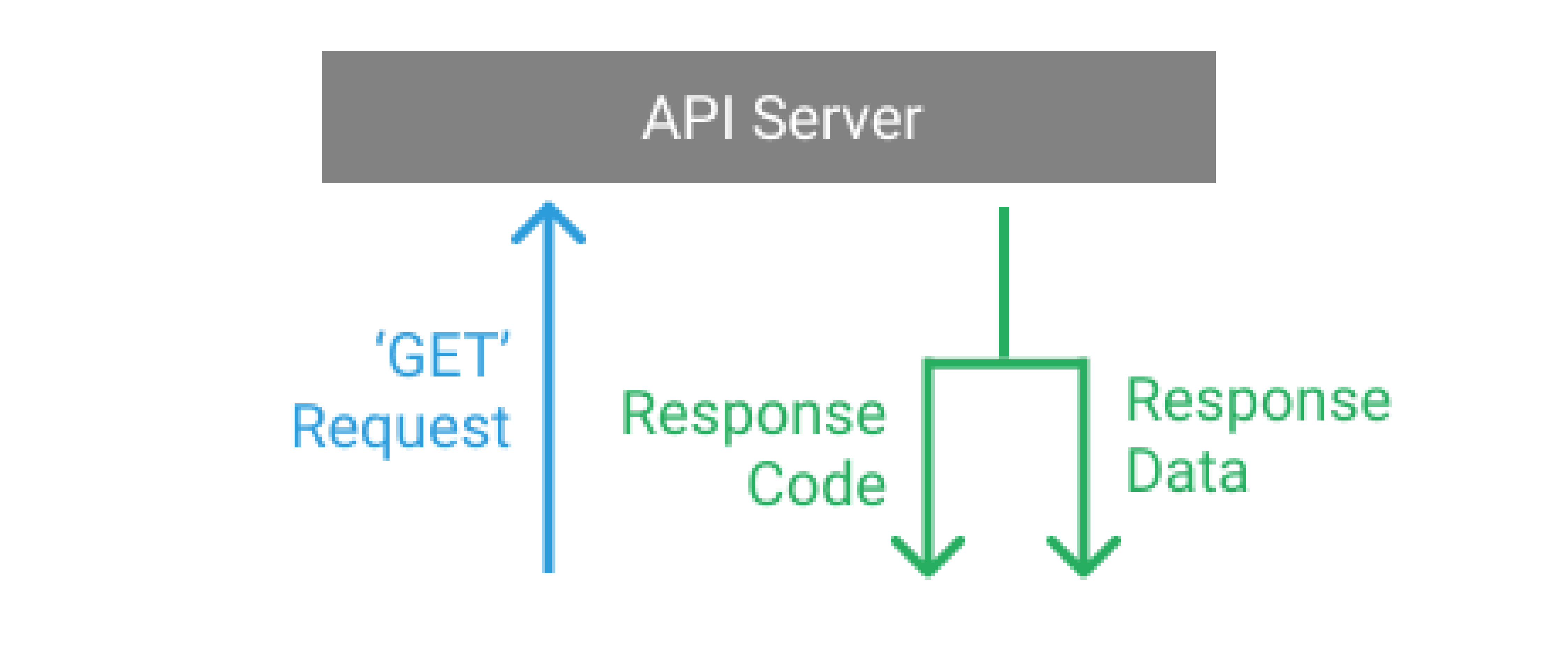
API Requests



API

An **application programming interface (API)** is a way for two or more computer programs to communicate with each other. It is a type of software interface, offering a service to other pieces of software.

A document or standard that describes how to build or use such a connection or interface is called an **API specification**.

A computer system that meets this standard is said to implement or expose an API. The term API may refer either to the specification or to the implementation.

Whereas a system's user interface dictates how its end-users interact with the system in question, its API dictates how to write code that takes advantage of that system's capabilities.

Source: Wikipedia. Source for illustration on the first slide.

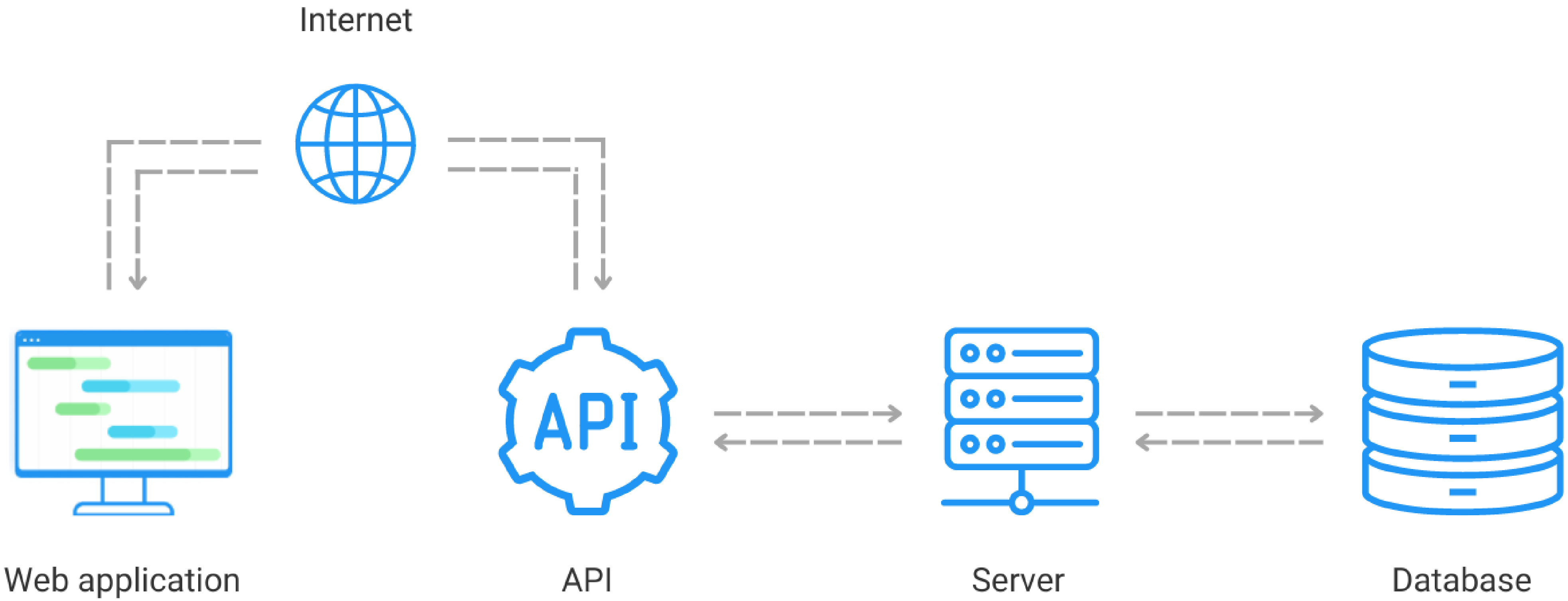


Image source

What is an API endpoint?

An **API endpoint** is a *public URL* exposed by a server that acts as the point of contact between an API client and the API server.

API clients send requests to API endpoints in order to access the API's functionality and data.

How do API endpoints work?

API endpoints work by connecting API clients and servers—and handling the transfer of data between them. A well-designed API should have clear and intuitive endpoints that provide a predictable way for clients to interact with the server's resources. For example, a REST API that powers a simple blogging application might have the following endpoints, which can be accessed with the indicated HTTP methods:

- `/authors` – to retrieve a list of users (`GET`) or create a new user (`POST`)
- `/authors/:id` – to retrieve a specific user (`GET`), update an existing user (`PUT` or `PATCH`), or delete a specific user (`DELETE`)
- `/articles` – to retrieve a list of articles (`GET`) or create a new article (`POST`)
- `/articles/:id` – to retrieve a specific article (`GET`), update an existing article (`PUT` or `PATCH`), or delete a specific article (`DELETE`)

In this example, we can see that the API exposes two sets of endpoints:

- one for the **Author** resource, and
- one for the **Article** resource.

Each resource can be accessed through two different endpoints, depending on the type of operation the client would like to perform.

For example, if the client is interested in seeing all of the authors in the database, it would send a `GET` request to the `/authors` endpoint.

In contrast, the `/authors/:id` endpoint enables the client to view, update, or delete a specific author's data by including the author's id as a request parameter.

The API client is responsible for assembling and sending the request to the API server.

In addition to the endpoint and method, which are required, the request may also include parameters, HTTP headers, and a request body:

- **Parameters** are variables that are passed to an API endpoint, and they provide specific instructions for the API to process. For example, the `/articles` endpoint a blogging application might accept a `category` parameter, which it would use to retrieve articles of the specified category.
- **Request headers** are key-value pairs that provide additional information about the request. For instance, the *Accept header* specifies the media types that the client can accept, while the *Authorization header* is used to send tokens and API keys to authenticate the client.
- **A request body** includes the actual data that is required to *create*, *update*, or *delete* a resource. For instance, if an author wants to create a new article in an example blogging application, they would send a `POST` request to the `/articles` endpoint with the content of the article in the request's body.

Once the client sends the request to the appropriate endpoint, the API server authenticates it, validates the input, retrieves or manipulates the relevant data, and returns the response to the client.

The response typically includes a **status code**, which indicates the result of the request, as well as a **body**, which contains the actual data that the client requested (if the request was successfully executed).

API documentation is a set of human-readable instructions for using and integrating with an API.

API documentation includes detailed information about an API's available endpoints, methods, resources, authentication protocols, parameters, and headers, as well as examples of common requests and responses.

Read the documentation

Most APIs do work in similar ways, there's even a specification of sorts, but they can still have wildly different approaches and functionality, so **Read The Documentation**!

Also, many of the APIs, although free, still require keys to use. Logging in to the service will give you access to automatically created keys.

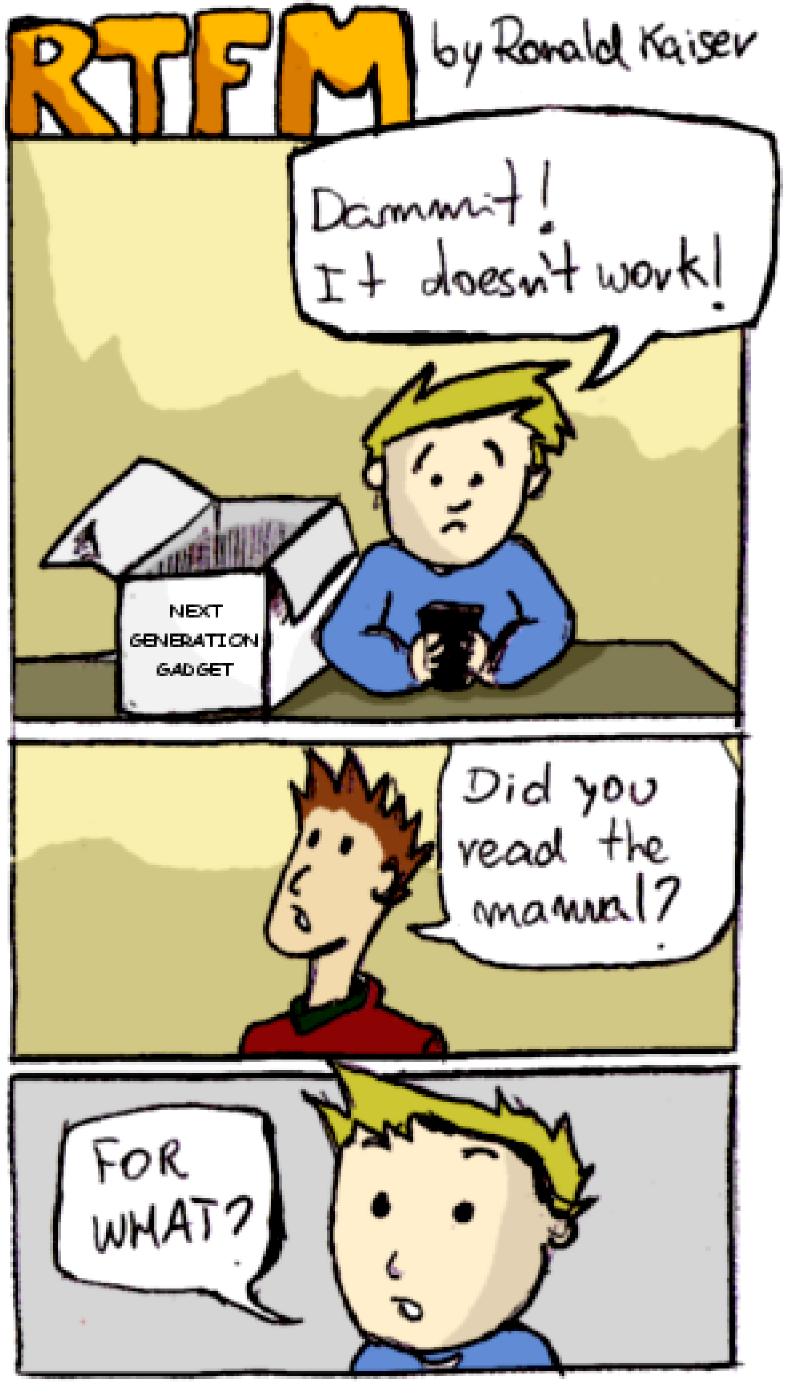


Image source

What is a REST API?

Representational state transfer (**REST**) is a de-facto standard for a software architecture for interactive applications that typically use multiple Web services.

A Web service is said to be **RESTful** when it:

- provides an application access to its Web resources
    - in a textual representation
    - support reading and modification of them with
        - a stateless protocol
        - a predefined *set of operations*.

Requests made to a resource's *URI* will generate a response with a payload formatted in HTML, XML, *JSON*, or some other format.

Representational state transfer

`fetch()`

The global `fetch()` method starts the process of fetching a resource from the network, returning a promise that is fulfilled once the response is available.

The promise resolves to the `Response` object representing the response to your request.

A `fetch()` promise only rejects when a network error is encountered (which is usually when there's a permissions issue or similar). A `fetch()` promise does not reject on HTTP errors (404, etc.). Instead, a then() handler must check the Response.ok and/or Response.status properties.

A basic `fetch()` request looks like this, using `async` / `await`

```
async function getData() {
  const api = "https://jsonplaceholder.typicode.com/todos/1";
  const response = await fetch(api);
  const obj = await response.json();
  console.log(obj);
}
getData();
```

...or using a `then()` chain:

```
const api = "https://jsonplaceholder.typicode.com/todos/2";
fetch(api)
    .then (response => response.json())
    .then (obj => console.log(obj))
```

Note that both these are without any error handling, so let's revisit the first one:

Here we check the response, and add a `try/catch`:

```
async function getData() {
  try {
    const api = "https://jsonplaceholder.typicode.com/todos/1";
    const response = await fetch(api);
    if (!response.ok) throw new Error(`HTTP error! ${response.status}`);
    const obj = await response.json();
    console.log(obj);
  } catch (error) {
    console.error(error.message);
  }
}

getData();
```

Codepen

In the `then()` we need to check the response, too, then add a `catch()` to the chain.

```
const api = "https://jsonplaceholder.typicode.com/todos/2";
fetch(api)
    .then ((response) => {
        if (!response.ok) throw new Error(`HTTP error! ${response.status}`);
        return response.json();
    })
    .then (obj => console.log(obj))
    .catch (error => console.error(error.message))
```

Codepen

HTTP Request Methods

HTTP Request Methods

HTTP defines a set of request methods to indicate the desired action to be performed for a given resource.

Although they can also be nouns, these request methods are sometimes referred to as HTTP verbs.

Each of them implements a different semantic, but some common features are shared by a group of them: e.g. a request method can be safe, idempotent, or cacheable.

MDN Web Docs: HTTP request methods

The word "idempotent" means that any number of repeated, identical requests will leave the resource in the same state.

HTTP `GET` Request Method

The HTTP `GET` method requests a representation of the specified resource.

Requests using `GET` should only be used to request data (they shouldn't include data).

MDN Web Docs: GET

Example using `fetch()`, using the `options` parameter to specify method (`GET` is default):

```
const url = "https://example.com?foo=1&bar=2'";
fetch(url, {
    method: 'GET'
}).then (/* */)
```

> **Note**: Sending body/payload in a GET request may cause some existing implementations to reject the request — while not prohibited by the specification, the semantics are undefined. It is better to just avoid sending payloads in GET requests.

HTTP `HEAD` Request Method

The HTTP `HEAD` method requests the headers that would be returned if the `HEAD` request's URL was instead requested with the HTTP `GET` method.

For example, if a URL might produce a large download, a `HEAD` request could read its `Content-Length` header to check the filesize without actually downloading the file.

MDN Web Docs: HEAD

> **Warning**: A response to a HEAD method should not have a body. If it has one anyway, that body must be ignored: any representation headers that might describe the erroneous body are instead assumed to describe the response which a similar GET request would have received.

HTTP `POST` Request Method

The HTTP `POST` method sends data to the server. The type of the body of the request is indicated by the `Content-Type` header.

The difference between `PUT` and `POST` is that `PUT` is idempotent: calling it once or several times successively has the same effect (that is no side effect), where successive identical `POST` may have additional effects, like passing an order several times.

MDN Web Docs: POST

A `POST` request is typically sent via an HTML form and results in a change on the server.

In this case, the content type is selected by putting the adequate string in the `enctype` attribute of the `<form>` element or the `formenctype` attribute of the `<input>` or `<button>` elements:

- `application/x-www-form-urlencoded`: the keys and values are encoded in key-value tuples separated by `'&'`, with a `'='` between the key and the value. Non-alphanumeric characters in both keys and values are percent encoded: this is the reason why this type is not suitable to use with binary data (use `multipart/form-data` instead)
- `multipart/form-data`: each value is sent as a block of data ("body part"), with a user agent-defined delimiter ("boundary") separating each part. The keys are given in the `Content-Disposition` header of each part.
- `text/plain`

When the `POST` request is sent via a method other than an HTML form — like via an `XMLHttpRequest` — the body can take any type.

As described in the HTTP 1.1 specification, `POST` is designed to allow a uniform method to cover the following functions:

- Annotation of existing resources
- Posting a message to a bulletin board, newsgroup, mailing list, or similar group of articles;
- Adding a new user through a signup modal;
- Providing a block of data, such as the result of submitting a form, to a data-handling process;
- Extending a database through an append operation.

```
// Example POST method implementation:
async function postData(url = '', data = {}) {
  // Default options are marked with *
  const response = await fetch(url, {
    // see fetch() for the full options available
    method: 'POST',
    mode: 'cors',
    cache: 'no-cache',
    credentials: 'same-origin',
    headers: {
      'Content-Type': 'application/json'
    },
    redirect: 'follow',
    referrerPolicy: 'no-referrer',
    body: JSON.stringify(data) // body data type must match "Content-Type" header
  });
  return response.json(); // parses JSON response into native JavaScript objects
}

postData('https://jsonplaceholder.typicode.com/posts', { answer: 42 })
  .then((data) => {
    console.log(data); // { "answer": 42, "id": 101 }
  })
  .catch((error) => console.error(error.message))
;
```

Codepen

HTTP `PUT` Request Method

The HTTP `PUT` request method creates a new resource or replaces a representation of the target resource with the request payload.

The difference between `PUT` and `POST` is that `PUT` is idempotent: calling it once or several times successively has the same effect (that is no *side* effect), whereas successive identical `POST` requests may have additional effects, akin to placing an order several times. (R)

MDN Web Docs: PUT

HTTP `TRACE` Request Method

The HTTP `TRACE` method performs a message loop-back test along the path to the target resource, providing a useful debugging mechanism.

The final recipient of the request should reflect the message received, excluding some fields described below, back to the client as the message body of a `200` (OK) response with a `Content-Type` of `message/http`. The final recipient is either the origin server or the first server to receive a `Max-Forwards` value of 0 in the request.

MDN Web Docs: TRACE

HTTP `PATCH` Request Method

The HTTP `PATCH` request method applies partial modifications to a resource.

`PATCH` is somewhat analogous to the "update" concept found in CRUD
(in general, HTTP is different than CRUD, and the two should not be confused).

A `PATCH` request is considered a set of instructions on how to modify a resource. Contrast this with `PUT`; which is a complete representation of a resource.

MDN Web Docs: PATCH

A `PATCH` is not necessarily idempotent, although it can be. Contrast this with `PUT`; which is always idempotent. The word "idempotent" means that any number of repeated, identical requests will leave the resource in the same state.

For example if an auto-incrementing counter field is an integral part of the resource, then a `PUT` will naturally overwrite it (since it overwrites everything), but not necessarily so for `PATCH`.

`PATCH` (like `POST`) may have side-effects on other resources.

To find out whether a server supports `PATCH`, a server can advertise its support by adding it to the list in the `Allow` or `Access-Control-Allow-Methods` (for CORS) response headers.

Another (implicit) indication that `PATCH` is allowed, is the presence of the `Accept-Patch` header, which specifies the patch document formats accepted by the server.

HTTP CONNECT Request Method

The HTTP `CONNECT` method starts two-way communications with the requested resource. It can be used to open a tunnel.

For example, the `CONNECT` method can be used to access websites that use SSL (HTTPS). The client asks an HTTP Proxy server to tunnel the TCP connection to the desired destination. The server then proceeds to make the connection on behalf of the client. Once the connection has been established by the server, the Proxy server continues to proxy the TCP stream to and from the client.

`CONNECT` is a hop-by-hop method.

MDN Web Docs: CONNECT

HTTP OPTIONS Request Method

The HTTP `OPTIONS` method requests permitted communication options for a given URL or server. A client can specify a URL with this method, or an asterisk (`*`) to refer to the entire server.

MDN Web Docs: OPTIONS

HTTP DELETE Request Method

The HTTP `DELETE` request method deletes the specified resource.

MDN Web Docs: DELETE

HTTP Response Codes

HTTP Response Status Codes

HTTP response status codes indicate whether a specific HTTP request has been successfully completed. Responses are grouped in five classes:

1. Informational responses (`100` – `199`)
2. Successful responses (`200` – `299`)
3. Redirection messages (`300` – `399`)
4. Client error responses (`400` – `499`)
5. Server error responses (`500` – `599`)

The status codes are defined by RFC 9110.

404 Not Found

The server can not find the requested resource.

In the browser, this means the URL is not recognized.

In an API, this can also mean that the endpoint is valid but the resource itself does not exist.

Servers may also send this response instead of 403 Forbidden to hide the existence of a resource from an unauthorized client.

This response code is probably the most well known due to its frequent occurrence on the web.

200 OK

The request succeeded. The result meaning of "success" depends on the HTTP method:

- `GET`: The resource has been fetched and transmitted in the message body.
- `HEAD`: The representation headers are included in the response without any message body.
- `PUT` or `POST`: The resource describing the result of the action is transmitted in the message body.
- `TRACE`: The message body contains the request message as received by the server.

301 Moved Permanently

The URL of the requested resource has been changed permanently.

The new URL is given in the response.

401 Unauthorized

Although the HTTP standard specifies "unauthorized", semantically this response means "unauthenticated".

That is, the client must authenticate itself to get the requested response.

403 Forbidden

The client does not have access rights to the content; that is, it is unauthorized, so the server is refusing to give the requested resource.

Unlike 401 Unauthorized, the client's identity is known to the server.

500 Internal Server Error

The server has encountered a situation it does not know how to handle.

503 Service Unavailable

The server is not ready to handle the request.

Common causes are a server that is down for maintenance or that is overloaded.

Note that together with this response, a user-friendly page explaining the problem should be sent. This response should be used for temporary conditions and the Retry-After HTTP header should, if possible, contain the estimated time before the recovery of the service. The webmaster must also take care about the caching-related headers that are sent along with this response, as these temporary condition responses should usually not be cached.

HTTP headers

HTTP headers let the client and the server pass additional information with an *HTTP request or response*.

An HTTP header consists of its case-insensitive name followed by a colon (:), then by its value. Whitespace before the value is ignored.

Custom proprietary headers have historically been used with an X- prefix, but this convention was deprecated in June 2012 because of the inconveniences it caused when nonstandard fields became standard.

Source

Request headers

Contain more information about the resource to be fetched, or about the client requesting the resource.

For example, the `Accept-*` headers indicate the allowed and preferred formats of the response. Other headers can be used to supply authentication credentials (e.g. `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 header is referred to as a representation header.

In addition, CORS defines a subset of request headers as simple headers, request headers that are always considered authorized and are not explicitly listed in responses to preflight requests.

Setting Custom Headers

Using **options**, an object containing any custom settings you want to apply to the request.

Option objects can contain several information, eg. method, headers and body. Here we set method (explicitly), and a custom header:

```
fetch('https://jsonplaceholder.typicode.com/todos/1', {
  method: 'GET',
  headers: {
    'X-Custom-Header': 'CustomValue'
  }
})
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(error => console.error('Error:', error));
```

You can also pass the options object in as an argument, by making it a variable:

```
const options = {
  method: 'GET',
  headers: {
    'X-Custom-Header': 'CustomValue'
  }
};

fetch('https://jsonplaceholder.typicode.com/todos/1', options)
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(error => console.error('Error:', error));
```

A post example:

```
// Example POST method implementation:
async function postData(url = "", data = {}) {
  // Default options are marked with *
  const response = await fetch(url, {
    method: "POST",
    headers: {
      "Content-Type": "application/json",
      //"Content-Type": "text/html",
    },
    body: JSON.stringify(data), // body data type must match "Content-Type" header
  });
  return response.json();
}

postData("https://jsonplaceholder.typicode.com/todos", { answer: 42 })
  .then((data) => { console.log(data); });
// Expected response: Object { answer: 42, id: 201 }
// Expected response (for "Content-Type": "text/html"): Object { id: 201 }
```

Codepen

Response headers

Hold additional information about the response, like its location or about the server providing it.

Not all headers appearing in a response are categorized as response headers by the specification.

For example, the `Content-Type` header is a representation header indicating the original type of data in the body of the response message (prior to the encoding in the `Content-Encoding` representation header being applied).

However, "conversationally" all headers are usually referred to as response headers in a response message.

Accessing Response Headers

```
console.clear();
fetch('https://jsonplaceholder.typicode.com/todos/1')
          .then(response => {
              console.log('All headers:', response.headers);
              console.log('Content-Type:', response.headers.get('Content-Type'));
              return response.json();
          })
          .then(data => console.log(data))
          .catch(error => console.error('Error:', error));

// All headers:
// Headers(4) {
//   "cache-control" → "max-age=43200",
//   "content-type" → "application/json; charset=utf-8",
//   expires → "-1",
//   pragma → "no-cache"
// }
// Content-Type: application/json; charset=utf-8
// Object { userId: 1, id: 1, title: "delectus aut autem", completed: false }
```

Examples / Demos

- Cat Fact Documentation
- Dog Images Documentation
- Random Joke Documentation

List of free APIs

Free API – Huge List of Public APIs For Testing [No Key]

Free Public APIs for Developers

Big List of Free and Open Public APIs (No Auth Needed)

{JSON} Placeholder - Free fake API for testing and prototyping.

Todos

1. Find an API (from the lists above or elsewhere), fetch the datas (using async/await) and print to a webpage.
2. Sign up for Postman (for tomorrow)

Mollify

Read API

Read Body Encoding in HTML Requests

Read REST API

Read fetch

Read HTTP Request Methods

Read HTTP Response Codes

Read Request and Response Headers