HTTP Methods

HTTP defines a set of methods. We must choose one to use each time we make an HTTP request. The most common ones include:

- `GET`
- `POST`
- `PUT`
- `DELETE`

The GET method is used to "get" a representation of a specified resource. It doesn't take (remove) the data from the server but rather gets a representation, or copy, of the resource in its current state. A GET request is considered a safe method to call multiple times because it shouldn't alter the state of the server.

GET - Making a GET request using the Fetch API

In this course, we have been and will continue to use the Fetch API to make HTTP requests. The `fetch()` method accepts an optional `init` object parameter as its second argument that we can use to define things like:

- `method`: The HTTP method of the request, like GET.
- `headers`: The headers to send.
- `mode`: Used for security, we'll talk about this in future courses.
- `body`: The body of the request. Often encoded as JSON.

Example `GET` request using fetch:

```
await fetch(url, {
  method: 'GET',
  mode: 'cors',
  headers: {
    'sec-ch-ua-platform': 'macOS'
  }
})
```

Why use HTTP methods?

The primary purpose of HTTP methods is to indicate to the server what we want to do with the resource we're trying to interact with. At the end of the day, an HTTP method is just a string, like `GET`, `POST`, `PUT`, or `DELETE`, but by convention, backend developers write their server code so that the methods correspond with different "CRUD" actions.

The "CRUD" actions are:

- Create
- Read
- Update

- Delete

The bulk of the logic in most web applications is "CRUD" logic. Even a complex social media site is mostly just CRUD. Users create, read, update and delete their accounts. They create, read, update, and delete their posts. It's CRUD all the way down!

The 4 most common HTTP methods map nicely to the CRUD actions:

HTTP Method	CRUD Action
GET	Read
POST	Create
PUT	Update
DELETE	Delete

POST Requests

An HTTP POST request sends data to a server, typically to create a new resource.

Adding a body

The `body` of the request is the payload sent to the server. The special `Content-Type` header is used to tell the server the format of the body: `application/json` for JSON data in our case. `POST` requests are generally not safe methods to call multiple times because that would create duplicate records. For example, you wouldn't want to accidentally send a tweet twice.

```
await fetch(url, {
  method: 'POST',
  mode: 'cors',
  headers: {
    'Content-Type': 'application/json'
  },
  body: JSON.stringify(data)
})
```

Status Codes

The `Status Code` of an HTTP response tells the client whether or not the server was able to fulfill the request. Status codes are 3-digit numbers that are grouped into categories:

- `100-199`: Informational responses. These are very rare.
- `200-299`: Successful responses. Hopefully, most responses are 200's!
- `300-399`: Redirection messages. These are typically invisible because the browser or - HTTP client will automatically do the redirect.
- `400-499`: Client errors. You'll see these often, especially when trying to debug a - client application
- `500-599`: Server errors. You'll see these sometimes, usually only if there is a bug on the server.

Here are some of the most common status codes, but there is also a full list here if you're interested.

- `200` - OK. This is by far the most common code, it just means that everything worked - as expected.
- `201` - Created. This means that a resource was created successfully. Typically in - response to a POST request.
- `301` - Moved permanently. This means the resource was moved to a new place, and the - response will include where that new place is. Websites often use 301 redirects when - they change their domain name, for example.
- `400` - Bad request. A general error indicating the client made a mistake in their - request.
- `401` - Unauthorized. This means the client doesn't have the correct permissions. - Maybe they didn't include a required authorization header, for example.
- `404` - Not found. You'll see this on websites quite often. It just means the resource - doesn't exist.
- `500` - Internal server error. This means something went wrong on the server, likely a - bug on their end.

Don't memorize

It's good to know the basics by heart, like **"2XX is good"**, **"4XX is a client error"**, and **"5XX is a server error"**. But don't memorize all the status codes... they're easy to look up.

Status Code Property

The `Response` has a `.status` property that contains the status code of the response.

Example of using the `.status` property

Here's a simple example of how to use the `.status` property to check the status code of a response:

```
fetch(url)
  .then(response => {
    if (response.status === 200) {
    console.log('Request was successful!');
    } else {
      console.log('Request failed with status:', response.status);
    }
  })
  .catch(error => {
    console.error('Error:', error);
  });
```

PUT Requests

The HTTP `PUT` method creates or more commonly, updates a representation of the target resource with the contents of the request's `body`. In short, it updates a resource's properties.

```
await fetch(url, {
   method: 'PUT',
   mode: 'cors',
   headers: {
```

```
    'Content-Type': 'application/json'
    },
    body: JSON.stringify(data)
})
```

POST vs PUT

While `POST` and `PUT` are both used for creating resources, `PUT` is more common for updates and is idempotent, meaning it's safe to make the request multiple times without changing the server state. For example:

`POST /users/bob (create bob user) POST /users/bob (create duplicate bob user) POST /users/bob (create duplicate bob user)`

`PUT /users/bob (create bob user if it doesn't exist) PUT /users/bob (update bob user with the same data) PUT /users/bob (update bob user with the same data)`

PATCH vs PUT

You may encounter the `PATCH` method from time to time. It's frankly not used very often. It's meant to partially modify a resource, whereas `PUT` is meant to completely replace a resource.

`PATCH` is not as popular as `PUT`, and many servers, even if they allow partial updates, just use PUT.

DELETE

The `DELETE` method does exactly what you expect: it deletes a specified resource.

Example

```
// This deletes the location with ID: 52fdfc07-2182-454f-963f-5f0f9a621d72
const url = 'https://api.boot.dev/v1/courses_rest_api/learn-http/locations/52fdfc07-2182-454f-963f-5f0f9a621d72'

await fetch(url, {
  method: 'DELETE',
  mode: 'cors'
})
`js
```