E-Book

JSON Web Tokens (JWTs) Are Not Safe

An in-depth guide to why security experts believe JWTs aren't safe for user sessions—and a battle-tested alternative

redis

Table of Contents

Chapter 1:
HTTP Sessions, Authentication, and Authorization 3
The use case 3
1. Where to store the session data (client vs. database) 4
2. How to send session data to the client 5
3. How the client can send session tokens to the server for future requests 5
4. How the server can handle authentication and authorization 6
5. When will the session expire 6
Section summary 7

Chapter 2:
Storing Sessions in a Traditional Database 7
The main problem with this approach: 8
There are two ways to solve this problem: 8
› Option 1: Eliminate database lookup (step four): 8
› Option 2: Make the database lookup so fast that the additional call won't matter 9

Chapter 3:
Storing Sessions in JWT 9
Token expiration 11
JWT is too liberal for a security spec and so is vulnerable 11
› The "none" algorithm 12
› The algorithms are passed in an array 13
› Claims are optional 13
Other considerations and issues 14
› Length of tokens 14
› The state needs to be maintained anyway (for rate-limiting, IP-whitelisting, etc.) 14
So why is JWT dangerous for user authentication? 15
Still trying to make JWT work? 15
Bottom line 16
When can I use JWT? 16
If I can't use JWT, what else can I do? 17

Chapter 4:
Storing Sessions in Redis 17
An example code snippet (Node.js) 19

Chapter 5:
Sessions When Redis Is Your Primary Database 20
Is anyone using this architecture? 21

Chapter 6:
Using Both Redis and JWT 22
[1] References: 25

About Redis 26

Introduction

JSON Web Tokens are popularly used for managing user sessions. However, there are many in-depth articles and videos from subject matter experts (SMEs) of security companies like Okta talking about the potential dangers and inefficiencies of using JWT tokens[1]. Yet, these warnings are overshadowed by marketers, YouTubers, bloggers, course creators, and others who knowingly or unknowingly continue to promote them.

Introduction

Below are some examples of SMEs talking about the security problems of JWT:

"JSON Web Tokens can be used to validate user locally without the need for a database but then you put yourself at risk for massive security issues."

Source: "Why JWTs Are Bad for Authentication"—[Randall Degges](), Head of Developer Advocacy, Okta, a leading enterprise identity provider.

"To be clear: This article does *not* argue that you should *never* use JWT—just that it isn't suitable as a session mechanism, and that it is dangerous to use it like that. Valid usecases *do* exist for them, in other areas."

Source: "Stop using JWT for sessions" (see reference below for links).

"I don't care if you want to use stateless client tokens. They're fine. You should understand the operational limitations (they may keep you up late on a Friday scrambling to deploy a token blacklist), but, we're all adults here, and you can make your own decisions about that.

The issue with JWT in particular is that it doesn't bring anything to the table, but comes with a whole lot of terrifying complexity. Worse, you as a developer won't see that complexity: JWT looks like a simple token with a magic cryptographically protected bag-of-attributes interface. The problems are all behind the scenes."

Source: Thomas H. Ptacek, a well-known security researcher on Hacker News (see references below for links).

"Adopting them comes with drawbacks. You either forgo revocation, or you need to have infrastructure in place that is way more complex than simply adopting a session store and opaque tokens."

Source: "JWT should not be default for your sessions" (see reference below for links).

Introduction

One of the main reasons for JWTs usage is the need for speed. The other reason is that it's simple to use. The last reason is it's a buzzworthy and friendly name that's excellent for marketing. The name combines "JSON" (which is generally well liked), "Web" (for web), and "Token" (which implies stateless), and all of this may make people think it's perfect for their web authentication. In reality, it's not.

So this is a case in which the marketers have beaten out the engineers and security experts.

But it's not all bad, because there are regular long and passionate debates about JWT on Hacker News (see here, here and here), so there is hope. This amount of debate should give you pause because security should ideally be a black-and-white issue: either something is secure or it's not.

By the end of this book, you'll know the benefits and the dangers of JWTs, and also the battle-tested solution that thousands of companies use to overcome this.

Chapter 1

HTTP Sessions, Authentication, and Authorization

Before we get into JWTs, let's take a look at a use case to better understand sessions, authentication, and authorization.

The use case

Imagine that you are using Twitter. You log in to the platform, you "like" someone's tweet, and then you write a new tweet of your own. So you perform two additional actions after you log in. You need to be authenticated and authorized before you can perform each of the three specific actions. This is because HTTP is a stateless protocol, which means that the HTTP request doesn't store who you are from one request to the next.

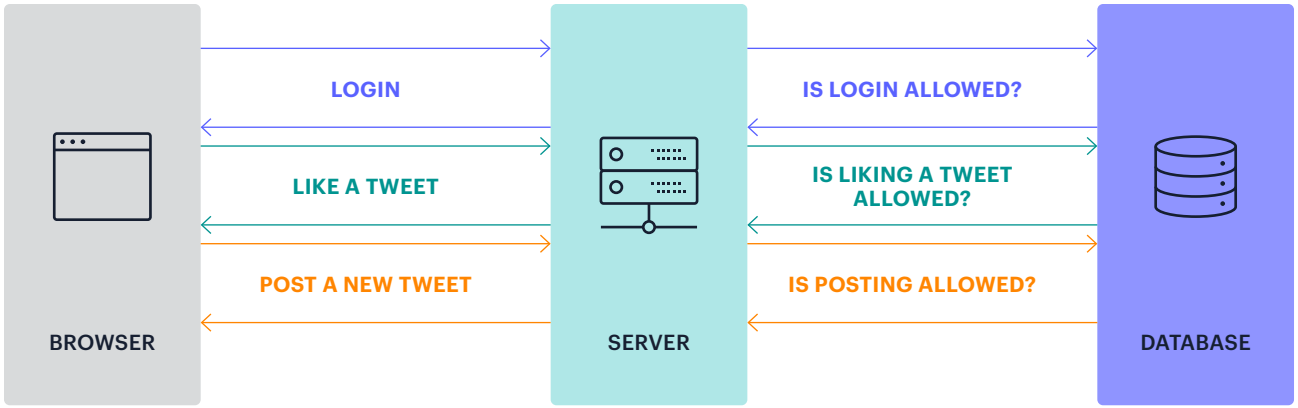


Figure 1: In the example at right, you are making three different requests and the server is verifying if your request is valid three different times.

Chapter 1

After you log in, the servers typically create a **session**. The session is a container that houses data about the user's interaction with the website or service, and, as its name implies, its lifetime is typically bound by the user's log in and log out actions. A session typically will have the following information:

- User's profile information, such as name, date of birth, email address, etc.
- User's permissions, such as "user," "admin," "supervisor," "super-admin," etc.
- Other app-related data, such as shopping cart details if it's a retail app, etc.
- Session expiration, such as one hour from now, one week from now, etc.

Managing sessions presents five major challenges:

1. Session data needs to be stored somewhere.
2. Since HTTP is stateless, session data must be sent back to the client so that the client can keep adding this information to future requests.
3. The client then needs to send the session data back to the server for future requests.
4. The server needs to verify if the client's information is valid; that is, "authentication" and "authorization." For example, whether or not the user who is liking a tweet is a "user" or "admin," if the session expired or not, etc.
5. Session expiration. At some point, the session needs to expire to force people to log in again for security reasons.

Let's look at each of these five points and get a general idea of how they work in most applications.

1. Where to store the session data (client vs. database)

If you send the session data back to the client (say browser or mobile app), then you risk security issues. Someone could access or intercept the session data, change that data, and access the server. In addition, there could be a huge amount of data that's going back and forth between the client and the server. So it needs to be stored in the backend—typically in a database.

Chapter 1

2. How to send session data to the client

If you create a session in the server (upon login) and then keep that data in the database, how can the client know about it? In order to solve this problem, servers generate a session token that looks like a random string that points to the actual session data in the database and sends it back to the client. The server either sends the session token in the form of a cookie or an HTTP response.

The session token is an opaque random string that looks something like this:
`fsaf12312dfsdf364351312srw12312312dasd1et3423r`

In the database, that string points to the entire session data. It will look something like this:

Figure 2: Example of how a session token maps to a session inside a database.

User's session table

SESSION TOKEN	SESSION
`fsaf12312dfsdf364351312srw12312312dasd1et3423r`	`{user_name: raja, email: raja@redis.com, isAdmin: true, shoppingCart:3, session_expiration:Aug-25th}`
`sadfsdfsd24323456456dfdfasda454`	`{user_name: Mike, email: Mike@redis.com, isAdmin: false, shoppingCart:1, session_expiration:Aug-22th}`

Chapter 1

3. How the client can send session tokens to the server for future requests

Once the client receives the session token in the form of a cookie or as a token, it keeps this information and adds this session token information to every future request.

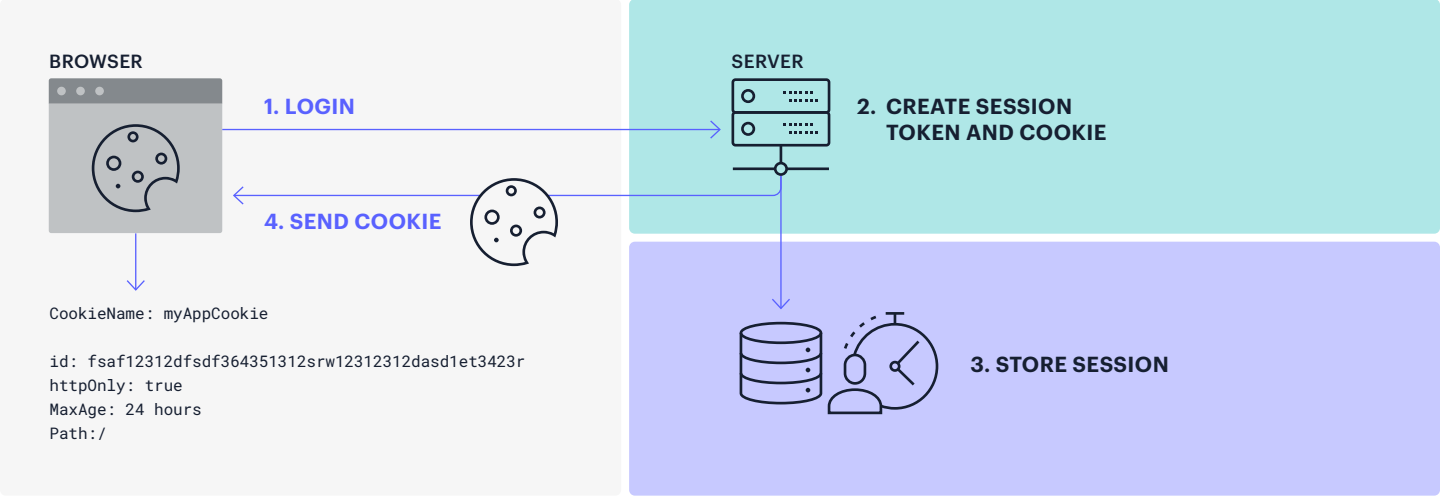


Figure 3: Example of how sessions and cookies are created and stored.

Here is how it works:

1. User logs in.
2. The server creates a session, session token, and cookie.
3. The server then stores the session and the session token into a database.
4. The server then sends the cookie that internally contains the session token back to the browser.

Chapter 1

4. How the server can handle authentication and authorization

Upon every future request, the server queries the database with the session token to get the actual session back, and then the server checks for two things:

1. **You are authenticated:** Your login data is still valid (verifies it is not tampered with, not expired, not logged out, etc.)
2. **You are authorized:** You can log in but do you have permission to do that specific action? (i.e., check if you are an admin, data owner, user, employee, super admin, etc.)

5. When will the session expire

Each session also has an expiration time, which can be set by the backend developer as anything from 5 minutes to 30 days. After that set time, the session data will be deleted. And if the user makes a call to perform some action, typically the user will be denied permission, and most client applications will redirect the user to the login page, forcing them to log in again. And when they log in, a new session is created with a new expiration time and the cycle starts over.

Note: If you authenticate using **OAuth**, you get multiple tokens such as "access token," "refresh token," and so on. These are all there to provide finer control of when the session should expire. For example, the client can use the refresh token to extend the session for additional time instead of logging people out.

Section summary

- HTTP is stateless, so to keep track of a user upon login, a "session" is created.
- A session is data about a user and their activity. It contains who they are, what they are authenticated and authorized to do, and also to keep track of any specific product-related data.
- Session data is typically stored in a database.
- A "session token" that points to the session is created and sent to the client for future references.
- The client sends this "session token" for every future request (via request header or through a cookie) to identify the user and other details that are stored in the session.
- The server retrieves the session from the session token by making an additional database call, checks if a valid session exists, and if the session token and the session itself are valid, it lets the user take future actions, such as liking a tweet, creating a tweet, etc.

Chapter 2

Storing Sessions in a Traditional Database

You just learned how sessions work. Now, let's continue with the Twitter example and see how the entire process works when you log in to Twitter and submit a tweet.

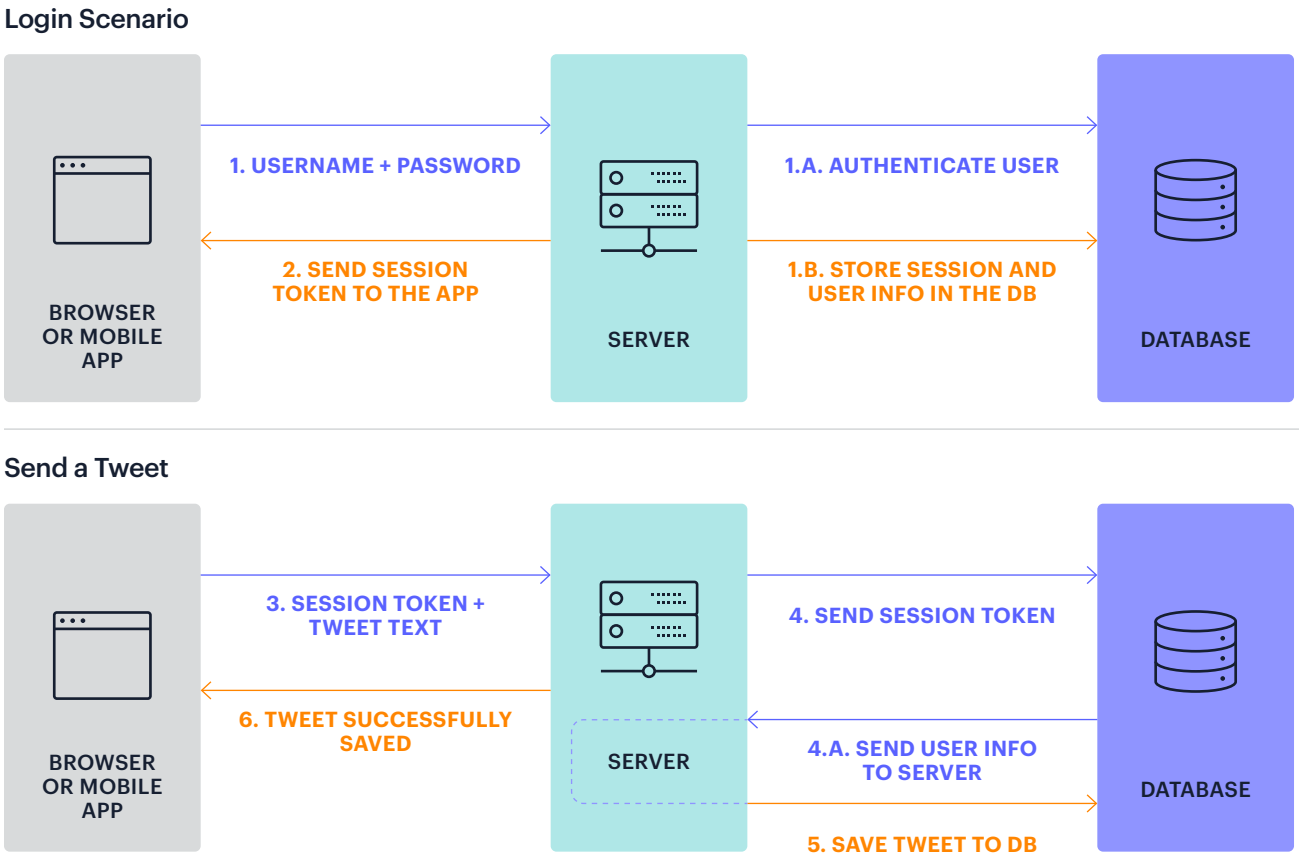
1. You log in with your username and password:
   a. The server first authenticates the user.
   b. The server then creates a session token and then stores that token along with the user's info in some database.

   > ***Note:*** *A session token is a long unidentifiable string—aka opaque string—that looks like this:*
   > `fsaf12312dfsdf364351312srw12312312dasd1et3423r`

2. The server then sends you a session token to the frontend mobile or web application.
   a. This token is then stored in the cookie or in the local storage of the app.
3. Next, say you write and submit a tweet. Along with your tweet, the app will then also send the session token (through a cookie or a header) so that the server can identify who you are. (But remember that the token is just a random string, so how can the server know who you are just from the session token?)
4. When the server receives the session token, it won't know who the user is, so it sends that to the database to retrieve the actual user's info (such as the userID) from that token.
5. If the user exists and is allowed to complete that action (i.e., send a tweet), the server allows them to do the action.
6. Finally, it tells the frontend that the tweet was sent.

Chapter 2

Figure 4: Illustrating the end-to-end flow of using a regular database as a session store.



Chapter 2

The main problem with this approach

The main problem with this approach is that step four is slow and needs to be repeated for every single action the user does. So every API call leads to at least two slow DB calls, which can slow down the overall response time.

There are two ways to solve this problem:

1. Somehow eliminate database lookup for users completely (i.e., eliminate step four).
2. Make the extra database lookup much faster so that the additional hop won't matter.

Option 1: Eliminate database lookup (step four)

There are three different ways of achieving option 1:

1. **Store the state in the server's memory.** However, this can cause issues when you scale since this state is only available on a specific server.
2. **Use "sticky sessions."** This is when you instruct the load balancer to always direct the traffic to a specific server even after you scale up. Again, this can cause different scaling issues and if the server goes down (scale down), you'll lose all the sessions.
3. **Use JSON Web Tokens.** We'll investigate how to do this in the following chapter.

Option 2: Make the database lookup so fast that the additional call won't matter

Simply use Redis. Tens of thousands of companies use Redis for session storage. With sub-milliseconds latency, it's as if you are storing this data in the server itself. We'll look into this more later.

In the next chapter we'll learn about how JWT works, including its perceived benefits as well as potential risks.

Chapter 3

Storing Sessions in JWT

JWT, especially when used as a session, attempts to solve the problem of time-consuming repeated database calls by completely eliminating the database lookup altogether.

The main idea is to store the user's info in the session token itself. This means, instead of some long, random string, the actual user info is passed in the session token itself. And to secure it, part of the token is signed using a secret that's only known to the server. So even though the client and the server can see the user info part of the token, the second part, the signed part, can only be verified by the server. In the example below, the pink section of the token contains the payload (user's info) and can be seen by both the client and the server.

But the blue part is signed using a secret string, the header, and the payload itself. And so if the client tampers with the payload (say impersonates a different user), the signature will be different and won't be authenticated.

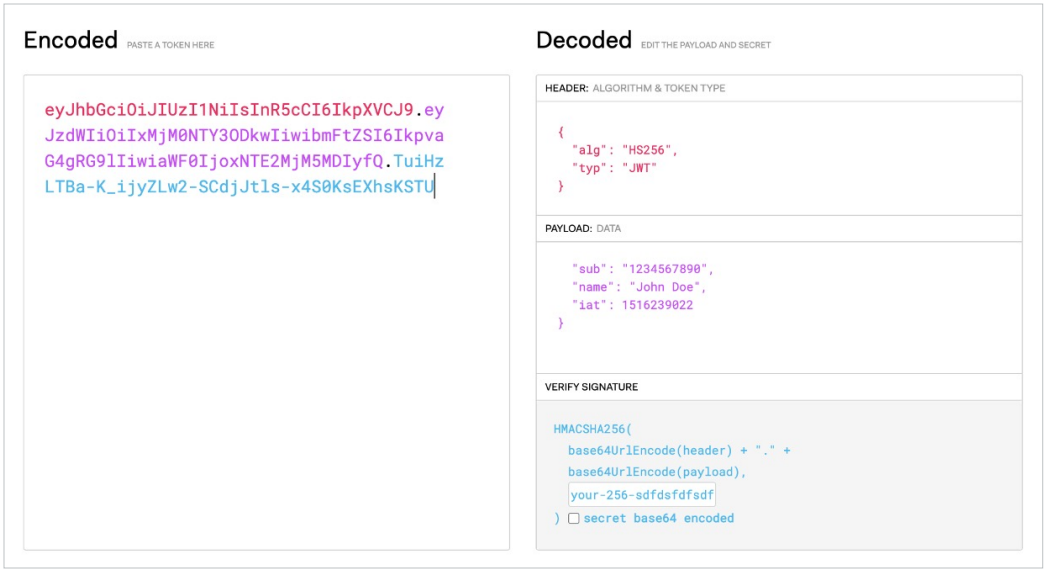


Figure 5: Example of the different sections of the JWT token.

Chapter 3

The previous image on page 14 shows a JWT token. It includes <header>.<payload>.<signature>. The header (highlighted in red) and the payload (highlighted in purple) are often not encrypted (and just base64 encoded), but the signature (highlighted in blue) is signed.

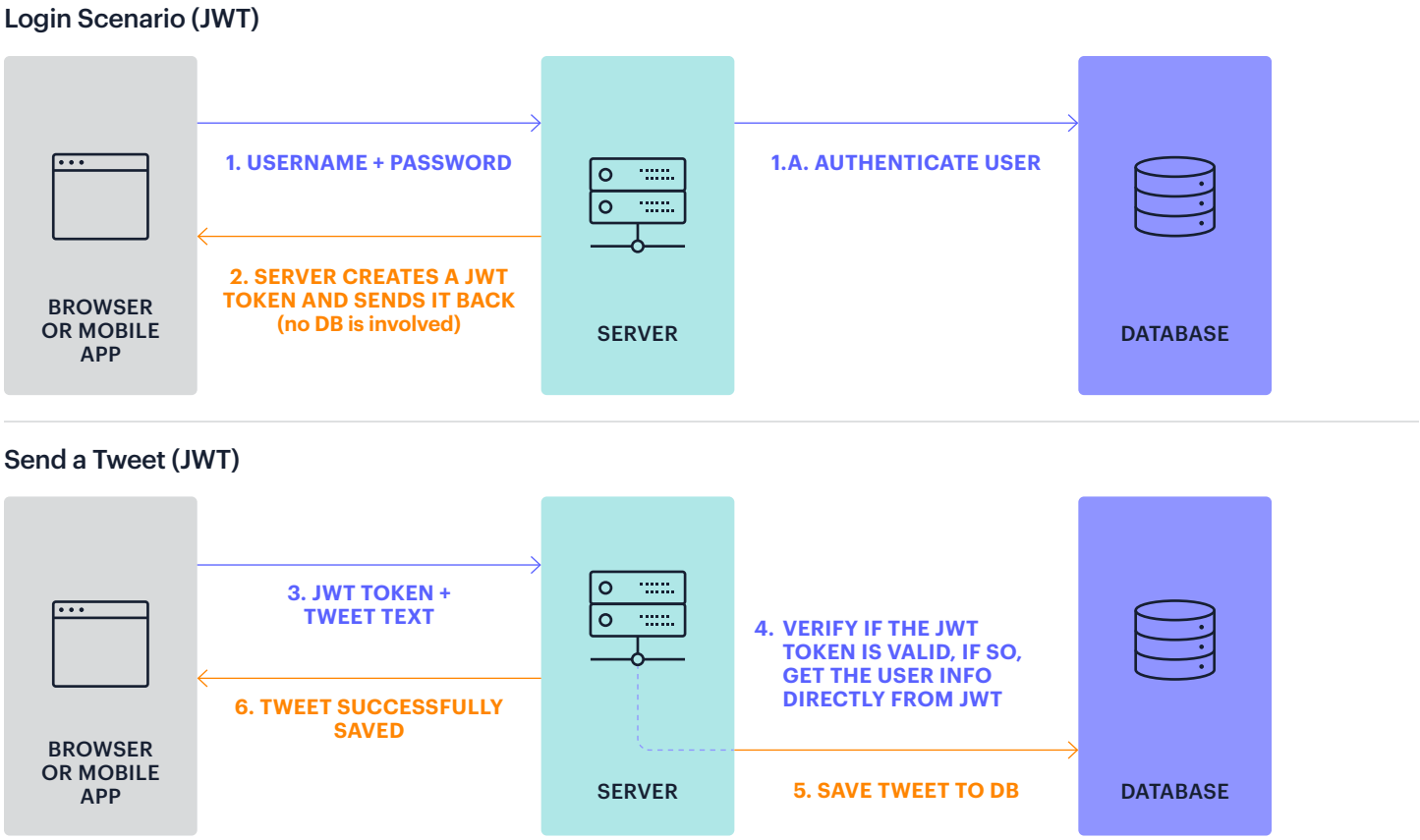
Here is how our use case would look with JWT:

1. You log in with your username and password:
   a. The server authenticates the user by querying the database.
   b. The server then creates a JWT session token using the user's info and the secret (no DB is involved).
2. The server then sends you a JWT token to the frontend application. For future activities, the user can just send the JWT token to identify the user instead of logging in every time.
3. Next, say you again write and submit a tweet. When you send it, along with your tweet's text, your app will also send the JWT token (through a cookie or a header) so that the server can identify who you are. But how can the server know who you are just from the JWT token? Well, part of the token already has the user information.
4. So when the server receives the JWT token, it uses the secret string to validate the signed section and gets the user info from the payload section, thus eliminating the DB call.
5. If the signature is verified, it allows them to do the action.
6. Finally sends the frontend that the tweet was saved (i.e., the result of the action the user was originally intended to take)

Going forward for every user action, the server simply verifies the signed section, gets the user info, and lets the user complete that action, effectively skipping the DB call completely.

Chapter 3

Figure 6: Example of how JWT helps eliminate DB lookup for session store.



Chapter 3

Token expiration

But there is one additional and important thing to know about the JWT tokens—it uses an expiration time to expire itself, which is typically set from 5 to 30 minutes. And because it's self-contained, you can't easily revoke/invalidate/update this expiration time. This is really where the crux of the problem lies, but let's look at some other specification flaws.

JWT is too "liberal" for a security spec and therefore is vulnerable

The JWT specification is written more like the HTML specification. In HTML, if you don't close an HTML tag (say a), it's still valid and the browsers continue to display it. The goal is to try to make the "best effort" to render something in the browser instead of throwing an error, so it's very "liberal" in that sense. But this causes a lot of problems for browser developers and frontend engineers, although thankfully the worst that can happen is just a bad webpage rendering.

The JWT spec is similarly written. Instead of being very strict in enforcing security rules and having built-in best practices, it provides a lot of workarounds to allow for edge cases and a variety of use cases. These workarounds in turn can lead to security breaches.

And just like with HTML, it burdens the backend engineers and library creators, who must know all the best practices to avoid these loopholes. However, unlike with HTML, if people don't follow those best practices, it could cause a lot more damage than a bad rendering, such as allowing weak authentication.

The other problem is because the spec itself is so liberal, JWT library creators don't have a choice but to comply. For that reason, backend engineers need to be very careful when using JWT because what might be technically valid according to the spec might not be secure.

On the next page are some examples of these potential security issues:

Chapter 3

1. The “none” algorithm

JWT allows various algorithms to sign the payload. One of them is the “none” algorithm. At a high level, if someone specifies algorithm “none,” it means that the JWT libraries should ignore validating the signature completely. So all the attacker needs to do is simply change the algorithm type to “none” and send whatever they want to the server. The libraries will think that there is no need to validate the signature and will give access.

For example, say the attacker passes “alg” = none in the header. And say you are reading the alg from the request header shown below (req.header.alg)—which is valid—you could then hit this vulnerability.

Client:

```
{
  alg: 'none' // algorithm none, HS256, RS256, etc etc
}
```

Server:

```
//read the algorithm from the request header
const token = jwt.sign(payload, secret, {algorithm: req.header.alg})
```

The solution to this problem is that the backend engineers need to ensure they ignore the “none” algorithm. Even Auth0, which promotes JWT, got hit with a big security issue (read more about this here: Critical vulnerabilities in JSON Web Token libraries [Auth0.com]).

Chapter 3

2. The algorithms are passed in an array

JWT allows specifying algorithms in an array during verification. This opens up another type of attack. It's complicated to explain, but the gist is that when you pass algorithms in an array, the library starts to check if one of these algorithms works for the payload. Apparently, you can exploit it because you can use the RS256 key as HS256 secret. So if you have the following code, an attacker could use the RS256 token as an HS256 secret and bypass the JWT security test. **You can learn more about this here** (also listed in references #9).

```
const payload = await jwt.verify(token, secret, {algorithms: ['HS256', 'RS256']})
```

The solution is to use just one algorithm, or use two different methods with just one algorithm each, and call two methods independently.

3. Claims are optional

JWT provides a nice way to organize and ensure different claims that could help improve security, but they are all optional. For example, in the sample code to the right, sub, iss, aud, and so on are all optional. This puts the burden on implementers to follow best practices. If the spec had made some of them mandatory, it would have solved a lot of security headaches.

For example, if the "aud" was mandatory, it would force engineers to think about it and ensure that the JWT for one service (e.g., "CartService") doesn't work for another service (e.g., "BackOfficeService"). Because these are optional they are often not required or configured correctly. To fix this, claims must be set and checked against what is expected.

```
{
  sub: '12345' //subject
  iss: 'MyService' //issuer
  aud: 'PaymentAPI' //audience
  exp: '1234565456', // expiration
  iat: '1232312312', //issue time
  name: 'Raja', //custom payload
  admin: 'true' //custom payload
}
```

Chapter 3

Other considerations and issues

1. **Length of tokens.** In many complex real-world apps, you may need to store a great deal of information, and storing it in the JWT tokens could exceed the allowed URL length or cookie lengths, leading to problems. Also, you are now potentially sending a large volume of data on every request.

2. **The state needs to be maintained anyway (for rate-limiting, IP-whitelisting, etc.).** In many real-world apps, servers have to maintain the user's IP and track APIs for rate-limiting and IP-whitelisting. So you'll need to use a blazing-fast database anyway. To think your app somehow becomes stateless with JWT is just not realistic.

So why is JWT dangerous for user authentication?

In addition to all the aforementioned issues, the biggest problem with JWT is the token revocation problem. Since it continues to work until it expires, the server has no easy way to revoke it.

Following are some use cases that could make this dangerous:

1. **Logout doesn't really log you out.** Imagine you logged out from Twitter after sending your tweet. You'd think that you are logged out of the server, but that's not the case because JWT is self-contained and will continue to work until it expires. This could be 5 minutes or 30 minutes or whatever the duration that's set as part of the token. So if someone gets access to that token during that time, they can continue to use it to authenticate until it expires.
2. **Blocking users doesn't immediately block them.** Imagine you are a moderator of Twitter or some online real-time game where real users are using the system. And as a moderator, you want to quickly block someone from abusing the system. You can't, again for the same reason. Even after you block them, the user will continue to have access to the server until the token expires.
3. **JWTs could contain stale data.** Imagine the user is an admin and got demoted to a regular user with fewer permissions. Again, this won't take effect immediately and the user will continue to be an admin until the token expires.
4. **JWT's are often not encrypted.** Because of this, anyone able to perform a man-in-the-middle attack and sniff the JWT now has your authentication credentials. This is made easier because the MITM attack only needs to be completed on the connection between the server and the client.

There are ways to encrypt JWT tokens called **JWE**, but when you use this method, the clients (especially browsers and mobile devices) won't have a way to decrypt them to see the actual payload. At this point, you are essentially using the JWT as a regular encrypted session token—at least from the perspective of the web apps and mobile apps.

Chapter 3

Still trying to make JWT work?

Assuming you got past all the spec's issues and just want to solve the expiration issue. One popular solution is to store a list of "revoked tokens" in a database and check it for every call. And if the token is part of that revoked list, then block the user from taking the next action. But now you are making that extra call to the DB to check if the token is revoked, and so this negates the entire purpose of using JWT altogether.

The diagram below does a good job of articulating the challenges of using JWT and the issues with all the workarounds. There are five different ways you can try to make JWT better, but in all five scenarios, you'll hit one bottleneck or another. You should ask yourself, why do you need to use a type of technology that needs so much workaround? And by the time you implement all of the workarounds to your satisfaction, you'd lose the benefits of it to begin with.

A handy dandy (and slightly sarcastic) flow chart about why "your solution" doesn't work.

I think I can make JWT work for sessions by... →

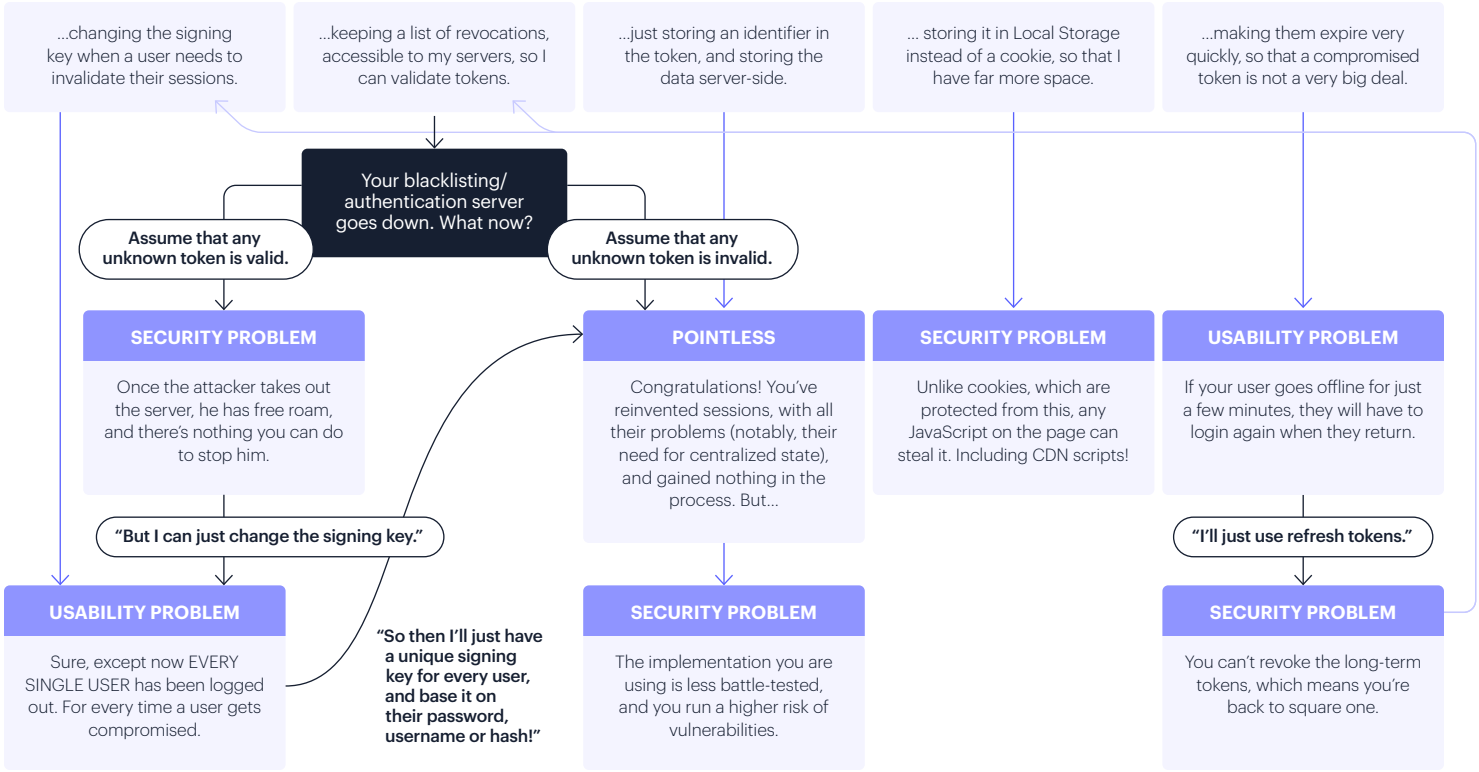


Figure 7. Source: [Stop using JWT for sessions, part 2: Why your solution doesn't work]

Chapter 3

Bottom line

Although JWT does eliminate the database lookup, it introduces security issues and other complexities while doing so, therefore making it risky to use for user sessions.

When can I use JWT?

There are scenarios when it might make sense to use JWT, such as when you are doing server-to-server (or microservice-to-microservice) communication in the backend and one service could generate a JWT token to send it to a different service for authorization purposes. Or other specific scenarios, such as a reset password, for which you can send a JWT token as a one-time, short-lived token to verify the user's email.

If I can't use JWT, what else can I do?

The solution is to not use JWT at all for session purposes. But instead, use the traditional but battle-tested way to more efficiently make the database lookup so blazing fast (sub-millisecond) that the additional call won't matter.

Chapter 4

Storing Sessions in Redis

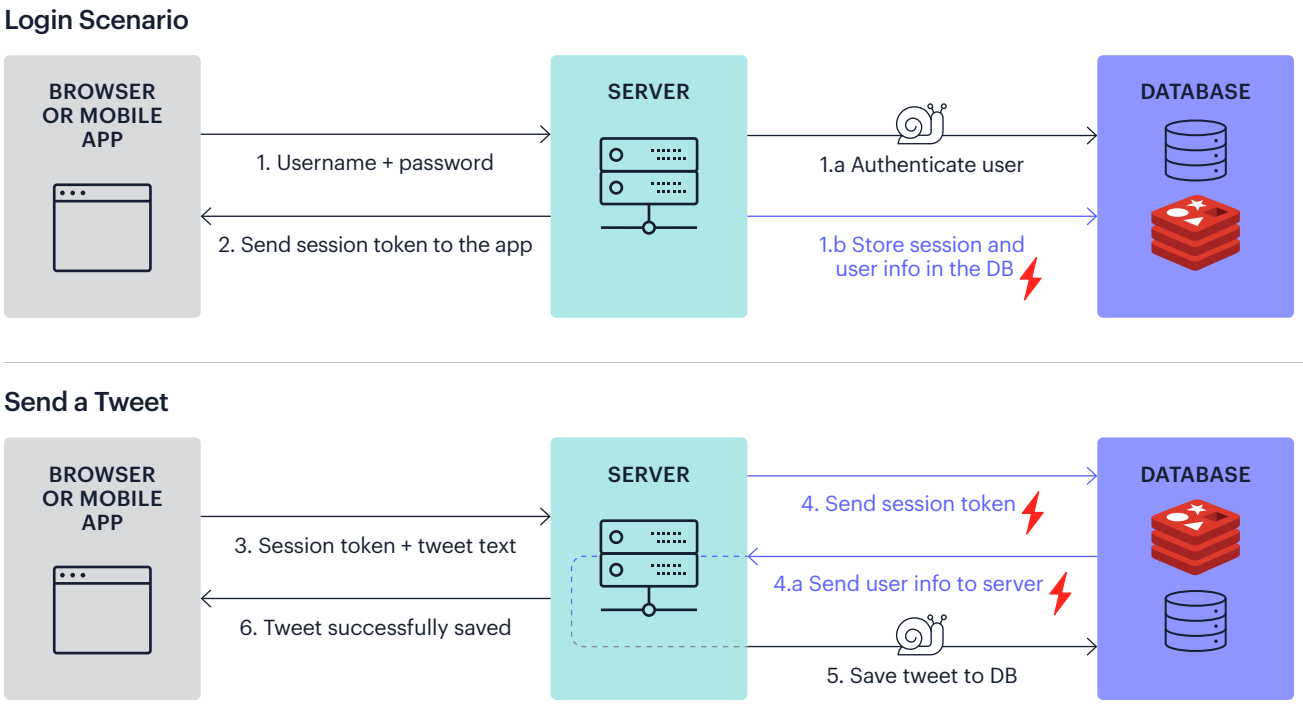
If the answer to the problem we've outlined so far is to use the tried-and-true method (i.e., store the sessions in a database), but to make that database lookup so fast that the additional call won't matter, how exactly can this be achieved?

What you need is a database that can serve millions of requests in sub-milliseconds. Thousands of companies, serving billions of users, use Redis for this exact purpose. With Redis, the additional database call is so fast that it no longer presents a problem.

Redis as a Session store (plus rate-limiter, etc.) →

Figure 8: Example of the complete step-by-step flow of using Redis as a session store.

Note that the lightning icon indicates a blazing-fast speed, and the snail icon indicates slow speed.



Chapter 4

Here is how storing a session in Redis works

1. You log in with your username and password:
   The server first authenticates the user.
   - The server then creates a session token and stores that token along with the user's info in Redis. It'll look something like this:
     i. **SET sess:12345 "{user:raja, shopping:3, DOB: 1/1/21}"**
     ii. Where
       - 12345 is the session token ID.
       - "session:12345" is the Redis key.
       - "{user: raja, shopping:3, DOB: 1/1/21}" is the value.
2. The server then sends you a session token to the frontend mobile or web application.
   - This token is then stored in the cookie or in the local storage of the app.
3. Next, say you write and submit a tweet as in our previous example. Along with your tweet, your app will also send the session token (through a cookie or a header) so that the server can identify who you are. But, as before, the token is just a random string, so how can the server use it to identify you?
4. When the server receives the session token, it won't know who the user is, so it sends that to Redis to retrieve the actual user's info (like userID) from that token.
   - **GET session:12345**
   - "{user:raja, shopping:3, DOB: 1/1/21}" //Response from Redis
5. If the user exists and is allowed to complete the action (i.e., send a tweet), the server allows them to do the action.
6. And finally, it tells the frontend that the tweet was sent (with the original response to the original request).

And because Redis is so fast, you don't need to send large user data back and forth between the client and server—just a session token is enough. You can get the actual payload from Redis in micro-seconds or sub-milliseconds. Since you are not sending the data to the client, it's harder for people to steal. And since the actual data is inside Redis, you don't have to depend on session expiration time, you can just verify it against the Redis database itself. And finally, you can delete the user data from Redis when they log out, so you can always be sure of authentication and authorization.

As you can see, it's an age-old approach of storing the data in a database, but by making it blazing fast, you effectively eliminate the speed issue. And it has no security vulnerabilities like JWT.

Since people have been using Redis for session storage for ages, there are plenty of examples of how to implement this in virtually all languages and frameworks. But here is a simple overview of how to store data in sessions in NodeJS. It works the same way in every language.

Chapter 4

An example code snippet (Node.js)

1. Import Redis module, create a redisClient, and then create a session.

```
const redis = require('redis')
const session = require('express-session')

let RedisStore = require('connect-redis')(session)
let redisClient = redis.createClient()

app.use(
  session({
    store: new RedisStore({ client: redisClient }),
    saveUninitialized: false,
    secret: 'keyboard cat',
    resave: false,
  })
)
```

Source: https://github.com/tj/connect-redis

Chapter 4

2. Store whatever user information you want into the session and it will be stored in Redis. The example below shows page view count (req.session.views++) being stored as part of the session data. So every time this page is visited, the count will increase.

Similarly, you can add req.session.userName, req.session.shoppingCartCount, and set additional properties and associated value to each property and store them all within the session. For example, you may add req.session.productName = "JSBook", req.sesion.quantity = "3", rea.session.totalPrice= "$30", etc.

```
// Access the session as req.session
app.get('/', function(req, res, next) {
  if (req.session.views) {
    req.session.views++
    res.setHeader('Content-Type', 'text/html')
    res.write('<p>views: ' + req.session.views + '</p>')
    res.write('<p>expires in: ' + (req.session.cookie.maxAge / 1000) + 's</p>')
    res.end()
  } else {
    req.session.views = 1
    res.end('welcome to the session demo. refresh!')
  }
})
```

Source: https://github.com/expressjs/session#reqsession

Chapter 5

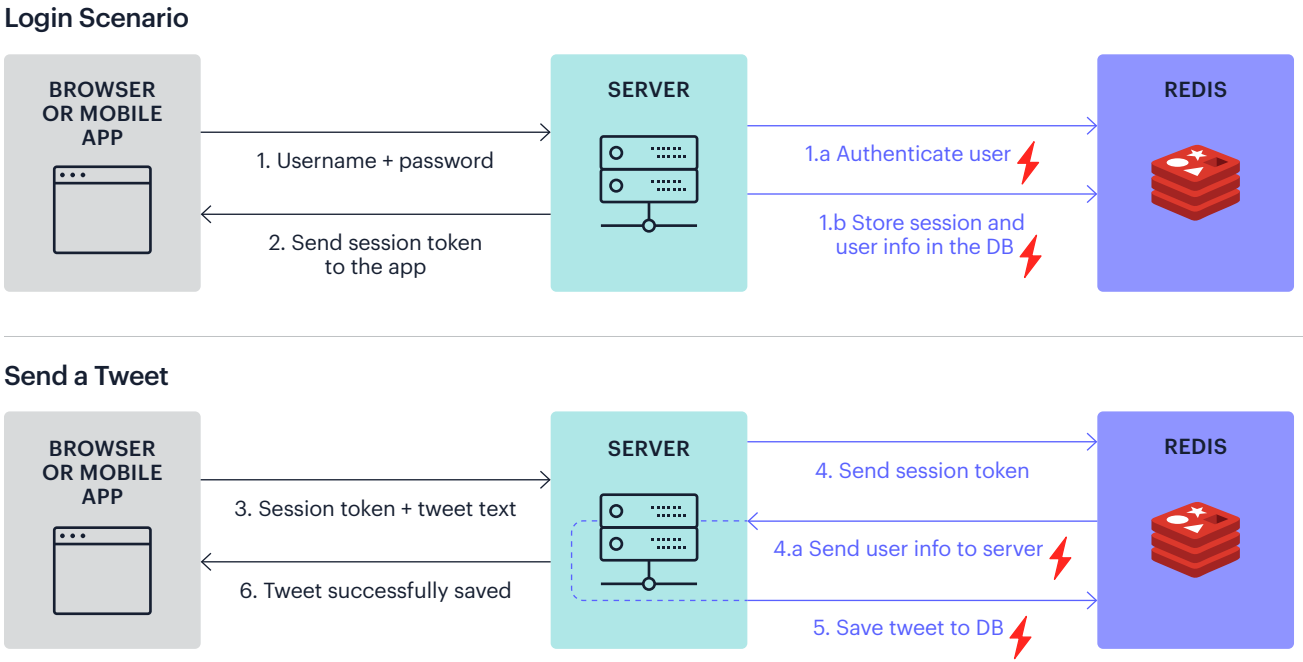
Sessions When Redis Is Your Primary Database

Redis has been the leading database for caching and session storage for more than a decade now. But over the last few years, Redis has evolved into a primary multi-model database that offers seven officially supported modules. For example, you can use RedisJSON (10X faster vs. the market leader) and essentially have a real-time MongoDB-like database, or use the RediSearch module (4X to 100X faster) and implement real-time full-text search like Algolia.

With Redis as your primary database, everything becomes blazing fast—not just your session storage. In this architecture, all the application's primary data and the sessions data—as well as everything else—lives side-by-side in the same database.

Redis as a Primary DB →

Figure 9: How using Redis as a primary database simplifies the overall architecture.



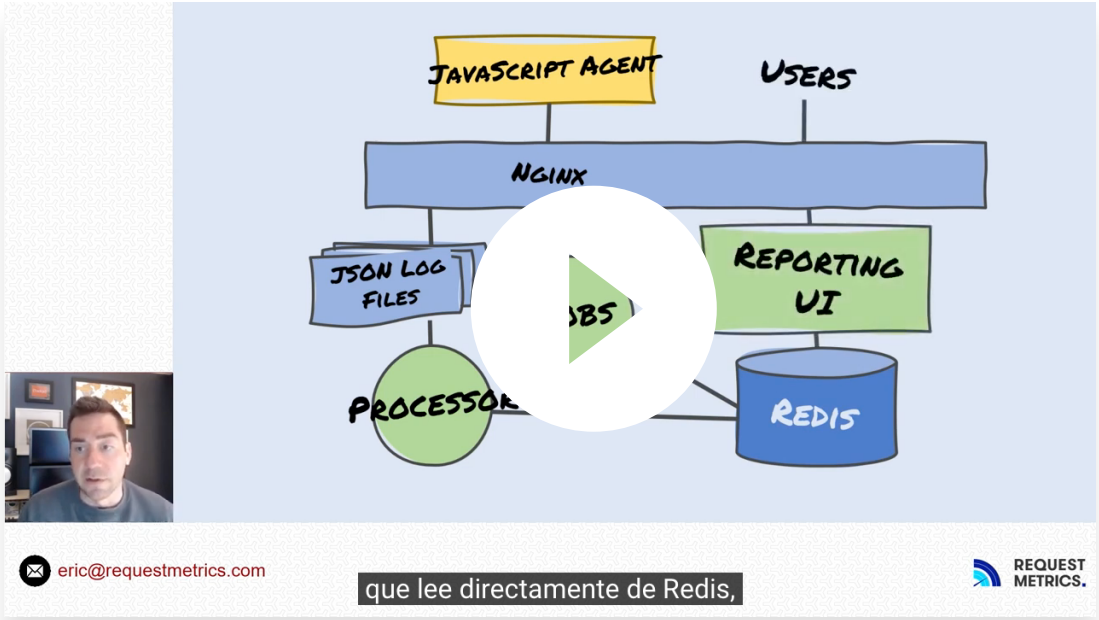
Chapter 5

1. You log in with your username and password:
   - The server first authenticates the user by getting the user information from Redis (instead of a slow database).
     i. **HGET user:01**
     ii. > username: raja password wer8wrw9werw8wrw //Response
       - Imagine the username and (encrypted) password are stored in "user:01" key as a Hashmap.
   - The server then creates a session token and stores that token along with the user's info into Redis. It will look something like this:
     i. **SET sess:12345 "{user:raja, shopping:3, DOB: 1/1/21}"**
     ii. Where
       - 12345 is the session token ID.
       - "session:12345" is the Redis Key.
       - "{user: raja, shopping: 3, DOB: 1/1/21}" is the value.
2. The server then sends you a session token to the frontend mobile or web application.
   - This token is then stored in the cookie or in the local storage of the app.
3. Next, say you wrote and submitted a tweet. Then along with your tweet, your app will also send the session token (through a cookie or a header) so that the server can identify who you are. But the token is just a random string, so how can the server know who you are just from the session token?
4. When the server receives the session token, it won't know who the user is, so it sends that to Redis to retrieve (4a) the actual user's info (like userID) from that token.
   - ***GET session:12345***
   - *> "{user:raja, shopping:3, DOB: 1/1/21}" //Response from Redis*
5. If the user exists and is allowed to do that action (i.e., send a tweet), the server allows them to complete the action.
6. And finally, it tells the frontend that the tweet was sent.

Chapter 5

Is anyone using this architecture?

Redis works with thousands of customers on a daily basis, and although Redis is still primarily used as a secondary database, we have started to see this new DBless architecture emerge over the last couple of years. It started to get more momentum as Redis itself became more feature-rich, powerful, and as more people found success. Companies like Request Metrics have built their entire startup on this architecture and find it incredibly successful. [Watch this video](#) in which Eric Brandes from Request Metrics explains how his company did it.



Learn more about how you can use Redis as a primary database by watching these videos:

- [Redis in 100 seconds](#) [2021, 200,000+ views]
- [Redis as a Primary DB (Ofer Bengal, CEO, Redis)](#)
- [Redis as a Primary DB(Yiftach Shoolman, CTO, Redis)](#)
- [Goodbye cache, Redis as a primary DB](#)
- [What is DBLess architecture?](#) [2021]
- [Can Redis be used as a Primary DB?](#) [2021]

Chapter 6

Using Both Redis and JWT

In this final chapter we will review another widely used option that provides some of the benefits of JWT but removes most of the security issues previously discussed (with the exception of man-in-the-middle attacks). It is possible to use JWT as a preliminary check while using Redis as the secondary check. In such a scenario, if the JWT verification succeeds, the server will still go to Redis and double-check the information there. However, if the JWT verification itself fails, there's no need to worry about checking the Redis database.

Another benefit of this approach is that you get to use existing JWT libraries on both the frontend and backend without having to develop your own custom way of storing the data in Redis (although it's not a big deal).

One last thing to note here is that, as mentioned above, this setup will still make the app vulnerable to potential man-in-the-middle attacks between the clients and server because the tokens are not encrypted.

As mentioned earlier, there is a way to encrypt **JWT** tokens called JWE, but when you use this method, the clients (especially browsers and mobile devices) won't have a way to decrypt them to see the actual payload. At this point, you are essentially using the JWT as a regular encrypted session token, at least from the perspective of the web apps and mobile apps.

If you are using it for machine-to-machine communication, such as with microservices when you want to share login info between two different services, you can then share public keys to decrypt and see the JWT data—but that's a different use case.

Chapter 6

The best approach is to simply use encrypted sessions and store them in Redis. If you want to go this route, there are plenty of libraries that support this.

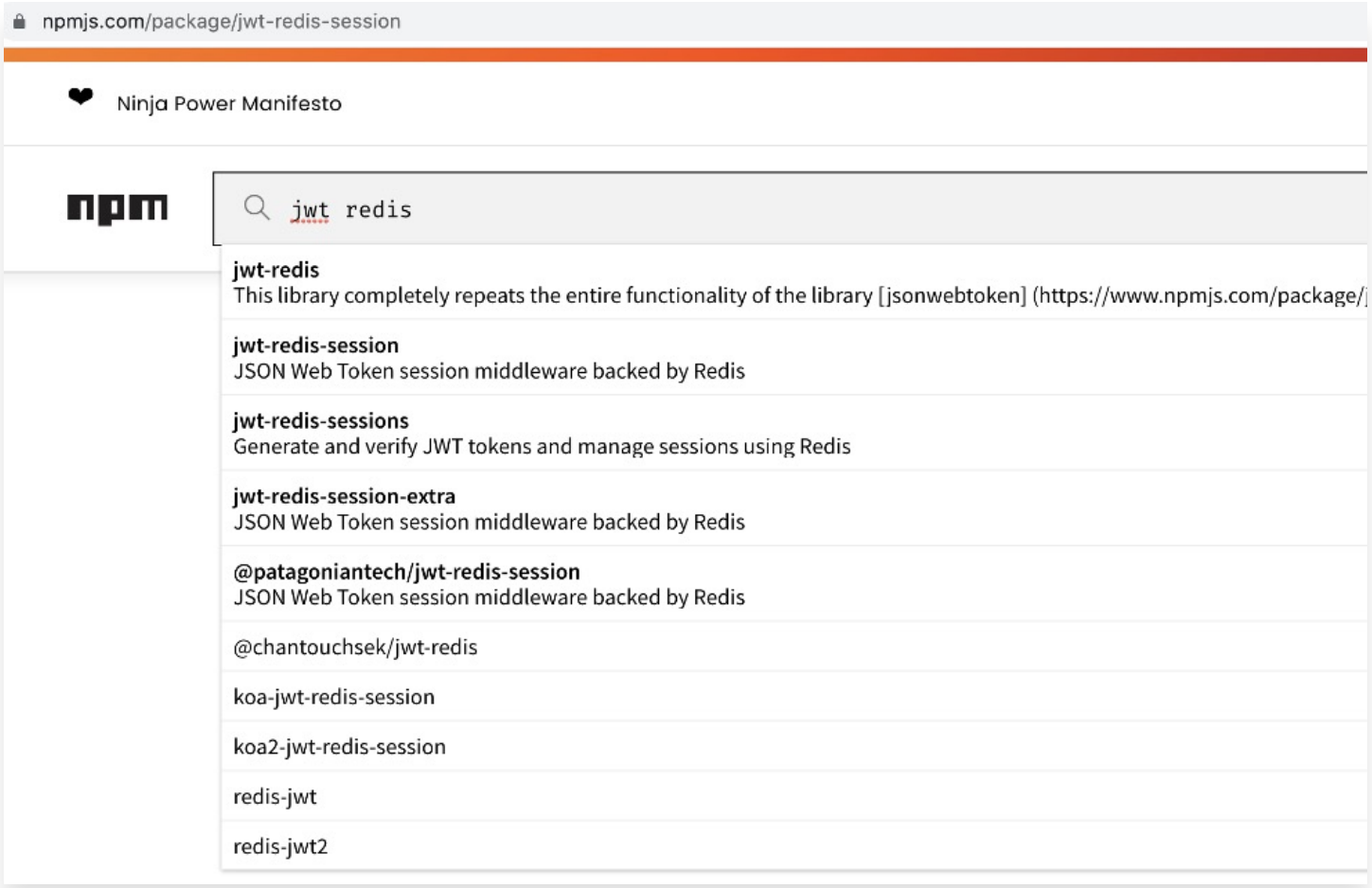


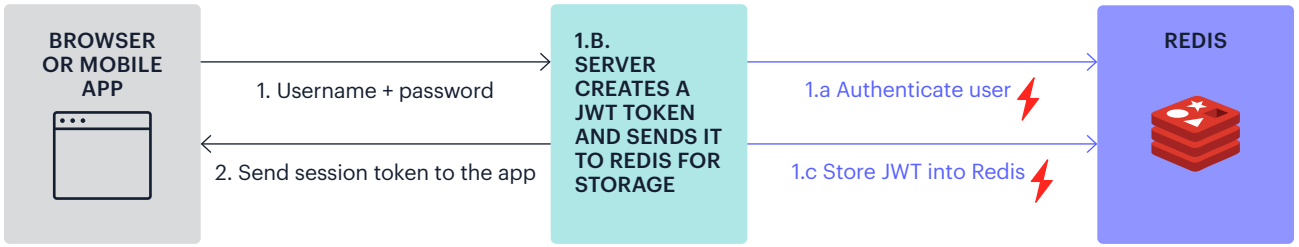
Figure 10: Example of JWT + Redis nodejs libraries from npmjs.com.

Chapter 6

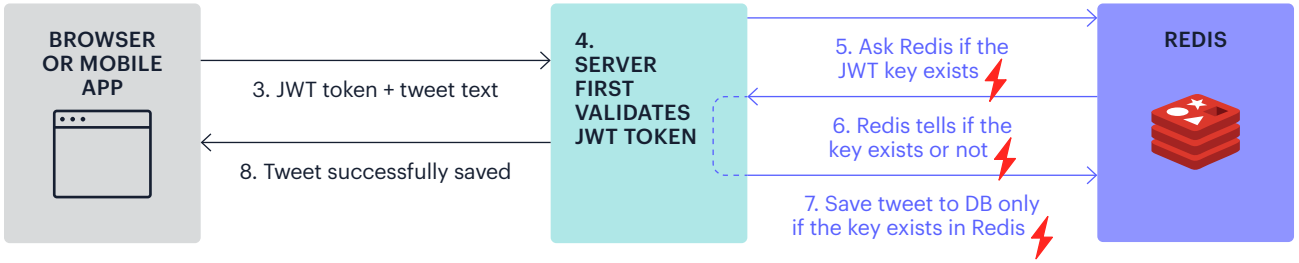
In the following example, we will use Redis as the primary DB, but the process is the same even if you have an additional DB.

JWT + Redis →

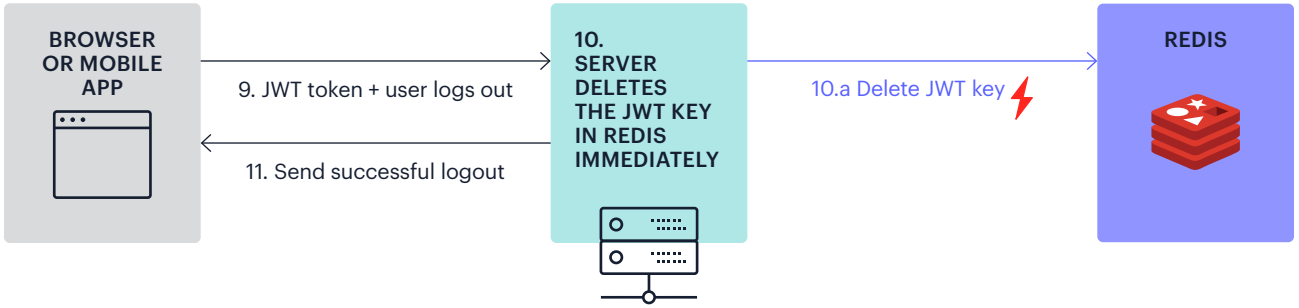
Login Scenario



Send a Tweet



User Logs Out



Chapter 6

There are slightly different implementations of this depending on the library, but in general this is how the process works:

1. User logs in using username and password.
   a. The server checks if they are valid.
   b. The server creates a JWT token (instead of just regular session tokens).
   c. The server sends the JWT to Redis for storage.
2. The server sends the JWT token back to the client.

The user sends a tweet:

3. The user sends a tweet (along with the JWT token).
4. The server first validates the JWT token.
5. If the JWT token is valid, it then asks if the JWT token exists in Redis.
6. Redis tells the server if the token still exists (we don't need to do any more validations; just checking if the key exists is sufficient because JWT already does the job).
7. If the key exists, the server saves the tweet to the DB.
8. The server sends a response back to the client saying the tweet was saved.

The user logs out:

9. The user then logs out.
10. The server immediately deletes the token in Redis. So going forward, step 7 fails.
11. The server sends a successful logout response to the client.

Tip:

In order to store the JWT token in Redis and also expire it automatically, use the following format.

```
SET <key> <value> EX <expiration time in seconds>
```

In the example below, we are storing the entire JWT token as the key (you can also store just parts of it) and storing its value as "1" to have some value (this could be anything), and setting the expiration to one week.

```
SET sess:<entire JWT key> 1 EX 604,800
```

You can learn more SET options here:
https://redis.io/commands/set

---

[1] References:

1. Stop using JWT for sessions
2. JWT should not be your default for sessions
3. Why JWTs Are Bad for Authentication - Randall Degges (Head of Dev Relations, Okta)
4. Stop using JWT for sessions, part 2: Why your solution doesn't work
5. Thomas H. Ptacek on Hacker News
6. My experience with JSON Web Token
7. Authentication on the Web (Sessions, Cookies, JWT, localStorage, and more)
8. Thomas H. Ptacek's blog
9. What makes JSON Web Tokens (JWT) secure?
10. Critical vulnerabilities in JSON Web Token libraries [Auth0.com]

About Redis

Data is the lifeline of every business, and Redis helps organizations reimagine how fast they can process, analyze, make predictions, and take action on the data they generate. Redis provides a competitive edge to any business by delivering **open source** and **enterprise-grade** data platforms to power applications that drive real-time experiences at any scale. Developers rely on Redis to build performance, scalability, reliability, and security into their applications.

Born in the cloud-native era, Redis uniquely enables users to unify data across multi-cloud, hybrid and global applications to maximize business potential. Learn more about Redis at **redis.com** and sign up for **your free trial**.