**JWTs vs. sessions: which authentication approach is right for you?￼**

**All**

**All about auth**

**May 19, 2022**



Your application just received a login request, and the credentials passed successfully prove the identity of a user in your system. Wonderful, you have a high degree of confidence in who this user is and what they should be able to access! …but wait, what happens on the next API call where they don’t include their login credentials? HTTP is stateless, meaning that each request is independent and doesn’t contain any context about previous requests, but asking your user to re-authenticate on every request isn’t exactly a friendly UX.

Session cookies and JSON Web Tokens (JWTs) are the two most popular ways to maintain this authentication state between calls. There are pros and cons to both, and choosing between them requires understanding these tradeoffs and how they relate to the specific needs of your application.

**Session-based authentication**

In session-based authentication (also known as cookie-based authentication), the server is responsible for creating and maintaining a record of the user’s authentication and providing a way for the client to reference that record in each subsequent request.

This flow starts by the user authenticating and providing some credentials to the server for verification. If the credentials are accepted, the server will create a persistent record that represents this authenticated browsing session. This record will have some sort of primary identifier (typically a random string that is at least 128 bits long) in addition to an identifier for the user, the time the session started, the expiry of the session, and perhaps context information like the IP address and User Agent. This information will be stored in the database, and the session identifier will be sent back to the client to be stored as a cookie in the user’s web browser.

Each subsequent request from the browser will include the session cookie in the HTTP headers, which the server can then use to look up the session record, confirm that it is valid and then make authorization decisions about what information to return based on the confirmed identity of the user.

**The benefits of session cookies**

The appeal of this approach is in its simplicity and reliability.

The database record of the session serves as a clear, centralized source of truth for the state of the session, which allows for a high degree of confidence that the session information is up to date and can be used to make authorization decisions. Revoking a user’s access to the system is quick and reliable with sessions, as you can simply delete the session record from the database or mark it as invalid. For any subsequent requests after revocation, the server will fail to find a valid session that matches the identifier in the headers and will return a 401 unauthenticated error to prompt the user to re-authenticate.

By offloading the state management to the server, we are able to reduce the data transfer overhead down to a single opaque string, which is lightweight and does not leak any information about the associated user or the context of the session.

**The downside of session cookies**

While session-based authentication is very reliable, at scale it can begin to introduce latency and performance issues.

Since you need the session record to be highly reliable and accessible from any host, this means inserting a write request to the database for every authentication and more importantly a read request to the database for every subsequent request that contains the session header. Since session expiry is often extended with constant use, this can also mean an additional update on every request. Over time all these database interactions can add up, and introduce notable latency across your application.

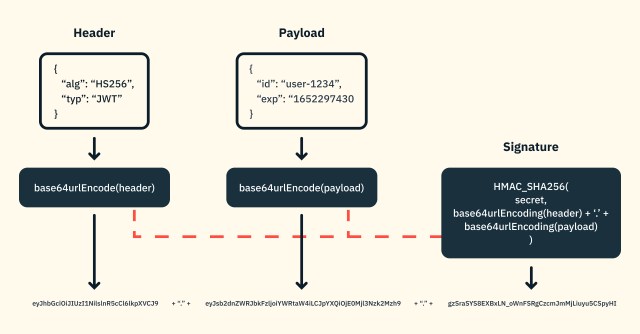
For applications that have highly dynamic clients, this latency overhead might not be worth the benefits that session-based authentication provides.

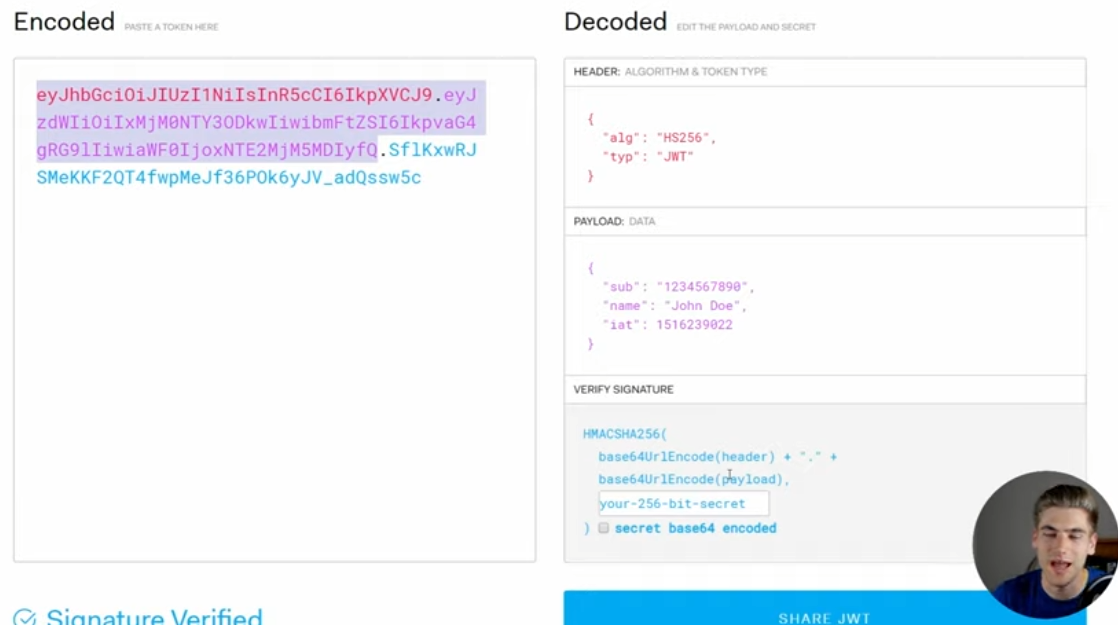
**JWT authentication**

JSON Web Tokens (JWTs) achieve similar goals of identifying and authorizing the logged-in user during subsequent requests, but solve the problem of how to manage that information in a very different way.

This flow also starts with the user providing some form of credentials that the server uses to authenticate that particular request. However, while the session-based flow relies on storing all the necessary state in a database and looking it up on every request, in the JWT flow all that context is self-contained in the string being sent back to the client.

At a high level JWTs are JSON objects that follow a particular protocol for communicating “claims” or authorization context, and are then either signed or encrypted by the issuing server in order to provide assurance that these claims can be trusted.

*Clients can verify that the JWT has not been tampered with since the signature of the JWT contains the original header and payload data.*



JWTs consist of three parts – the header, the payload and the signature.

The payload contains the core claims, such as the identity of the user the JWT was issued for, the permissions that the token might grant, and the expiry of the JWT, which indicates the time after which the JWT should no longer be accepted. The header contains information about the algorithm used to sign or encrypt the token. The header and payload are then base64url encoded, which makes the value easier to send and store. Since this is just as easy to decode, it means that the information stored in them can be viewed by anyone.

The signature is created by combining the header and the payload and then hashing that combination with a secret key, providing a way to detect if a malicious actor has tampered with the claims after the issuer signed the JWT. In distributed systems, this is typically an asymmetric signature, where the issuing server will use a private key to hash the contents and then the audience can use the corresponding public key to verify that the current payload is the same one that was signed by the issuer.

Once the JWT is constructed and signed, it is sent back to the client to store. This JWT can be used safely for authorization by verifying that the expiry has not passed and the signature is valid for the payload provided, all of which can be done on the client without checking with the server who initially issued the JWT.

**The benefits of JWTs for authentication**

JWTs contain all the information required to both verify the authenticity of the claims, as well as the information you’d need about the user to make authorization decisions. This self-contained quality of JWTs means that there is no longer a dependency on the server and database in order to validate the token and feel confident in making authorization decisions for the identified user.

There are several advantages to this, most obviously, a reduction in latency for your application since client-side authorization is possible and server-side authorization can happen much faster without a call to the database.

The other advantage is that it opens up a wider range of possible applications that can sign, verify and leverage the identity information and authorization granted through the JWT. Signatures and self-contained data make it possible to develop server-to-server applications that programmatically self-sign tokens and refresh them without needing manually entered credentials. In addition, the flexibility of the claims allows you to communicate other important information to these external applications within the token itself. This can be very useful when exposing APIs to external applications.

**The downside of JWTs for authentication**

The self-contained, stateless nature of JWTs has a significant downside though – once a JWT is signed, there is no way to invalidate the JWT or update the information contained within it. Provided the signature is valid and the expiry timestamp has not passed, the JWT will be considered valid by any process leveraging it for authorization decisions.

If a user requests to log out of all devices, there is no practical way to honor this request through local validation before the natural expiry of all currently issued JWTs. In theory, JWTs can also be invalidated by revoking the secret key used to sign the JWT but that would invalidate all JWTs that used that key and would require handling to evict any cached validation keys, making secret key revocation an unsustainable option for something as simple as a user log out.

Similarly, in the case where the JWT contains role-based authorization information (such as “admin” vs “member”), if the user is downgraded to a lower role that reduces the scope of what they are allowed to access, this change in authorization permissions would not be reflected until their existing JWTs expired.

**JWTs versus sessions cookies**

As we have seen, both JWTs and session cookies are viable approaches to solving the issue of persisting authentication and authorization context in a stateless HTTP world, but they take fairly different approaches that have their own pros and cons.

JWTs enable faster authorization and more interoperability with external apps, but they demand more developer investment to address their security complexities, and might not be the best fit for applications that enable access to sensitive data or actions.

On the other hand, while sessions provide stronger guarantees that each individual request is authorized and are simpler to implement securely, their bottleneck on the server-side database validation comes with a latency overhead that might ruin the user experience for highly responsive applications.

While there are maximalists out there who will tell you that one approach is always superior, at Stytch we recognize that every application is unique and the security and latency tradeoffs need to be evaluated in context.

**Using a combination of session cookies and JWTs**

For those looking to blend the performance benefits of JWTs and the security benefits of session cookies, Stytch’s session management product offers a powerful hybrid option that can be customized for your particular needs.

An extremely security-conscious organization, like a bank or government agency, might want to just use session cookies in order to ensure that every single call is authorized at that exact moment. Other applications might want the performance improvements of being able to do client-side JWT validation but do have some sensitive actions that require  a source of truth to check in with before granting access. Applications without any sensitive info might value the performance benefits of JWTs and almost never want to ask their users to re-authenticate, but still want to be able to honor explicit logout requests.

When you’re using Stytch session management, you can configure the duration of a user’s session, which defines how long Stytch will keep the session active before prompting the user to re-authenticate. Stytch will return both a session\_token, which is a static value that is good for the lifetime of the session, as well as a JWT that is associated with the underlying session but has its own, shorter lived, expiry of 5 minutes. Expired JWTs can be passed to the Stytch session API in order to retrieve a fresh JWT, and the Stytch servers will ensure that the underlying session is still active before passing back a new JWT. If the user logs out, this revocation of access will take place within a maximum of 5 minutes. If you have a particularly sensitive route, you can configure the Stytch client libraries to have an even more restrictive max\_token\_age which Stytch will use to request a new JWT, lowering the threshold for potential staleness even further.

This solution allows for a nice balance between performance and security. Getting started with session cookies is easy and secure. But as your app scales, you may start to notice the latency of the required API call. JWTs can help you reduce the number of calls you need for non-sensitive routes. If the API call only needs to happen every 5 minutes or before granting access to particularly sensitive actions, this greatly reduces the performance overhead while also protecting you and your end users from the risk of authorizing actions based on stale information.

**Conclusion**

While there may never be a clear consensus on which method is superior, the good news is that Stytch provides both options, with the ability to configure the perfect blend of latency vs security for your particular use case. Stytch is a passwordless authentication platform that lets you abstract over complexity and enables developers to choose what works for them. This choice includes the ability to [switch between JWTs and session cookies](https://stytch.com/products/session-management) as needed.

Regardless of your developers’ favored approach, Stytch makes authentication easier and more secure. [Read more](https://stytch.com/) about how Stytch can provide simple and secure implementations out of the box, or [take a look](https://stytch.com/docs/example-apps) at how you can use the Stytch platform to build frictionless, secure authentication flows.

**SHARE**

**Get started with Stytch**

Sign up or talk to an auth expert to learn how you can improve conversion, retention, and security with Stytch.

[**Talk to an auth expert**](https://stytch.com/contact/sales/)

**Products**

* [Email Magic Links](https://stytch.com/products/email-magic-links/)
* [Embeddable Magic Links](https://stytch.com/products/magic-links/)
* [Passwords](https://stytch.com/products/passwords/)
* [One-Time Passcodes](https://stytch.com/products/one-time-passcodes/)
* [0Auth Logins](https://stytch.com/products/oauth/)
* [Web3 Logins](https://stytch.com/products/web3/)
* [Session Management](https://stytch.com/products/session-management/)
* [WebAuthn](https://stytch.com/products/webauthn/)
* [Native Mobile Biometrics](https://stytch.com/products/mobile-biometrics/)
* [TOTP Authenticator Apps](https://stytch.com/products/authenticator-passcodes/)

**Use cases**

* [SaaS](https://stytch.com/use-cases/saas/)
* [eCommerce](https://stytch.com/use-cases/ecommerce/)
* [FinTech](https://stytch.com/use-cases/fintech/)
* [Web3](https://stytch.com/use-cases/web3/)
* [Consumer tech](https://stytch.com/use-cases/consumer-tech/)
* [Healthcare](https://stytch.com/use-cases/healthcare/)

**Resources**

* [Docs](https://stytch.com/docs)
* [Pricing](https://stytch.com/pricing/)
* [Customer stories](https://stytch.com/customer-stories/)
* [Blog](https://stytch.com/blog/)

**Company**

* [About](https://stytch.com/about/)
* [Careers](https://stytch.com/careers/)
* [Contact](https://stytch.com/contact/)

© 2022 Stytch. All rights reserved.

[Terms of use](https://stytch.com/legal/terms-of-service)[Privacy policy](https://stytch.com/legal/privacy-policy)