TL;DR *Many modern web applications use JSON Web Tokens (JWT), rather than the traditional session-based authentication. Quite a few challenges have been found with using server-side sessions in modern-day applications. In this post, we'll identify those challenges and explain how JWT and sessions work in practice.*

There was a bit of controversy recently about the use cases where JWT really shines, and the ones where it doesn't do such a great job. Namely, are JSON web tokens good enough for sessions – *or should we keep using cookies instead?*

Prosper from **Auth0** approached us about writing a guest post on Pony Foo, and he'll be addressing exactly those questions. 🍪

— Editor's note.

What Are JSON Web Tokens?

JSON Web Token (JWT) is an open standard (RFC 7519) that defines a compact and self-contained way to securely transmit information between parties as a JSON Object. This information can be verified and trusted because it is digitally signed. JWTs can be signed using a secret (with HMAC algorithm) or a public/private key pair using RSA.

JWT Anatomy

JWTs basically consist of three parts separated by a `.` . This is the header, payload and signature. Check out this excellent article for a comprehensive explanation of the JWT Structure.

How JSON Web Tokens Work

In authentication, when the user successfully logs in using his credentials, a JSON Web Token will be returned and must be saved locally (typically in local storage, but cookies can be also used), instead of the traditional approach of creating a session in the server and returning a cookie.

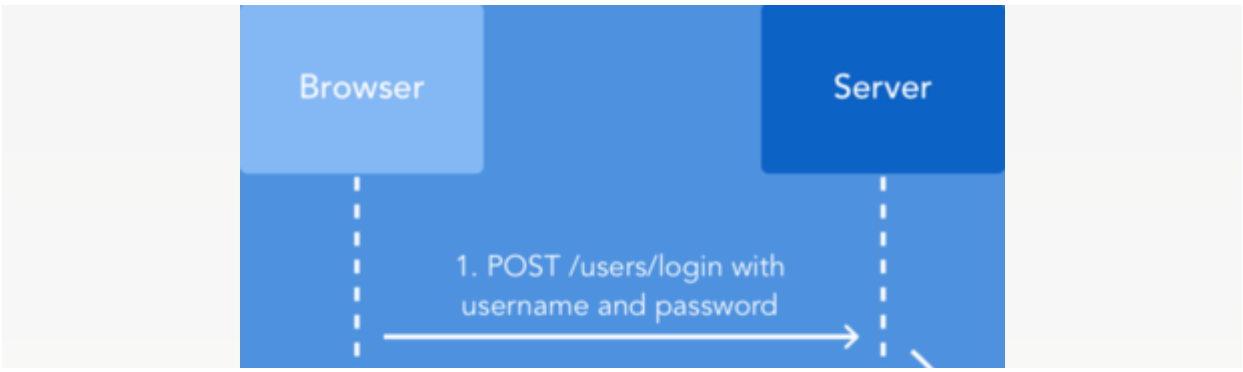
Whenever the user wants to access a protected route, it should send the JWT, typically in the Authorization header using the Bearer schema. Therefore, the content of the header should look like the following:

```
Authorization: Bearer <token>
```

This is a stateless authentication mechanism as the user state is never saved in the server memory. The server's protected routes will check for a valid JWT in the Authorization header, and if it is there, the user will be allowed. As JWTs are self-contained, all the necessary information is there, reducing the need to go back and forth to the database.

This allows the user to fully rely on data APIs that are stateless and even make requests to downstream services. It doesn't matter which domains are serving your APIs, as Cross-Origin Resource Sharing (CORS) won't be an issue since it doesn't use cookies.

Authentication flow



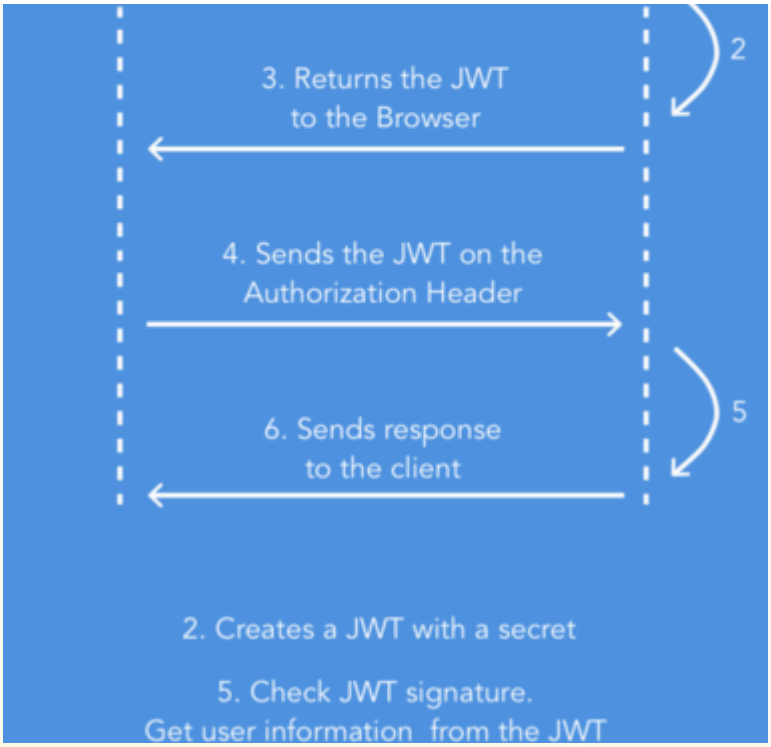


image description

Why Should You Use JWTs

There are several reasons that you should use JSON Web Tokens.

◯ They are easy to scale horizontally
◯ They are easier to maintain and debug
◯ They have the ability to create truly RESTful Services
◯ They have built-in expiration functionality.
◯ JSON Web Tokens are self-contained.

The points highlighted above will be explained in detail in the next section.

JWTs vs. Sessions

Before the emergence of JSON Web Tokens, we had the predominant server-based authentication. As we all know, HTTP Protocol is stateless, this means that if we authenticate a user with a username and password, then on the next request, our application won't know who we are. We would have to authenticate again. So there was a need to ensure that after a user has

logged in, the user's authentication status can still be verified on every subsequent HTTP request.

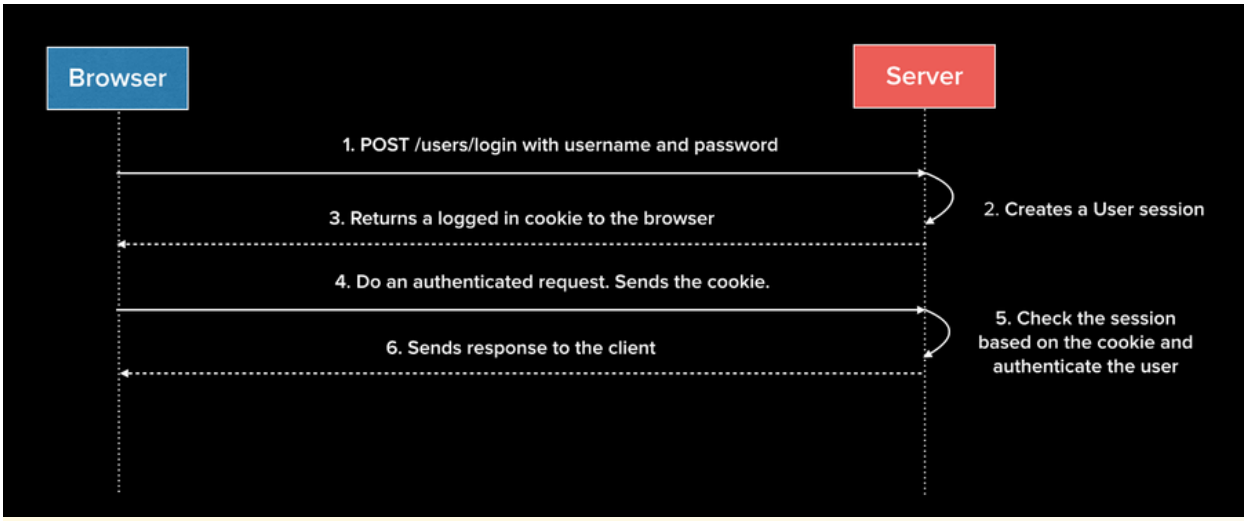


image description

A user's credentials are sent as a POST request to the server. The server authenticates the user. If the credentials are valid, the server responds with a cookie, which is set on the user's browser and includes a **SESSION ID** to identify the user. The user sessions are stored in memory either via files or in the database on the server. In this section, I'll elaborate on several points that will be used as a basis for comparing JWTs with sessions in practice.

1. Scalability: As your application grows and your user base increases, you'll have to start scaling either horizontally or vertically. Session data is stored in memory on the server either via files or in a database. In a horizontal scaling scenario, where you have to start replicating servers, you have to come up with a separate central session storage system that all of your application servers have access to. Otherwise, you won't be able to scale your application because of the session-store drawback. Another way to solve this challenge is to use the concept of sticky sessions. You can also store your sessions on disk to make your application easy to scale in a cloud environment. These types of workarounds don't really play well with modern large applications. Setting up and maintaining this type of distributed system involves in-depth technical knowledge and subsequently incurs higher financial costs. Using JWTs, in this case, is seamless; there is no need to store user information in the session since token-based authentication is stateless. Our application can scale easily because we can use tokens to access resources from different servers without worrying if the user was actually logged in on a particular server. You also save costs because you don't need a dedicated server to store your sessions. Why? Because there are no sessions!

Note: If you are building small applications that absolutely don't need to scale up to running on multiple servers and have no need for RESTful APIs, sessions will definitely work fine for you. And if you can use a dedicated server to run a tool like Redis for your session storage, then sessions might also work perfectly for you!

2. Security: Signing JWTs already aim to prevent tampering on the client side, but they can also be encrypted to ensure that the claim that the token carries is very secure. Now, JWTs are mostly either directly stored in web storage (local/session storage) or in cookies. And JavaScript has access to web storage on the same domain. This simply means that your JWTs might be vulnerable to XSS (Cross-site Scripting). Malicious JavaScript can be embedded on a page to read and compromise the contents of your Web Storage. In fact, a lot of people advocate that very sensitive data shouldn't be stored in Web Storage because of XSS attacks. A very typical example is ensuring that your JWTs are not encoded with very sensitive/trusted data, such as a user's Social Security Number.

Initially, I mentioned that JWTs can be stored in cookies. In fact, JWTs are stored as cookies on many occasions, and cookies are vulnerable/susceptible to CSRF (Cross-site Request Forgery) attacks. One of the many ways to prevent CSRF attacks is to ensure that your cookie is accessible by only your domain. As a developer, ensure that necessary CSRF protections are put in place to avoid these attacks, regardless of the use of JWTs.

Now, JWTs and session ids can also be exposed to unmitigated replay attacks. It is totally up to the developers to establish what replay-mitigation techniques are appropriate for their systems. One way of solving this problem is ensuring that JWTs rely on short expiration times. Although, this technique doesn't totally solve the problem. However, other alternatives for solving this challenge are issuing JWTs to specific IP addresses and using browser fingerprinting.

Note: Use HTTPS/SSL to ensure that your cookies and JWTs are encrypted by default during client and server transmission. This helps avoid man-in-the-middle attacks!

3. RESTful API Services: A common pattern for modern applications is to retrieve and consume JSON data from a RESTful API. Most applications these days have RESTful APIs for other developers or applications to consume. Serving data from an API has several distinct advantages, one of them which is the ability for data to be used in more than just one application. The traditional approach of using sessions and cookies for the user's identity doesn't work well in this case because they introduce the **state** to the application.

One of the tenets of a RESTful API is that it should be stateless, which means that when a request is made, a response within certain parameters can always be anticipated without side effects. A user's authentication state introduces such a side effect, which breaks this principle. Keeping the API stateless and therefore without side effects means that maintainability and debugging are made much easier.

Another challenge here is that it is quite common for an API to be served from one server and for the actual application to consume it from another. To make this happen, we need to enable Cross-Origin Resource Sharing (CORS). Since cookies can only be used for the domain from which they originated, they aren't much help for APIs on different domains than the application. Using JWTs for authentication in this case ensures that the RESTful API is stateless, and you also don't have to worry about where the API or the application is being served from!

4. Performance: A critical analysis of this is very necessary. When making requests from the client to the server, if a lot of data is encoded within the JWT, it creates a significant amount of overhead with every HTTP request. However, with sessions, there is only a tiny amount of overhead because **SESSION IDs** are actually very small. Look at this example below:

A JWT has 5 claims like so:

```
{
  "sub": "1234567890",
  "name": "Prosper Otemuyiwa",
  "admin": true,
  "role": "manager",
  "company": "Auth0"
}
```

When encoded, the size of the JWT will be several times the size of a **SESSION ID (identifier)**, thus making this JWT add more overhead than a **SESSION ID** with every HTTP request. With sessions, there is also a server side lookup to find and deserialize the session on each request.

JWTs trade size for latency by keeping the data on the client side. The data model of your application is a significant factor here because latency is saved by preventing incessant calls and queries to the database on the server. The idea here is to be careful not to store too many claims in a JWT to avoid huge, over-bloated requests.

Worthy of mention is the fact that tokens may require access to the database on the backend. This is particularly the case for refresh tokens. They may require access to a database on the authorization server for blacklisting. Get more info about refresh tokens and when to use them. Also, check out this article for more information on blacklisting.

Note: Developers need to strike a balance to make the usage of JWTs really worth it!

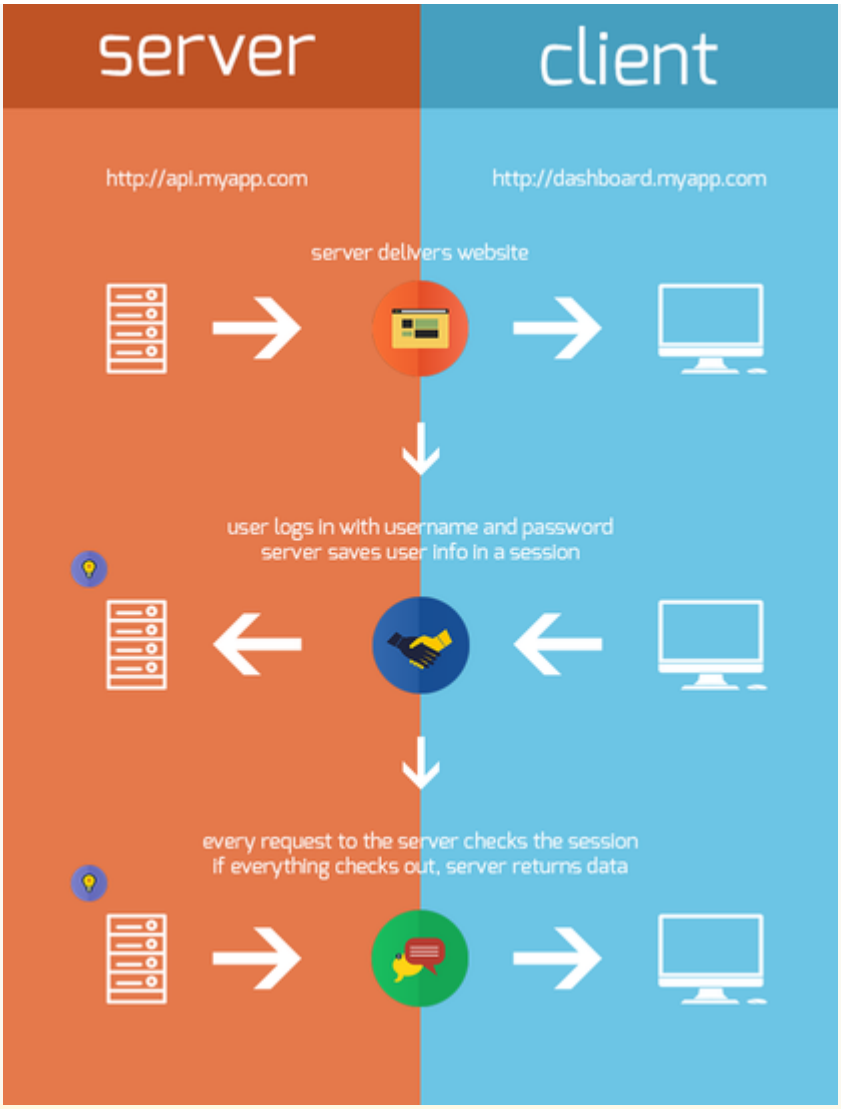


image description

Source: Quora

5. Downstream Services: Another common pattern seen with modern web applications is that they often rely on downstream services. For example, a call to the main application server might make a request to a downstream server before the original request is resolved. The issue here is that cookies don't flow easily to the downstream servers and can't tell those servers about the user's authentication state. Since each server has its own scheme for cookies, there is a lot of resistance to flow, and connecting to them is difficult. JSON web tokens again makes these a breeze!

Authentication with Auth0 using JWTs

In **Auth0**, we issue JWTs as a result of the authentication process. When the user logs in using Auth0, a JWT is created, signed, and sent to the user. **Auth0** supports signing JWT with both HMAC and RSA algorithms. The user has the flexibility to actually choose any of these two algorithms from the dashboard. This token will be then used to authenticate and authorize with APIs, which will grant access to their protected routes and resources.

We also use JWTs to perform authentication and authorization in Auth0's API v2, replacing the traditional usage of regular opaque API keys. Regarding authorization, JSON Web Tokens allow granular security, which is the ability to specify a particular set of permissions in the token, thus improving debuggability.

Conclusion

JSON Web Tokens (JWTs) are lightweight and can easily be used across platforms and languages. They are a clever way to authenticate & authorize without sessions. There are several JWT libraries available for signing and verifying the tokens. There are also many reasons to use tokens, and Auth0 can help implement token authentication in an easy and secure way.

Personally, I don't think there is a one-size-fits-all approach. It will always depend on your application architecture and use case.

Would you still prefer using sessions in practice over JSON Web Tokens? Let us know your thoughts in the comment section!