JSON Web Token (JWT)

Table of Contents

[1 Authorization Strategies 2](#_Toc78546904)

[1.1 Server-Side Sessions 2](#_Toc78546905)

[1.2 Session Drawbacks 3](#_Toc78546906)

[1.2.1 External Session Storage 3](#_Toc78546907)

[1.2.2 In Memory Session Storage 3](#_Toc78546908)

[1.3 TODO 4](#_Toc78546909)

Resources used:

* What is JWT authorization really about - Java Brains: <https://www.youtube.com/watch?v=soGRyl9ztjI>
* What is the structure of a JWT - Java Brains: <https://www.youtube.com/watch?v=_XbXkVdoG_0>
* Sessions vs JWT: <https://medium.com/@vethursan/https-medium-com-vethursan-sessions-vs-jwt-9f3284899f0f>

1. Authorization Strategies

* Server-Side Sessions (Reference token)
* JSON Web Token (Value token)

**Why do we need any of this?**

HTTP is a stateless protocol. Meaning every HTTP command is run independently, it has no recollection of the commands run before it.

This means that whenever a user makes a request that requires authentication to your web application they will have to pass in their credentials. This is not what we want, we don’t want the user to have to log in every time they make a request, we want them to be able to log in once and then continue browsing the site. To do this we need to implement a state mechanism, which brings us to **sessions**.

* 1. Server-Side Sessions

Sessions are a mechanism used to bring **state** to our application. When the user makes their first request to log in, the server creates a session object containing:

1. Text

   Description automatically generateda unique session id
2. session expiry date
3. information about the user such as the user id
4. and any other information you may want to store. (Information you will need to perform future requests.)

The server can either store this information into a database/cache, or in memory. (Both come with some drawbacks that will be explained below.)

Diagram, timeline

Description automatically generatedWhen the user makes their first request to login you validate their credentials, create a session and store it. Then in the response you send back the session id in a cookie.

Now whenever the user makes a request you check for the session cookie, validate it, retrieve the session from your data store and perform the request.

The user only has to log in once and the rest of the requests can be validated automatically using the session.

* 1. Diagram

     Description automatically generatedSession Drawbacks

There are a few problems with this approach. The biggest problem is that this approach assumes that our application is always just a monolith. This used to be the case in the past but nowadays we generally have multiple application servers scaled horizontally.

A picture containing text, whiteboard

Description automatically generatedIf each of these apps have their own data storage, then we run into another problem. A user request will create a new session in server 1 but if that user is routed to server 2 on the next request, server 2 won’t be able to use the same session because they have separate databases.

* + 1. External Session Storage

The solution is obvious (Centralized Storage), you use a shared session cache (or a database like MySQL which is shared by all your app instances). This is a typical usecase for Redis Session store.

* **Cache vs Session Store**: <https://redislabs.com/blog/cache-vs-session-store/>

The drawbacks (centralized storage) are:

* **Signle point of failure**: if your DB or Redis goes down, then you won’t be able to authenticate users.
* **It becomes the bottleneck of your application**: All your requests have to query this central store which increases latency and limits how much we can horizontally scale.

The solution (Data Replication) these drawbacks is that we can replicate our session data across all your servers to prevent bottle-necking the application (or single point of failure). Data replication is expensive, has a lot of overhead and increases the complexity of your application.

* + 1. In Memory Session Storage (Sticky Session Pattern)

You might be thinking, why not just store the session data on the server itself? You can just store the session objects on your server and grab it from there. You don’t need an extra data store and you won’t be funneling all the requests into one place.

This works, if you only have one web server. Imagine you have multiple servers, you have to make sure that every single one of the users requests always hit the same server (Sticky session pattern) or else you can’t get their session.

* **Sticky session pattern**: The load balancer remembers which server has the given session for a user and it always redirects to that server.

This isn't easy because there are many networking issues you would have to handle. (e.g client roaming between networks.)

And also, isn’t scalable. Image that you have multiple microservices contained in one app. How does the session information get carried over between all those microservices?

Even if you do get past these issues, what happens when a server crashes or goes down for maintenance? Memory is reset every time the application relaunches. Do you lose all those sessions?

* 1. JSON Web Token (Value token)

TODO <https://www.youtube.com/watch?v=_XbXkVdoG_0>

* 1. TODO

TODO - How to deal with JWT expiration? : <https://gist.github.com/soulmachine/b368ce7292ddd7f91c15accccc02b8df>