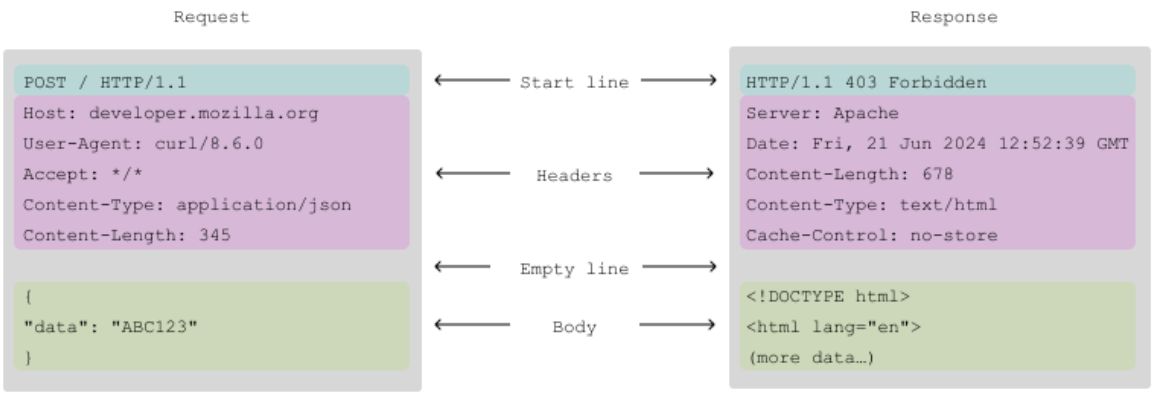
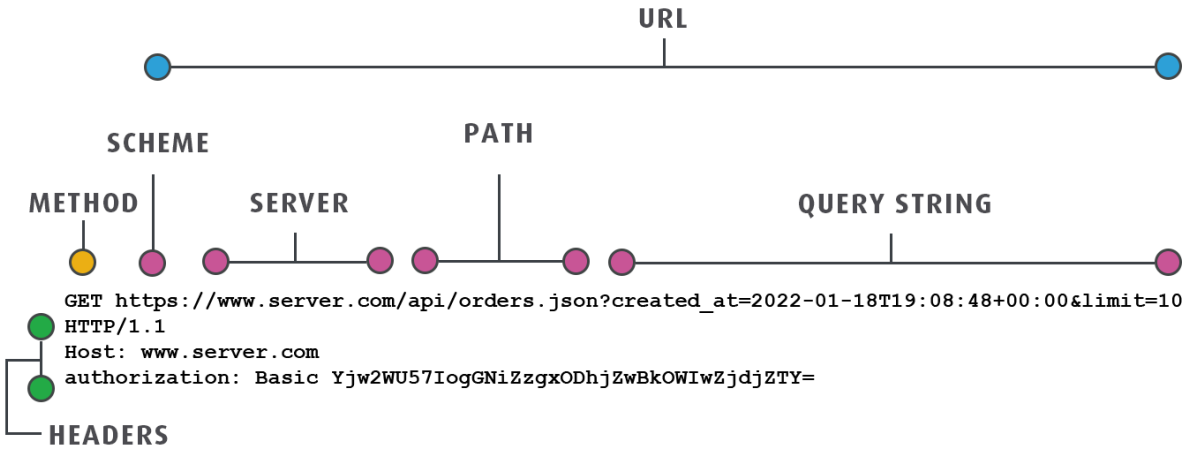
Prerequisite knowledge
A couple terms that are important to know:
Authentication- who are you?
Authorization- what can you do?
Before we can determine what a user can do, we first need to determine who a user is. Unauthenticated users shouldn't be authorized to do anything except login (which should authenticate them if successful) or register.

—

An HTTP request has the following components:



The URL of an HTTP request has the following structure:



The problems we're trying to solve

- When a user logs into our website, and they send requests to our website after logging in, how do we know that those requests are from that specific logged-in user?
- How can we ensure that a user is who they claim to be? How can we ensure that they can't impersonate other users?

One solution is storing JWTs in cookies.

What is JWT? (https://www.jwt.io/introduction#why-use-json-web-tokens)
JWT stands for JSON web token, and is used for secure transfer of information between parties, which allows for it to be used for authentication.

Structure of a JWT
A string of 3 parts concatenated, delimited by ".":

Header.Payload.Signature

- Header- usually consists of two parts, which are then Base64Url encoded:
  - alg: the signing algorithm (ex: "HS256")
  - typ: the type of the token ("JWT")
- Payload- contains data called claims, which are then Base64Url encoded. An example payload could be:
  - "token_type": "access",
  - "exp" (expires at): 1694800500
  - "iat" (issued at): 1694800200,
  - "jti" (JSON token ID): "f43f8f5f2c124e88a693e8df4fbb8070"
  - "user_id": 42
- Signature: used to verify that the Header and payload weren't tampered with. Uses the algorithm specified in the Header. The secret (a private string key) is stored on the server, and should not be shared. If the algorithm was HS256, the signature would be calculated as follows:
  - HMACSHA256(
    base64UrlEncode(Header) + "." + base64UrlEncode(Payload),
    secret
)

How JWTs are used to authenticate
When a user logs in, the server creates two JWTs for the user- one for the access token, and one for the refresh token- and then sends the JWTs to the user. The JWTs' payloads contain data about the user who logged in, when they were created, and when they expire. The JWTs' signatures are created using the server's secret.

For future requests to the server, the user sends the JWTs in the requests. The server can extract the JWTs from the request, and check that hashing/encoding the JWTs' Headers and Payloads using the specified algorithm with the server's secret key matches the signatures sent in the JWTs. If they don't match, the server can reject the request. This could happen, for example, if the user tried changing the payload so that it said a different user besides themselves, but then would be unable to forge the correct signature since they don't have the server's secret key. This makes authentication reliable by making users unable to impersonate other users.

The access token expires quickly, while the refresh token takes longer to expire. When the access token expires, the refresh token is used to create a new valid access token. If the refresh token happens to be expired too, the user will need to ask the server to generate new access and refresh tokens.

When a client receives JWTs from the server, they can choose how to:

1. Store the JWTs- localStorage, cookies
2. Send them with future requests- request Header (not the same as a JWT Header), request Body, or as a query parameter

We'll choose to store JWTs in cookies and send them in request Headers. The next sections explain why.

What is a cookie? (https://developer.mozilla.org/en-US/docs/Web/HTTP/Guides/Cookies)
A cookie is a small piece of data that a server sends to a browser.

The browser may store cookies, create new cookies, modify existing ones, and send them back to the same server with later requests in the request's Header.
On websites that have logins, a cookie can contain data related to your current login session, so you should never share your cookies with someone else- that may allow them to access your account.

Annoyingly, cookies are also used to track you so that websites can collect data about your browsing habits and send you targeted, personalized ads. This is particularly common with third-party cookies, which are cookies set by websites besides the one you are visiting, such as by the server of an ad that is embedded in the website you're viewing.

Why should we store data in cookies instead of localStorage?
Cookies:

- More complicated to use
- Are meant to be read by the server
- Only stores strings
- Max amount of data they can hold is lower (usually 4 KB)
- Data can be encrypted with secure attribute
- Are always sent in the request Header (if the cookie's samesite attribute allows it*)
    - If it's not necessary to send a cookie to a server, this can cause unnecessary overhead as you are sending extra data with each request
- Can be set to automatically be deleted after a specific time

localStorage:

- Simpler to use
- Can only be read by the client
- Can store JSON

- Max amount of data they can hold is higher (usually 5 MB)
- Data can't be encrypted automatically, but you can choose to encrypt the data manually and then store the data
- Can manually retrieve the data from localStorage and send it in request Header (if the API allows for it*), request Body, or as a query parameter
- Data persists until manually cleared by client (with sessionStorage: cleared when session ends)

Overall, cookies can be more complicated and less performant than localStorage, but it's more secure.

How should we send JWTs to the server?
Sending JWTs in request Headers is preferred because:

1. Headers are designed for passing in authentication credentials
2. It works regardless of the type of request- GET HTTP requests don't have a request Body.
3. Full URLs, including query parameters, are often logged by proxy servers, even in HTTPS

How to view cookies
In Firefox (this probably works similarly in other browsers like Google Chrome), you can view cookies in Developer tools > Storage > Cookies.
Then, click on a specific entry to see the data it's storing.

Example of setting a cookie
Here is some server code (Django backend) returning a response (type Response imported from the rest_framework.response module) to a client.

```
    accessToken = "placeholder for random string"

    response.set_cookie(
        "Access_token",
        accessToken,
        httponly=True,
        secure=True,
        samesite="Lax",
        max_age=7*24*60*60
    )

    response.data = {"detail": "Access token refreshed"}

    return response
```

Here's a breakdown of what this code is doing.

- **"Access_token"** is the Name of the cookie.
- **accessToken**, whose value is "placeholder for random string" in this case, is the Value of the cookie- the data that the cookie is actually storing.
- **httponly** makes it so that the cookie can't be accessed via javaScript (for example, by executing document.cookie).
- **secure** makes it so that the cookie can't be sent over http, only https. This is important because proxy servers can read cookie data, so if they read an unencrypted cookie that stores login session data, they could login to the website as you. If the cookie is sent over https, they can only see the encrypted cookie, so they can't steal the raw cookie.
- **samesite** controls what websites the cookie can be sent to when the user makes requests. It can take on 3 possible values:
    - "Strict"- the cookie is only sent in requests to the same site that set the cookie. Good if your frontend and backend are hosted on the same domain.
    - "Lax"- like previous, but has some more complicated rules that I don't care to put here
    - "None"- the cookie is sent in requests to every site. secure should be True if this is selected, otherwise you could be sending sensitive, unencrypted session information to other servers!
- **max_age** tells the browser to automatically delete the cookie after the specified time. In the API that this code snippet uses, it's specified as a number of seconds, in this case 1 week. If it's not specified, by default the cookie becomes a session cookie, meaning it'll be deleted after the user's session ends (i.e when they x-out the page.)

When the client receives the cookie, the browser will save it and know to send it in future requests' Headers.