***What the hell is Authentication?***

In this blog, we will cover the fundamentals of user authentication in modern web applications and websites. We will explore stateful (session-based) and stateless (token-based) auth, and examine cookies, JWT, and client storage. We will also touch on security aspects with each authentication scheme and lay out practical strategies for implementing authentication in single-page apps and RESTful APIs.

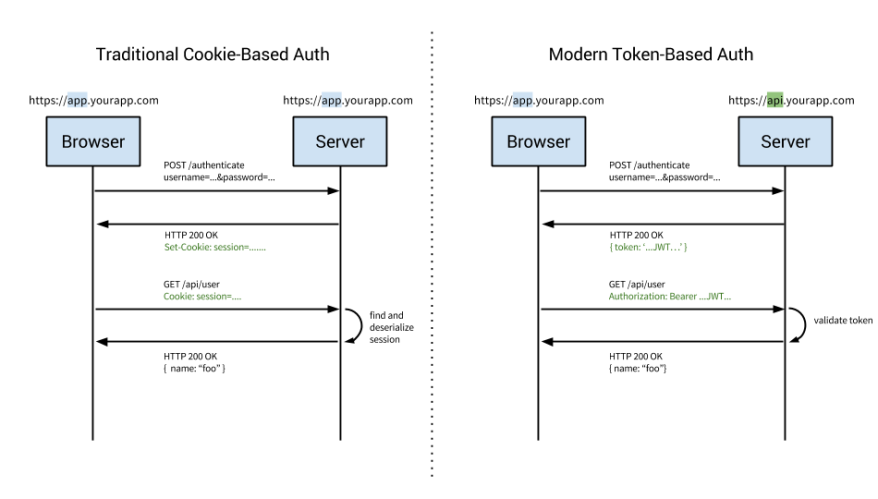
1. Authentication VS Authorization

Nowadays many developers are still confused about those two technical keywords and do not get the difference between them.

**Authentication**: is the process of verifying the identity of a user in your web application. So, it’s essentially determining who the user actually is. It answers the question: Who are you?

**Authorization**: is the process of verifying user permissions, which means what a user is able and not able to do in your app. It answers the question: What can you do?

1. Stateless VS Stateful Authentication



Well in the context of username/password scheme which is the most used scheme in modern webapps. We can distinguish between stateful (storing session\_id in a cookie) and stateless (token using JWT/OAuth with cookie) authentication.

II-1) Stateful authentication

It’s called stateful because sessions are stored in the ***server-side***. Let’s see the authentication ***flow*** in this case:

* user submits login *credentials*, e.g. username & password
* server verifies the credentials from the DB
* server creates a temporary user **session**
* sever issues a cookie with a **session ID**
* user sends the cookie with each request
* server validates it against the session store & grants access
* when user logs out, server destroys the session & clears the cookie

Stateful authentication ***features***:

* Every user session is stored server-side (**stateful**) it could be in:
  + memory (e.g. file system)
  + cache (e.g. Redis or Memcached) which is the best option
  + DB (e.g. Postgres, MySQL)
* Each user is identified by a session ID: no 3rd party script can extract data out using this ID. Only the issuer (in our case: the server) can map back to data using session ID.
* The Session ID is stored in a secret cookie, signed with a secret key. So that only the server can decrypt that cookie.

What is a cookie?



A ***Cookie*** is just a http request header sent by the client to the server. It is used in session-based authentication. It consists of NAME, VALUE, FLAGS/ATTRIBUTES(Optional).

The server set the cookie on the client browser via ***Set-Cookie*** response header. The example above shows a http response sent by the server.

HTTP/1.1 200 OK

Content-type: text/html

Set-Cookie: SESS\_ID=8bKnWqiZvuvYsIV1zmzJQeYUgINqXYeS; Domain=example.com; Path=/

II-2) Stateless authentication

It’s called stateless because the server doesn’t store the user token, everything is stored on the client side.

Let’s see the authentication ***flow*** in a stateless way:

* user submits login *credentials*, e.g. username & password
* server verifies the credentials against the DB
* sever generates a temporary **token** and embeds user data into it
* server responds back with the token (in response body or header)
* user stores the token in ***client storage***
* user sends the token along with each request
* server verifies the token & grants access
* when user logs out, token is cleared from client storage

Stateless authentication ***features***:

* tokens are *not* stored server-side, only on the client (**stateless**)
* *signed* with a secret against tampering that could be verified and trusted by the server
* Typically sent in Authorization request header
* when a token is about to expire, it can be refreshed

Using ***JWT*** standard in stateless authentication:

JSON Web Token (JWT) is an open standard that defines way for securely transmitting information between parties as a JSON object. This information can be verified and trusted because it is digitally signed.