource model has been significantly redesigned and a new permanent URI has been assigned to the client's requested resource. The REST API should specify the new URI in the response's Location header and all future requests should be directed to the given URI.

   You will hardly use this response code in your API as you can always use the API versioning for new API while retaining the old one.

6. 302 (Found)
   The HTTP response status code 302 Found is a common way of performing URL redirection. An HTTP response with this status code will additionally provide a URL in the location header field. The user agent (e.g. a web browser) is invited by a response with this code to make a second, otherwise identical, request to the new URL specified in the location field.

   Many web browsers implemented this code in a manner that violated this standard, changing the request type of the new request to GET, regardless of the type employed in the original request (e.g. POST). RFC 1945 and RFC 2068 specify that the client is not allowed to change the method on the redirected request. The status codes 303 and 307 have been added for servers that wish to make unambiguously clear which kind of reaction is expected of the client.

7. 303 (See Other)

A 303 response indicates that a controller resource has finished its work, but instead of sending a potentially unwanted response body, it sends the client the URI of a response resource. This can be the URI of a temporary status message, or the URI to some already existing, more permanent, resource.

Generally speaking, the 303 status code allows a REST API to send a reference to a resource without forcing the client to download its state. Instead, the client may send a GET request to the value of the Location header.

The 303 response MUST NOT be cached, but the response to the second (redirected) request might be cacheable.

8. 304 (Not Modified)

   This status code is similar to 204 (“No Content”) in that the response body must be empty. The key distinction is that 204 is used when there is nothing to send in the body, whereas 304 is used when the resource has not been modified since the version specified by the request headers If-Modified-Since or If-None-Match.

   In such case, there is no need to retransmit the resource since the client still has a previously-downloaded copy.

   Using this saves bandwidth and reprocessing on both the server and client, as only the header data must be sent and received in comparison to the entirety of the page being re-processed by the server, then sent again using more bandwidth of the server and client.

9. 307 (Temporary Redirect)

   A 307 response indicates that the REST API is not going to process the client’s request. Instead, the client should resubmit the request to the URI specified by the response message’s Location header. However, future requests should still use the original URI.

   A REST API can use this status code to assign a temporary URI to the client’s requested resource. For example, a 307 response can be used to shift a client request over to another host.

   The temporary URI SHOULD be given by the Location field in the response. Unless the request method was HEAD, the entity of the response SHOULD contain a short hypertext note with a hyperlink to the new URI(s). If the 307 status code is received in response to a request other than GET or HEAD, the user agent MUST NOT automatically redirect the request unless it can be confirmed by the user, since this might change the conditions under which the request was issued.

10. 400 (Bad Request)

    400 is the generic client-side error status, used when no other 4xx error code is appropriate. Errors can be like malformed request syntax, invalid request message parameters, or deceptive request routing etc.

    The client SHOULD NOT repeat the request without modifications.

11. 401 (Unauthorized)

    A 401 error response indicates that the client tried to operate on a protected resource without providing the proper authorization. It may have provided the wrong credentials or none at all. The response must include a WWW-Authenticate header field containing a challenge applicable to the requested resource.

    The client MAY repeat the request with a suitable Authorization header field. If the request already included Authorization credentials, then the 401 response indicates that authorization has been refused for those credentials. If the 401 response contains the same challenge as the prior response, and the user agent has already attempted authentication at least once, then the user SHOULD be presented the entity that was given in the response, since that entity might

12. 403 (Forbidden)

    A 403 error response indicates that the client's request is formed correctly, but the REST API refuses to honor it i.e. the user does not have the necessary permissions for the resource. A 403 response is not a case of insufficient client credentials; that would be 401 ("Unauthorized").

    Authentication will not help and the request SHOULD NOT be repeated. Unlike a 401 Unauthorized response, authenticating will make no difference.

13. 404 (Not Found)

    The 404 error status code indicates that the REST API can't map the client's URI to a resource but may be available in the future. Subsequent requests by the client are permissible.

    No indication is given of whether the condition is temporary or permanent. The 410 (Gone) status code SHOULD be used if the server knows, through some internally configurable mechanism, that an old resource is permanently unavailable and has no forwarding address. This status code is commonly used when the server does not wish to reveal exactly why the request has been refused, or when no other response is applicable.

14. 405 (Method Not Allowed)

    The API responds with a 405 error to indicate that the client tried to use an HTTP method that the resource does not allow. For instance, a read-only resource could support only GET and HEAD, while a controller resource might allow GET and POST, but not PUT or DELETE.

    A 405 response must include the Allow header, which lists the HTTP methods that the resource supports. For example:

    Allow: GET, POST

15. 406 (Not Acceptable)

    The 406 error response indicates that the API is not able to generate any of the client's preferred media types, as indicated by the Accept request header. For example, a client request for data formatted as application/xml will receive a 406 response if the API is only willing to format data as application/json.

    If the response could be unacceptable, a user agent SHOULD temporarily stop receipt of more data and query the user for a decision on further actions.

16. 412 (Precondition Failed)

    The 412 error response indicates that the client specified one or more preconditions in its request headers, effectively telling the REST API to carry out its request only if certain conditions were met. A 412 response indicates that those conditions were not met, so instead of carrying out the request, the API sends this status code.

17. 415 (Unsupported Media Type)

    The 415 error response indicates that the API is not able to process the client's supplied media type, as indicated by the Content-Type request header. For example, a client request including data formatted as application/xml will receive a 415 response if the API is only willing to process data formatted as application/json.

    For example, the client uploads an image as image/svg+xml, but the server requires that images use a different format.

18. 500 (Internal Server Error)

    500 is the generic REST API error response. Most web frameworks automatically respond with this response status code whenever they execute some request handler code that raises an exception.

    A 500 error is never the client's fault and therefore it is reasonable for the client to retry the exact same request that triggered this response, and hope to get a different response.

    API response is the generic error message, given when an unexpected condition was encountered and no more specific message is suitable.

19. 501 (Not Implemented)

    The server either does not recognize the request method, or it lacks the ability to fulfill the request. Usually, this implies future availability (e.g., a new feature of a web-service API).

REST Web Services & REST working

REST is used to build Web services that are lightweight, maintainable, and scalable in nature. A service which is built on the REST architecture is called a REST service. The underlying protocol for REST is HTTP, which is the basic web protocol. REST stands for REpresentational State Transfer.

The key elements of a REST implementation are as follows:

Resources – The first key element is the resource itself. Let assume that a web application on a server has records of several employees. Let's assume the URL of the web application is http://demo.guru99.com. Now in order to access an employee record resource via REST, one can issue the command http://demo.guru99.com/employee/1 -

This command tells the web server to please provide the details of the employee whose employee number is 1.

Request Verbs - These describe what you want to do with the resource. A browser issues a GET verb to instruct the endpoint it wants to get data. However, there are many other verbs available including things like POST, PUT, and DELETE. So in the case of the example http://demo.guru99.com/employee/1 , the web browser is actually issuing a GET Verb because it wants to get the details of the employee record.

Request Headers – These are additional instructions sent with the request. These might define the type of response required or the authorization details.

Request Body - Data is sent with the request. Data is normally sent in the request when a POST request is made to the REST web service. In a POST call, the client actually tells the web service that it wants to add a resource to the server. Hence, the request body would have the details of the resource which is required to be added to the server.

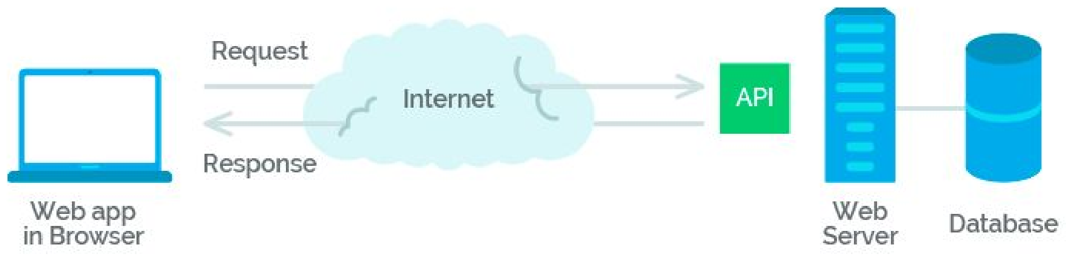
Response Body – This is the main body of the response. So in our example, if we were to query the web server via the request http://demo.guru99.com/employee/1 , the web server might return an XML document with all the details of the employee in the Response Body.

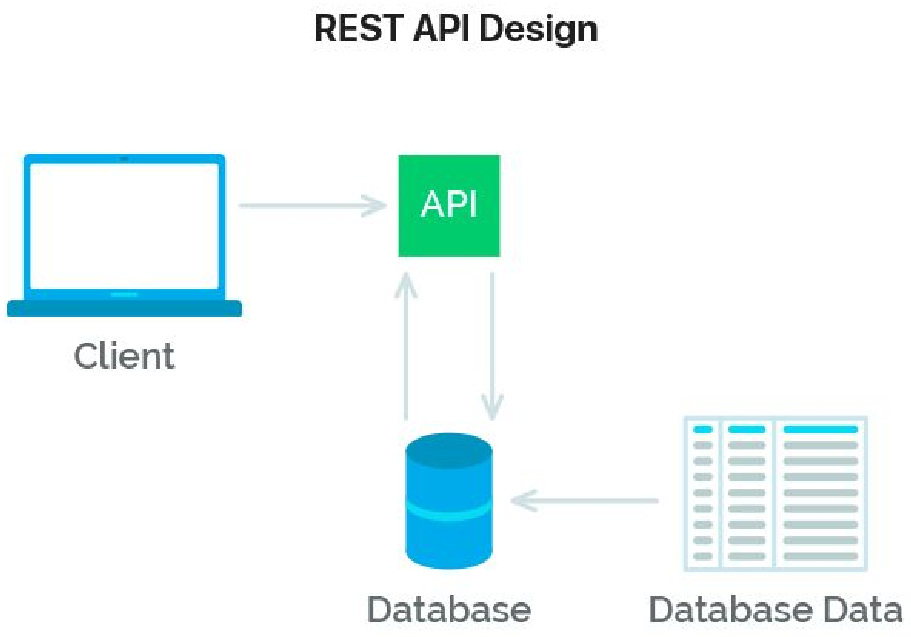
Response Status codes – These codes are the general codes which are returned along with the response from the web server. An example is the code 200 which is normally returned if there is no error when returning a response to the client.

<u>How Does REST Work?</u>

So before jumping into REST Architecture we need to know how an API works.

API (application programming interface) is a set of rules and mechanisms by which one application or component interacts with the others. It seems that the name speaks for itself, but let's get deeper into details. API can return data that you need for your application in a convenient format (e.g. JSON or XML).





A simple definition of RESTful API can easily explain the notion. REST is an architectural style, and RESTful is the interpretation of it. That is, if your back-end server has REST API and you make client-side requests (from a website/application) to this API, then your client is RESTful.

How it works?

RESTful API best practices come down to four essential operations:

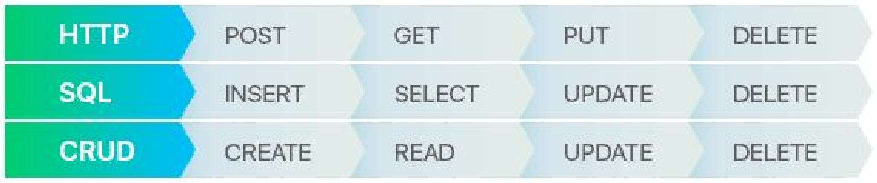
receiving data in a convenient format;
creating new data;

updating data;
deleting data.
REST relies heavily on HTTP. We will not explain the features of this protocol, but it is worth mentioning its great advantage in this situation.

Each operation uses its own HTTP method:

GET - getting;
POST - creation;
PUT - update (modification);
DELETE - removal.
All these methods (operations) are generally called CRUD. They manage data or as Wikipedia says, "create, read, update and delete" it.

The fact that REST contains a single common interface for requests and databases is its great advantage. This can be viewed in the table below.



HTTP	POST	GET	PUT	DELETE
SQL	INSERT	SELECT	UPDATE	DELETE
CRUD	CREATE	READ	UPDATE	DELETE

All requests you make have their HTTP status codes. There are a lot of them and they are divided into 5 classes. The first number indicates which of them a code belongs to:

1xx - informational;
2xx - success;
3xx - redirection;
4xx - client error;
5xx - server error.

You should always provide versioning for your REST API. For example, if the API is at the URL http://example.com/api, it is necessary to make changes to it at http://example.com/api/v1.

RESTful architecture design

All resources in REST are entities. They can be independent like:

GET /users - get all users;
GET /users/123 - get a particular user with id = 123;
GET /posts - get all posts.
There are also dependent entities, that rely on their parent models:

GET /users/123/projects - get all the projects that a user with id = 123 has.
The above examples show that GET implies getting the entity you request. It is idempotent which means receiving identical data while performing the same requests. A successful request returns an entity representation combined with status code 200 (OK). If there is an error you will get back code 404 (Not Found), 400 (Bad Request) or 5xx (Server Error).

Let’s move to the POST method (the creation of a new entity):

POST /users.
When creating a new entity you set parameters in the request body. For example:

```
{
  “first_name”: “Vasyl”,
  “last_name”: “Redka”
}
```

The POST request is not idempotent, which means that if you send the same data in the repeat request, it will create an entity duplicate but with a different identifier.

After that you will get the result which may be status code 200 (OK), for example. Then the response will contain the data of a saved entity. For example:

```
{
  “id”: “1”,
  “first_name”: “Vasyl”,
  “last_name”: “Redka”
}
```

It can also return status 201 (Created), resulting in the creation of a new entity. The server can specify its address in the response body. It is recommended to indicate it in the header “Location”.

The next request is PUT. It is used to update entities. When you send it the request body should contain updated entity data it refers to.

PUT /users/123 - upgrade a user entity with id = 123.
The changes need to be indicated in the parameters. If updated successfully the request returns code 200 (OK) and the representation of the updated entity.

The last request is DELETE. It is simple to understand and is used to remove a specific entity according to an identifier.

DELETE /users/123 - delete a user with id = 123.
If the removal is successful it returns 200 (OK) together with the response body that contains information about the status of the entity. For example, when you do not remove the entity from the database but just mark it as deleted, repeated requests will always return 200 (OK) and the response body with a status. DELETE can be considered as an idempotent request. It can also return code 204 (No Content) without the response body.

If you delete the entity from the database completely, the second request should return 404 (Not Found) because that resource is already removed and no longer accessible.