SECURING API COMMUNICATION

Qinshift Academy

AUTHENTICATION VS AUTHORIZATION?

Authentication and Authorization are key principles in securing any application. They have different meaning and they happen in a different point in the security cycle.

Authentication

Authentication and Authorization are key principles in securing any application. They have different meaning and they happen in a different point in the security cycle.

Authentication - The process where we identify and verify a user or process that is trying to access the application. We always do authentication before authorization.

It answers the question: **Who are you?**

Authorization

Authorization - The process where we give permission to the user or process on what and where they can access in the application. This process is done after authentication because we need to know who the user is before we can grant them access to features of our application.

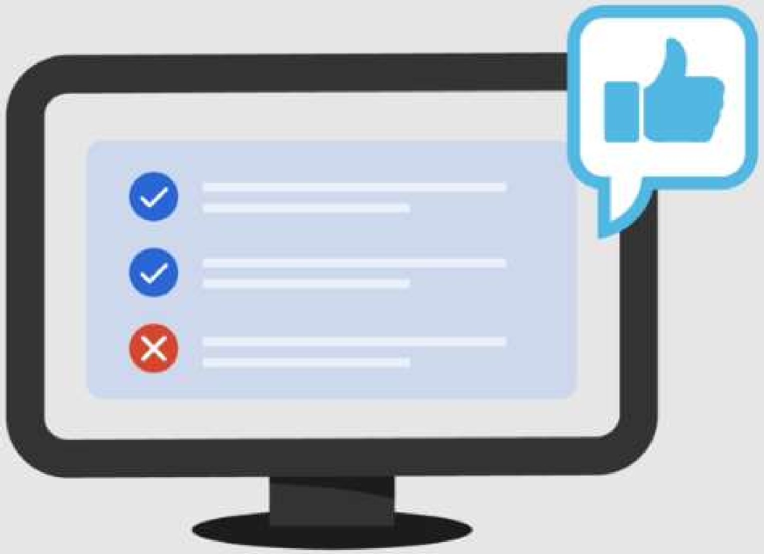
It answers the question: **What are you allowed to do/access?**

Authentication
VERSUS
Authorization

Authentication

Confirms users
are who they say they are.

Authorization



Gives users permission
to access a resource.

Securing API applications

Securing API applications is done through implementing security measures on the application it self as well as on the requests that are used for communication with the client. There should also be focus on the client experience and usability. There are multiple systems for managing security as well as accessibility to the client. Some of the most used ones are:

- Session based
- Token based

Sessions (stateful)

Working with sessions is one way of authenticating a source of requests and keeping that connection secure. This method works by the server creating sessions for every client that requests something from it. These sessions are kept on the server machine. Every time a client requests for something, they are authenticated and then they are given a cookie with a session id which represents their session id on the server along with a signature unique to the server called a secret that makes the cookie authentic and serves as a protection from the client changing the cookie. This method works as follows:

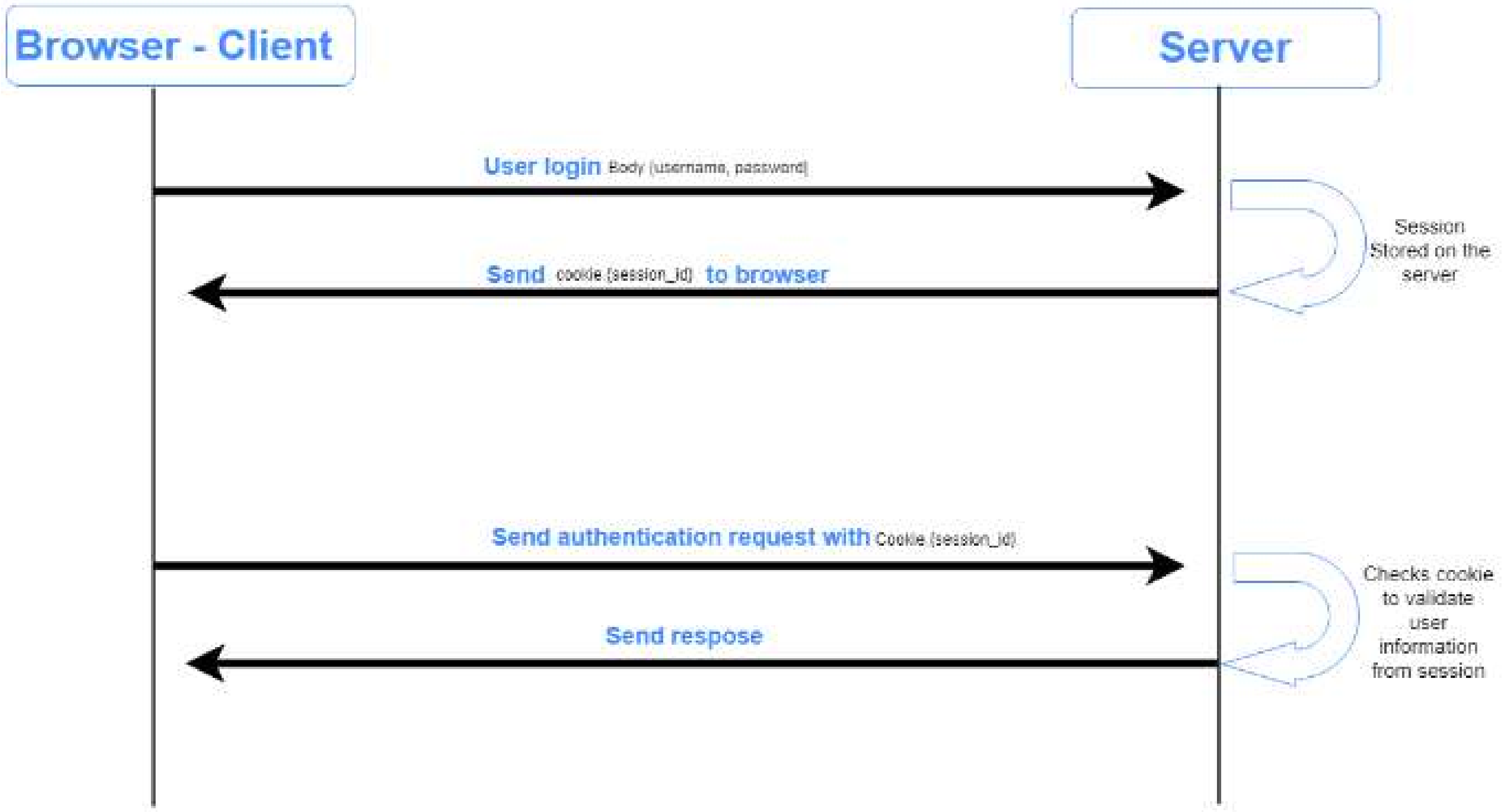
Sessions (stateful)

1. User submits login credentials (username, pass)
2. Server verifies those from the DB
3. Server creates a temporary user session
4. Server issues a cookie with a session ID
5. User sends the cookie with each request
6. Server validates it agains the stored session and grants access
7. When user logs out it destroys the session and clears the cookie

This system is also called stateful because we keep sessions on the server as well as keep data on the client (id).

Stateful Cloud

Cookie-Based


Browser - Client
Server
User login Body (username, password)
Session Stored on the server
Send cookie (session_id) to browser
Send authentication request with Cookie (session_id)
Checks cookie to validate user information from session
Send respose

Tokens (stateless)

Tokens are another way to secure a communication between client and server. This system involves a special string called token to be passed to the client on successful authentication. This token then can be passed with every request to get authenticated with the server. Tokens are not kept by the server. They usually carry data with them as well as a signature and a secret that only the server knows. All of these information are transformed in to a string, some parts are encrypted and it is given to the client. This way everything needed is in the cookie and the server does not need to keep data on it's side. The cookie is the data and the key to authorizing requests. The flow of this system goes:

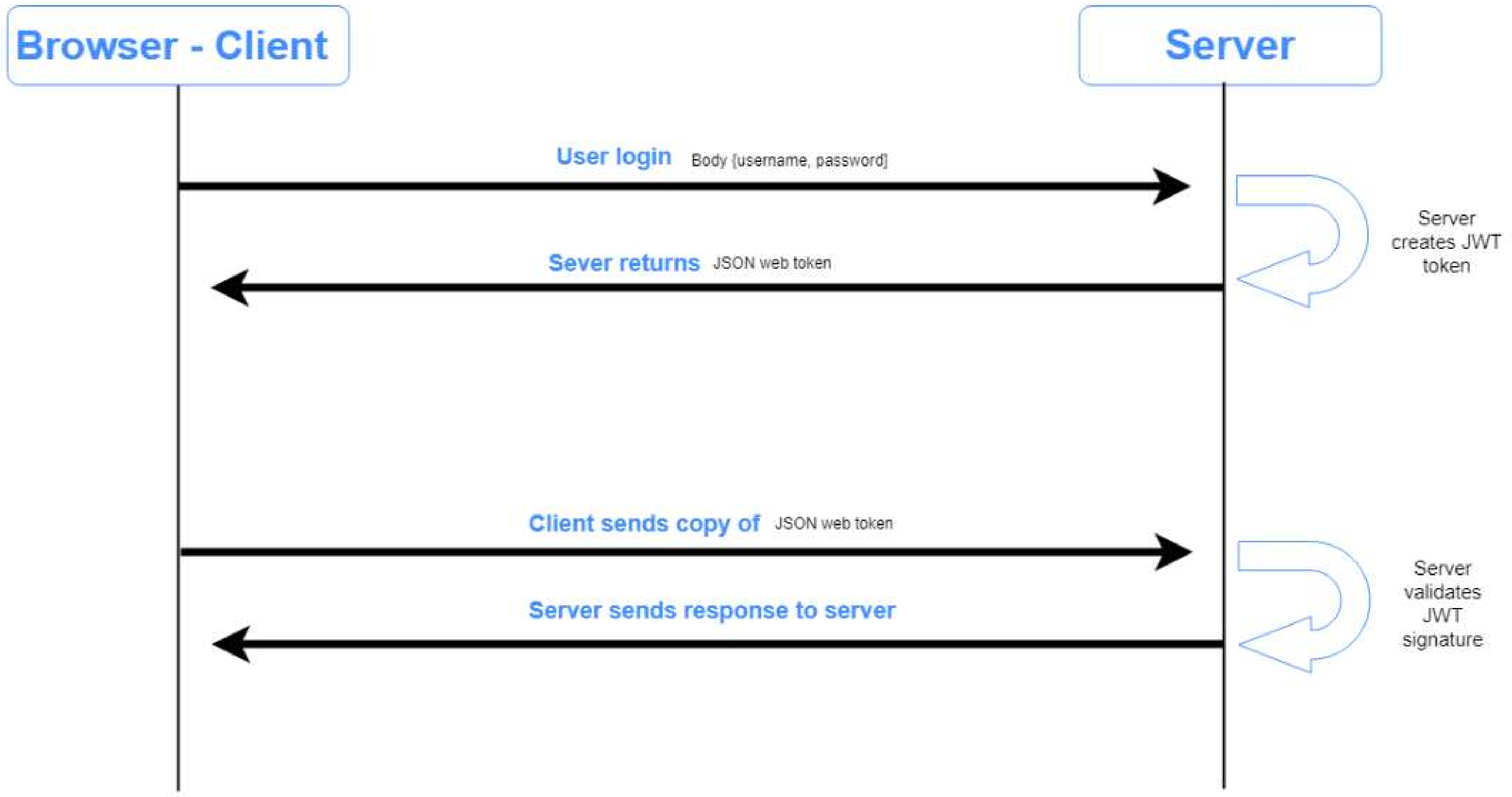
Tokens (stateless)

1. User submits login credentials (username, pass)
2. Server verifies those from the DB
3. Server generates a temporary token and embeds user data into it
4. Server responds back with the token
5. Client stores it (local storage)
6. User sends the token with each request
7. Server verifies the token and grants access
8. On Log-out the token is cleared from client storage

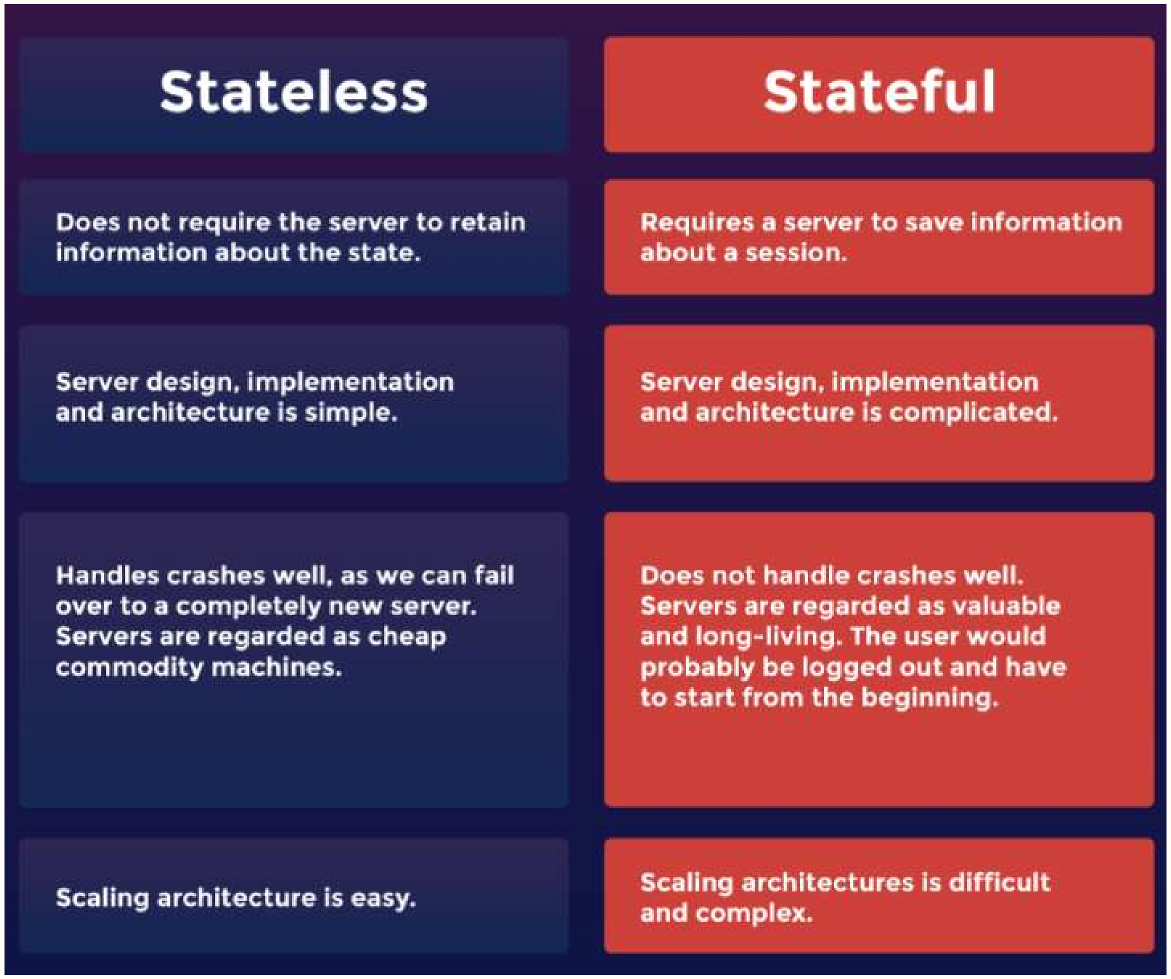
Communication secured by token is usually called stateless and self-contained. This means that the server does not keep any data on the state of the client. It also means that all necessary data is on the token it self.

Stateless Cloud

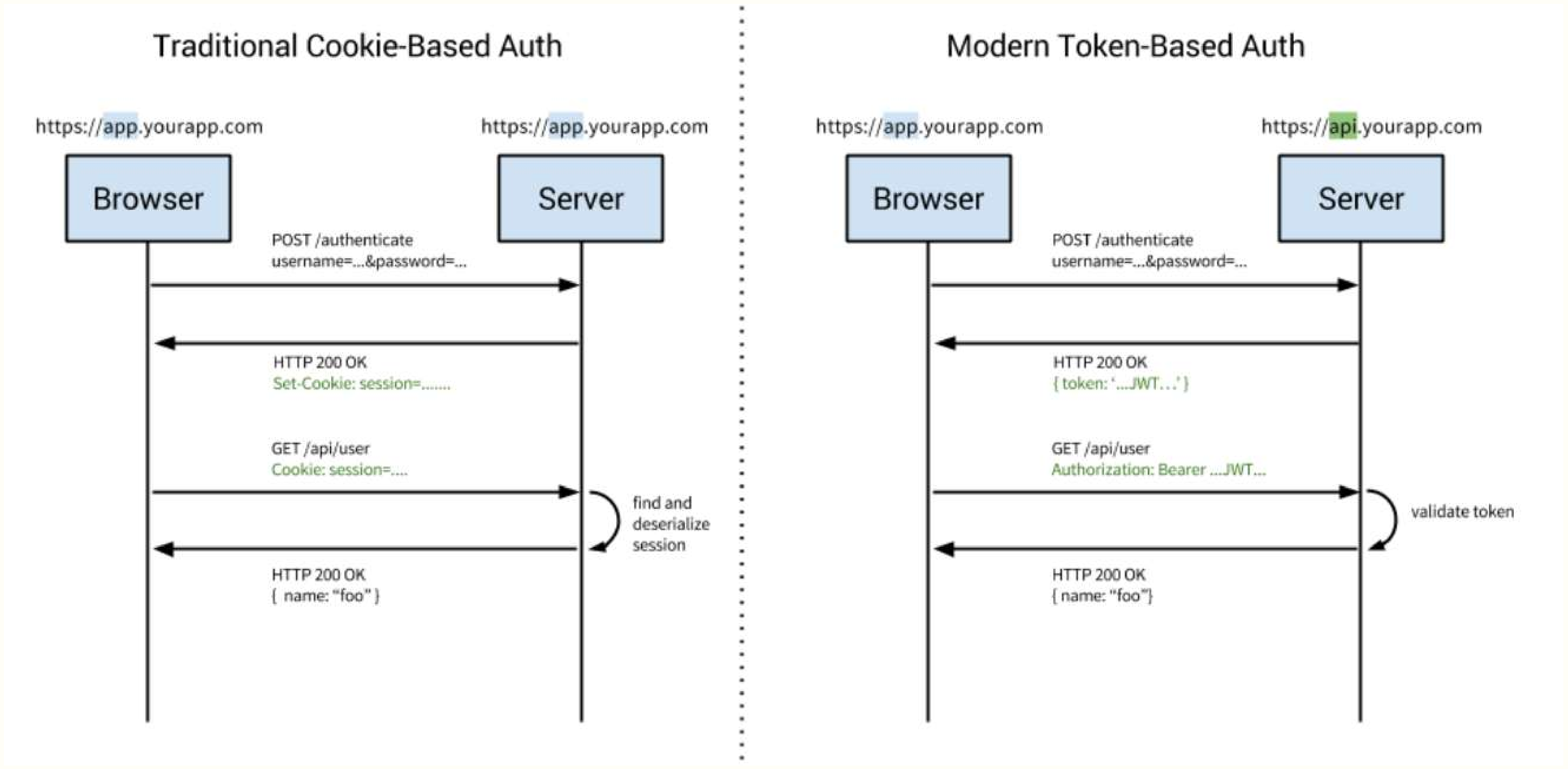
JSON Web Token



Stateful vs Stateless: A Comparison



Stateless	Stateful
Does not require the server to retain information about the state.	Requires a server to save information about a session.
Server design, implementation and architecture is simple.	Server design, implementation and architecture is complicated.
Handles crashes well, as we can fail over to a completely new server. Servers are regarded as cheap commodity machines.	Does not handle crashes well. Servers are regarded as valuable and long-living. The user would probably be logged out and have to start from the beginning.
Scaling architecture is easy.	Scaling architectures is difficult and complex.


Traditional Cookie-Based Auth
https://app.yourapp.com
https://app.yourapp.com
Browser
Server
POST /authenticate
username=...&password=...
HTTP 200 OK
Set-Cookie: session=.......
GET /api/user
Cookie: session=....
find and deserialize session
HTTP 200 OK
{ name: "foo" }
Modern Token-Based Auth
https://app.yourapp.com
https://api.yourapp.com
Browser
Server
POST /authenticate
username=...&password=...
HTTP 200 OK
{ token: '....JWT...' }
GET /api/user
Authorization: Bearer ....JWT...
validate token
HTTP 200 OK
{ name: "foo"}

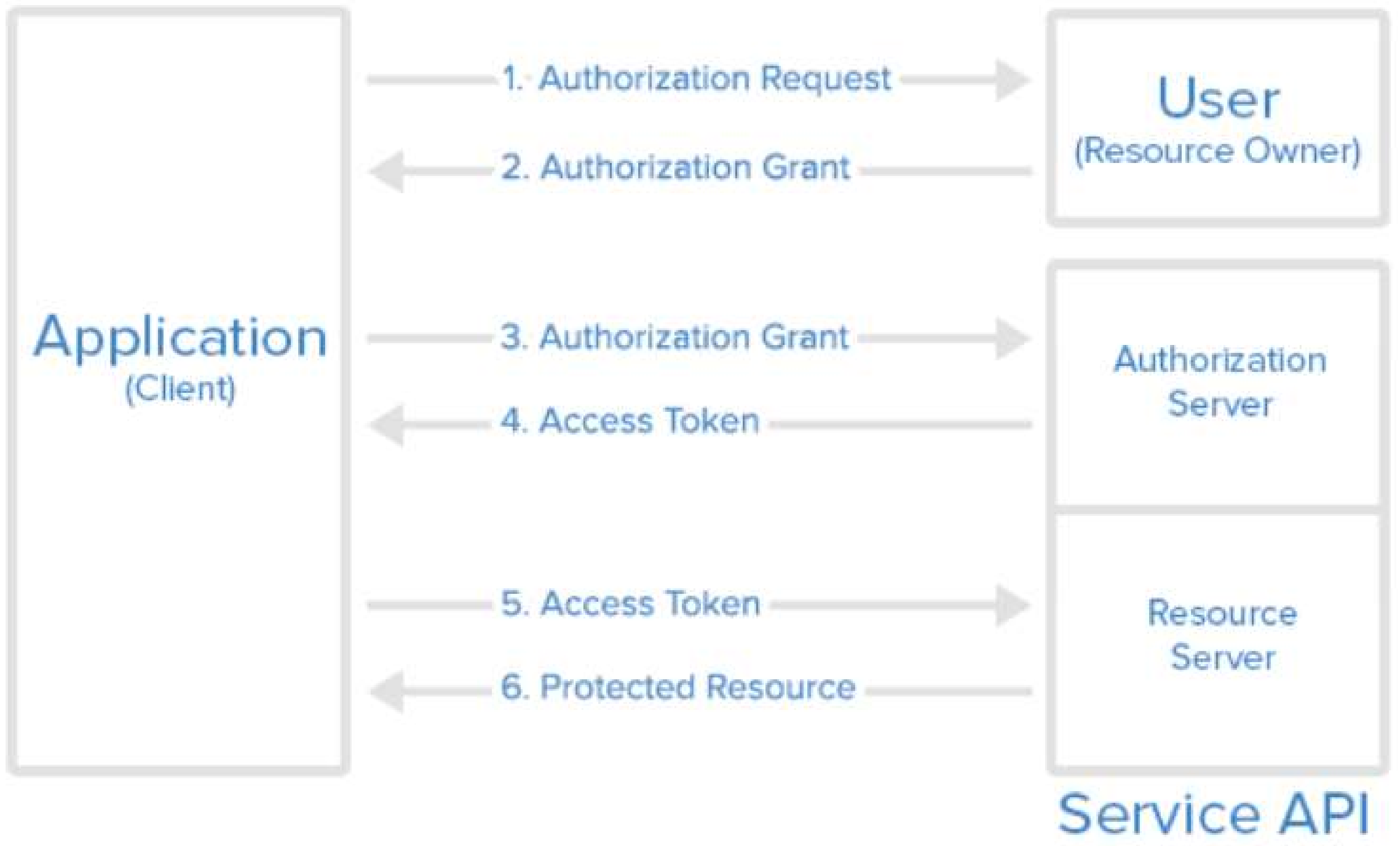
Oauth 2.0

The number of devices and services that we use is growing and every device and service requires authorization. This is why authorization frameworks such as OAuth 2.0 exist. With it our application or service can obtain a limited access to a user account that is enough to authenticate it from another system such as facebook, google, github etc. This is the framework that gives the option to sign in with a facebook/google account instead of creating an accout or logging in. In order for an application to have access this type of authorization from another system it needs to be registered on that system. After registering the system gives the application a unique ID and a secret that will identify that application when making requests. The flow of OAuth authentication usually goes like this:

Oauth 2.0

1. A user click to authorize through another service (ex: google)
2. Google will show an authorization screen to enter username/password (usually this is an authorization API)
3. When the user enters the correct data it grants permission to the application and the application gets an access token
4. The application then requests the user data that the user agreed to share (usually a different resource API)
5. The server validates the access token and gives the application the requested data
6. The application then authorizes the user (ex: with their google account)

Abstract Protocol Flow



Cross-Origin Resource Sharing (CORS)

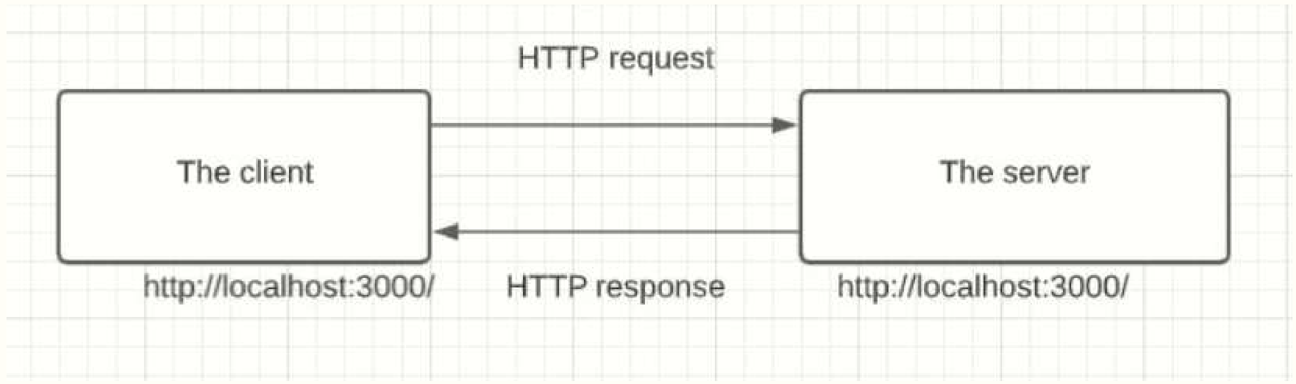
CORS stands for Cross-Origin Resource Sharing. It allows us to relax the security applied to an API. This is done by bypassing the Access-Control-Allow-Origin headers, which specify which origins can access the API.

In other words, CORS is a browser security feature that restricts cross-origin HTTP requests with other servers and specifies which domains access your resources.

How CORS works

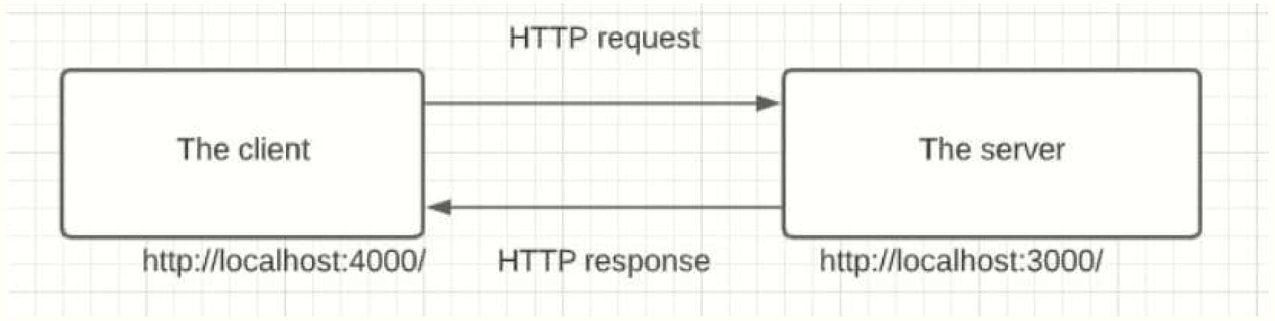
An API is a set procedure for two programs to communicate. This means that API resources are consumed by other clients and servers.

Here are two scenarios:



The client and the server have the same origin. In this example, accessing resources will be successful. You’re trying to access resources on your server, and the same server handles the request.

How CORS works



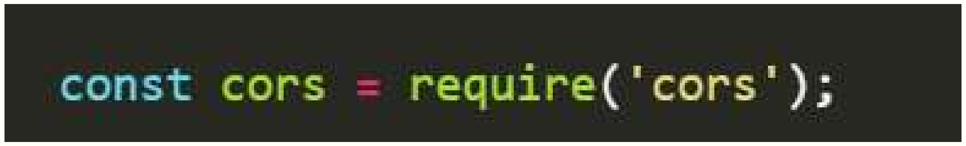
The client and server have a different origin from each other, i.e., accessing resources from a different server. In this case, trying to make a request to a resource on the other server will fail.

This is a security concern for the browser. CORS comes into play to disable this mechanism and allow access to these resources. CORS will add a response header access-control-allow-origins and specify which origins are permitted. CORS ensures that we are sending the right headers.

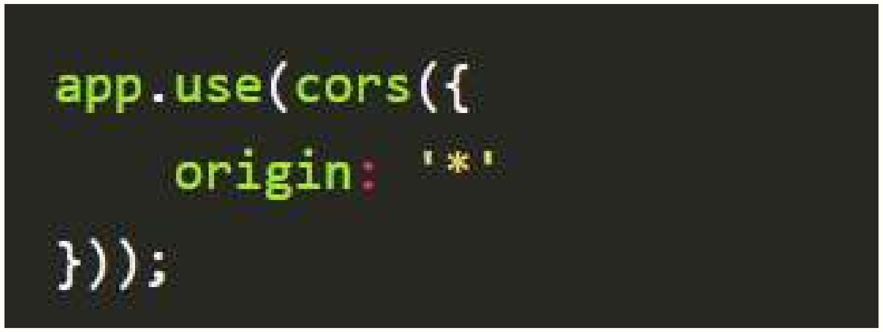
Therefore, a public server handling a public API will add a CORS related header to the response. The browser on the client machine will look at this header and decide whether it is safe to deliver that response to the client or not.

Setting up CORS with Express

First we need to add the CORS header to the server.



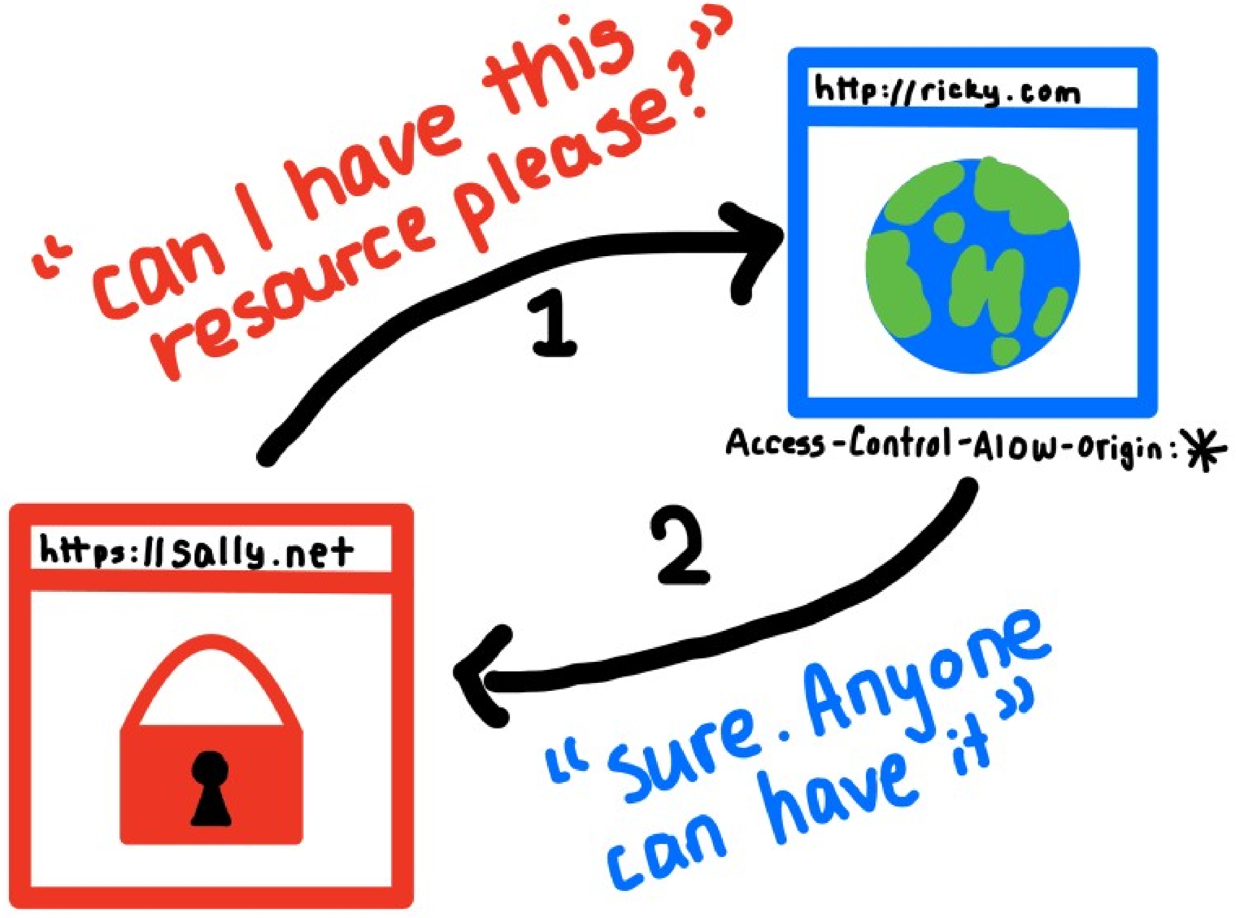
```
const cors = require('cors');
```



```
app.use(cors({
    origin: '*'
}));
```

If the API is public, and any cross-origin APIs and servers can access these resources. The code block above will ensure any page can access the ingredient resources.

CORS


"can I have this resource please?"
1
http://ricky.com
Access-Control-Alow-Origin: *
2
https://sally.net
"sure. Anyone can have it"

QUESTIONS?

You can find us at:

mail

main

Qinshift Academy