Session Invalidation in a Stateless Architecture

In this lesson, we'll study some session invalidation strategies and how to choose one that's best for your goals.

We'll cover the following

Stateful web architectures are more efficient

If you've ever built a web architecture, chances are that you've heard how stateless architectures scale better due to the fact that they do not have to keep track of state. This is true and represents a security risk, especially in the context of authentication state.

In a typical stateful architecture, a client is issued a session ID which is stored on the server and linked to the user ID. When the client requests information from the server, it includes the session ID, so that the server knows a particular request was made on behalf of a user with a particular ID. This requires the server to store a list of all the session IDs it generated with a link to the user ID, and it can be a costly operation.

JWTs, which we spoke about earlier in this chapter, rose to prominence due to the fact that they easily allow stateless authentication between the client and the server so that the server doesn't have to store additional information about the session. A JWT can include a user ID, and the server can simply verify its signature on-the-fly, without having to store a mapping between a session ID and a user ID.

True stateless architectures are difficult to secure

The issue with stateless authentication tokens (and not just JWTs) lies in a simple security aspect, it is hard to invalidate tokens, as the server has no knowledge of each one generated since they're not stored anywhere. If I logged in on a service yesterday and my laptop gets stolen today, an attacker could simply use my browser and would still be logged in on the stateless service, as there is no way for me to invalidate the previously-issued token.

Solution: take the middle ground

This can be easily circumvented, but it requires us to drop the notion of running a completely stateless architecture, as there will be some state-tracking required if we want to be able to invalidate JWTs. The key here is to find a sweet spot between stateful and stateless, taking advantage of both the pros of statelessness (performance) and statefulness (more control).

Let's suppose we want to use JWTs for authentication, we could issue a token containing a few pieces of information for the user

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkxlY
nJvbiBKYW1lcyIsImlhdCI6MTUxNjIzOTAyMn0.UJNHBHIBipS_agfTfTpqBmyOFaAR4mNz7eOwLOK
UdLk
```

```
$ cut -d'.' -f1 <<< $TOKEN | base64 -d
{"alg":"HS256","typ":"JWT"}%


$ cut -d'.' -f2 <<< $TOKEN | base64 -d
{"sub":"1234567890","name":"Lebron James","iat":1516239022}
```

As you can see, we included an *issued at* (`iat`) field in the token, which can help us invalidate expired tokens. You could then implement a mechanism whereby the user can revoke all previously issued tokens by simply clicking a button that saves a timestamp in a `last_valid_token_date` field in the database.

The authentication logic you would then need to implement for verifying the validity of the token would look like this:

```
function authenticate(token):
  if !validate(token):
    return false

  payload = get_payload(token)
  user_data = get_user_from_db(payload.name)
```

```
if payload.iat < user_data.last_valid_token_date:
  return false


return true
```

Minimizing database hits

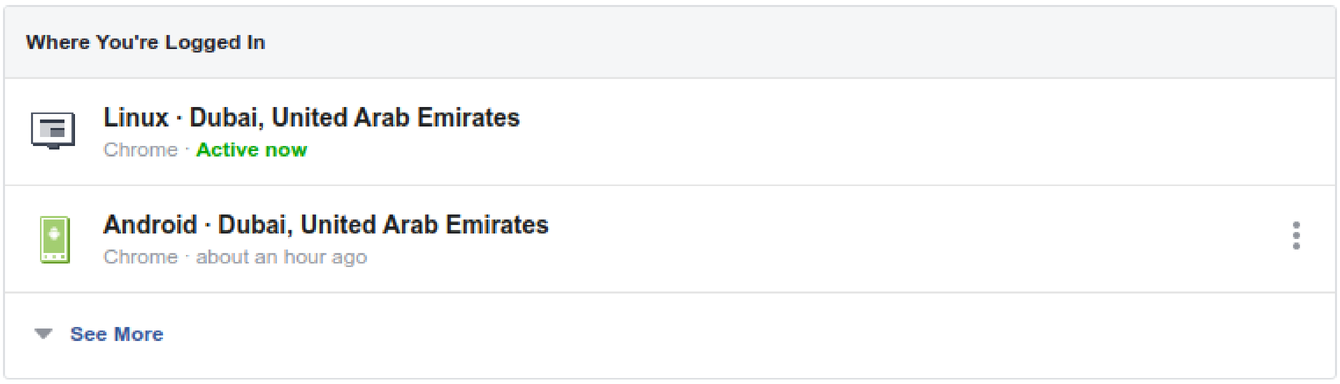
Easy-peasy! Unfortunately, this requires you to hit the database every time the user logs in, which might go against your goal of scaling more easily through being state less. An ideal solution to this problem would be to use two tokens, a long-lived one and a short-lived one (e.g., one to five minutes).

When your servers receive a request:

- If it has the long-lived one only, validate it and do a database check as well. If the process is successful, issue a new short-lived one to go with the long-lived one
- If it carries both tokens, simply validate the short-lived one. If it's expired, repeat the process on the previous point. If it's valid instead, there's no need to check the long-lived one as well

This allows you to keep a session active for a very long time (the validity of the long-lived token) but only check for its validity on the database every N minutes depending on the validity of the short-lived token. Every time the short-lived token expires, you can go ahead and re-validate the long-lived one, hitting the database.

Other major companies, such as Facebook, keep track of all of your sessions in order to offer an increased level of security.


Where You're Logged In

Linux · Dubai, United Arab Emirates
Chrome · Active now

Android · Dubai, United Arab Emirates
Chrome · about an hour ago

Facebook keeps track of all of your active sessions

This approach definitely costs more, but I'd argue it's essential for such a service where the safety of its users' information is extremely important. As we stated

multiple times before, choose your approach after carefully reviewing your priorities and goals.

In the next lesson, we'll study an extra layer of security that protects users if a CDN is compromised.