# JSON Web Tokens (JWTs) Are Not Safe

This webinar discusses why security experts believe JWTs aren’t safe for user sessions — and a battle-tested alternative. Based on the widely popular e-book.

Will Johnston

## Introduction

- Will Johnston, Developer Growth Lead at Redis
- Over 15 years of software development experience
- Worked in startups all the way to Fortune 50
- You may recognize me: youtube.com/c/Redisinc
- redis.com/try-free?coupon=JWT3

## What we will cover

In this webinar we will explore:

- HTTP Sessions, Authentication, and Authorization
- Why you might decide to use JWTs
- The perceived benefits and actual dangers of using JWTs
- JWT workarounds and their complexities
- How to use Redis for session storage
- Session storage using Redis as a primary database

**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.**

**Yet, these warnings are overshadowed by marketers, YouTubers, bloggers, course creators, and others who knowingly or unknowingly continue to promote them.**

"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 JWTs"

"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.

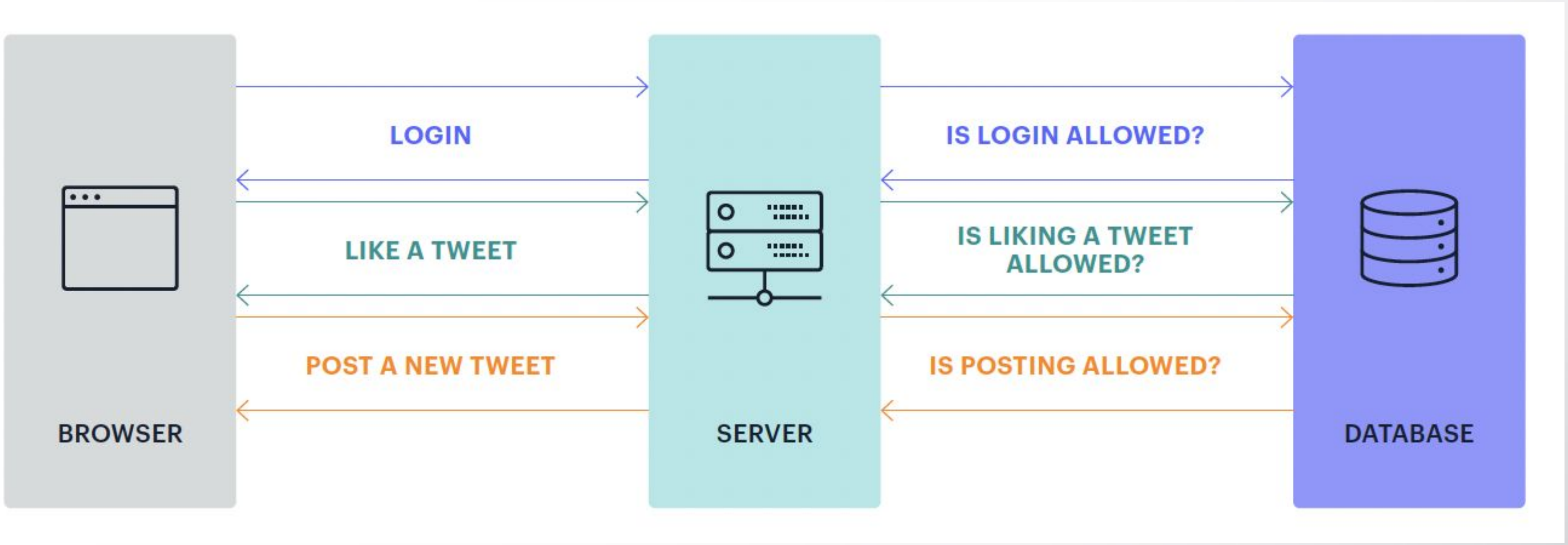
The issue with JWT in particular is that it doesn't bring anything to the table, but comes with a whole lot of terrifying complexity. Worse, you as a developer won't see that complexity: JWT looks like a simple token with a magic cryptographically protected bag-of-attributes interface. The problems are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

Source: Thomas H. Ptacek, a well-known security researcher on Hacker News.

redis

## HTTP Sessions, Authentication, and Authorization

**Use Case**: Twitter



## Challenges with Sessions

Five major challenges

1. Session data needs to be stored somewhere.
2. Session data must be sent back to the client so the client can add 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.

## 1. Where to store the session data

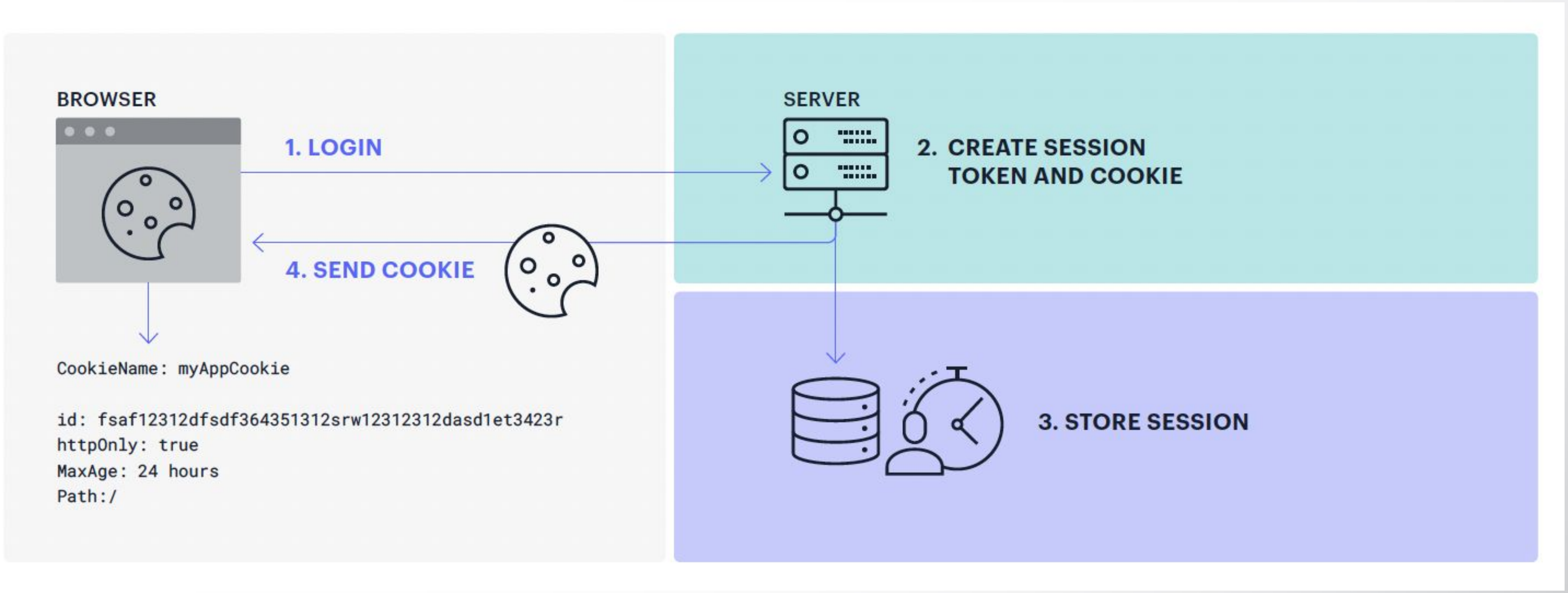
- On the client
- In server memory
- In the database

## 2. How to send session data to the client

| 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} |

redis

## 3. How the client sends session tokens to the server



```
CookieName: myAppCookie

id: fsaf12312dfsdf364351312srw12312312dasd1et3423r
httpOnly: true
MaxAge: 24 hours
Path:/
```

## 4. How the server handles authentication and authorization

1. **You are authenticated**: Your login data is still valid
2. **You are authorized**: You can login, but do you have permission to perform a specific action?

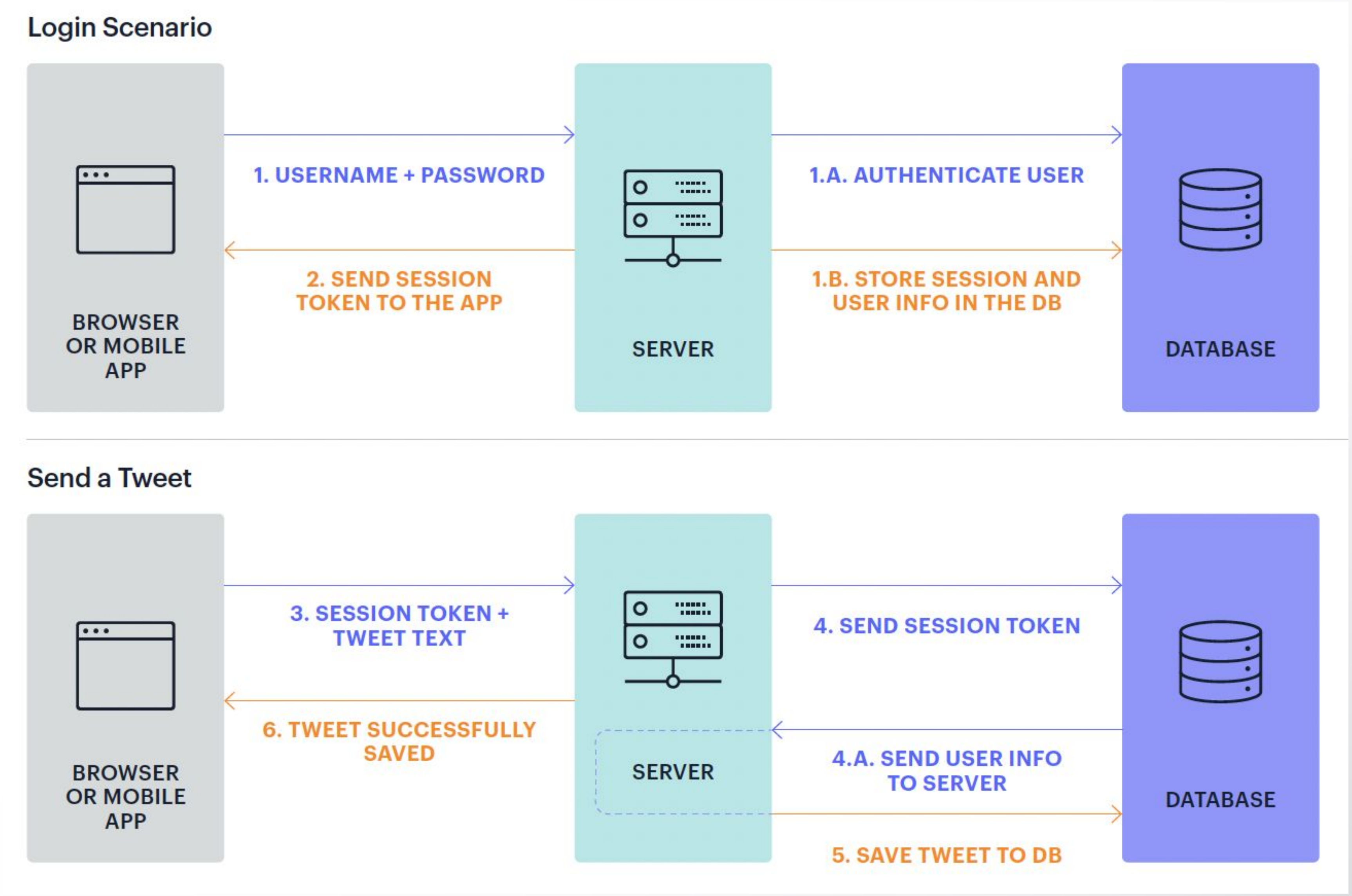
## 5. When will the session expire

- Whenever
- Once expired, re-login is enforced
- OAuth works a little differently

## 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.
- Session data is typically stored in a database.
- A “session token” is used to authorize and authenticate a user, and is exchanged between the server and client on every request.
- The server calls the database on every request to check if a session is valid and get the session information

# Storing Sessions in a Traditional Database



## Problem: Sessions stored in the database slow down requests

How to solve this:

- Eliminate database lookups for sessions completely
- Make the extra database lookup much faster so the additional hop won’t matter

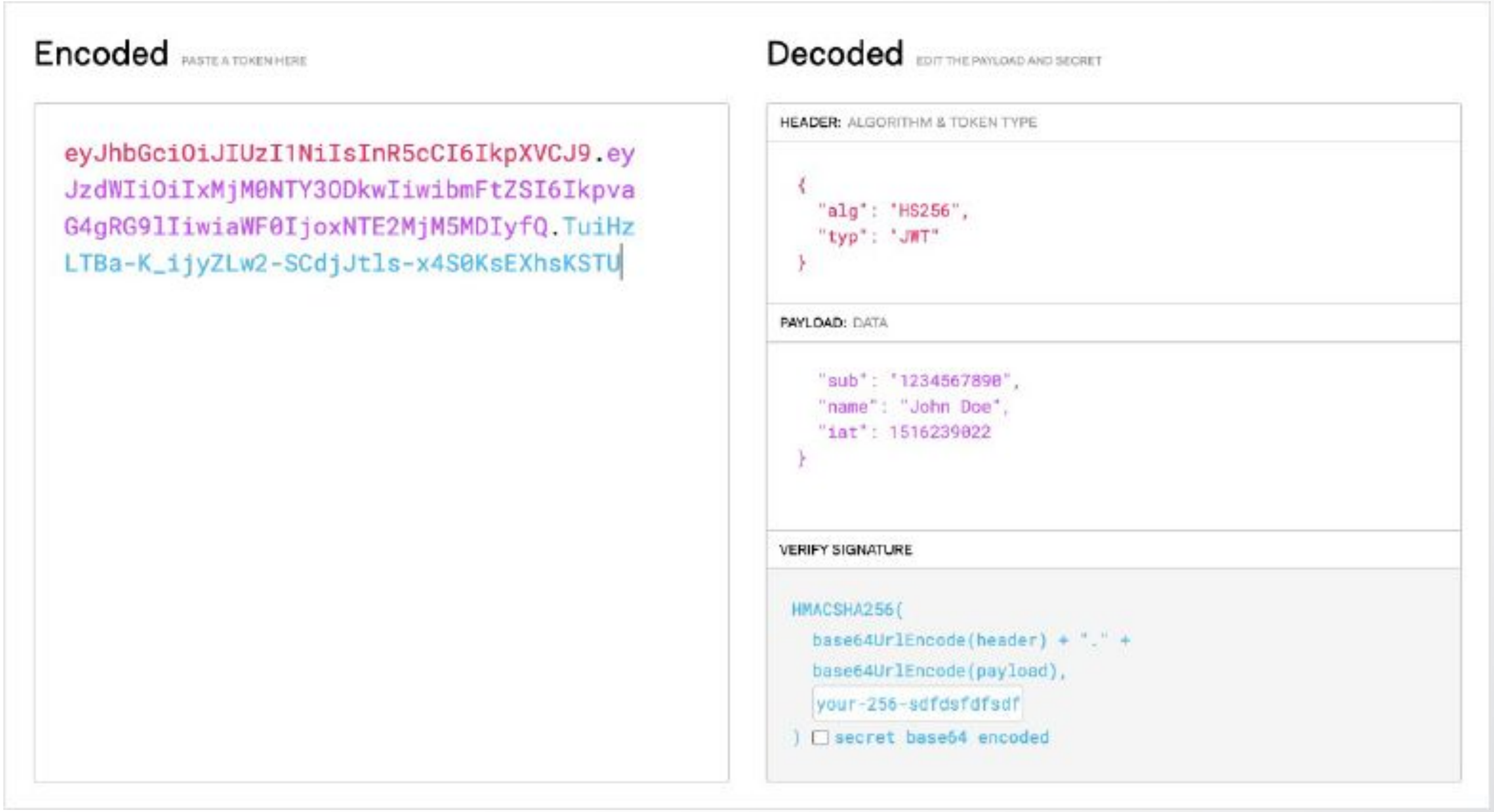
Eliminate lookups:

- Store the state in the server’s memory
- Use “sticky sessions”
- Use JSON Web Tokens

Make database lookups fast:

- Simply use Redis

## Using JWTs for Sessions



Encoded PASTE A TOKEN HERE

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.ey
JzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6Ikpva
G4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.TuiHz
LTBa-K_ijyZLw2-SCdjJtls-x4S0KsEXhsKSTU
```

Decoded EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

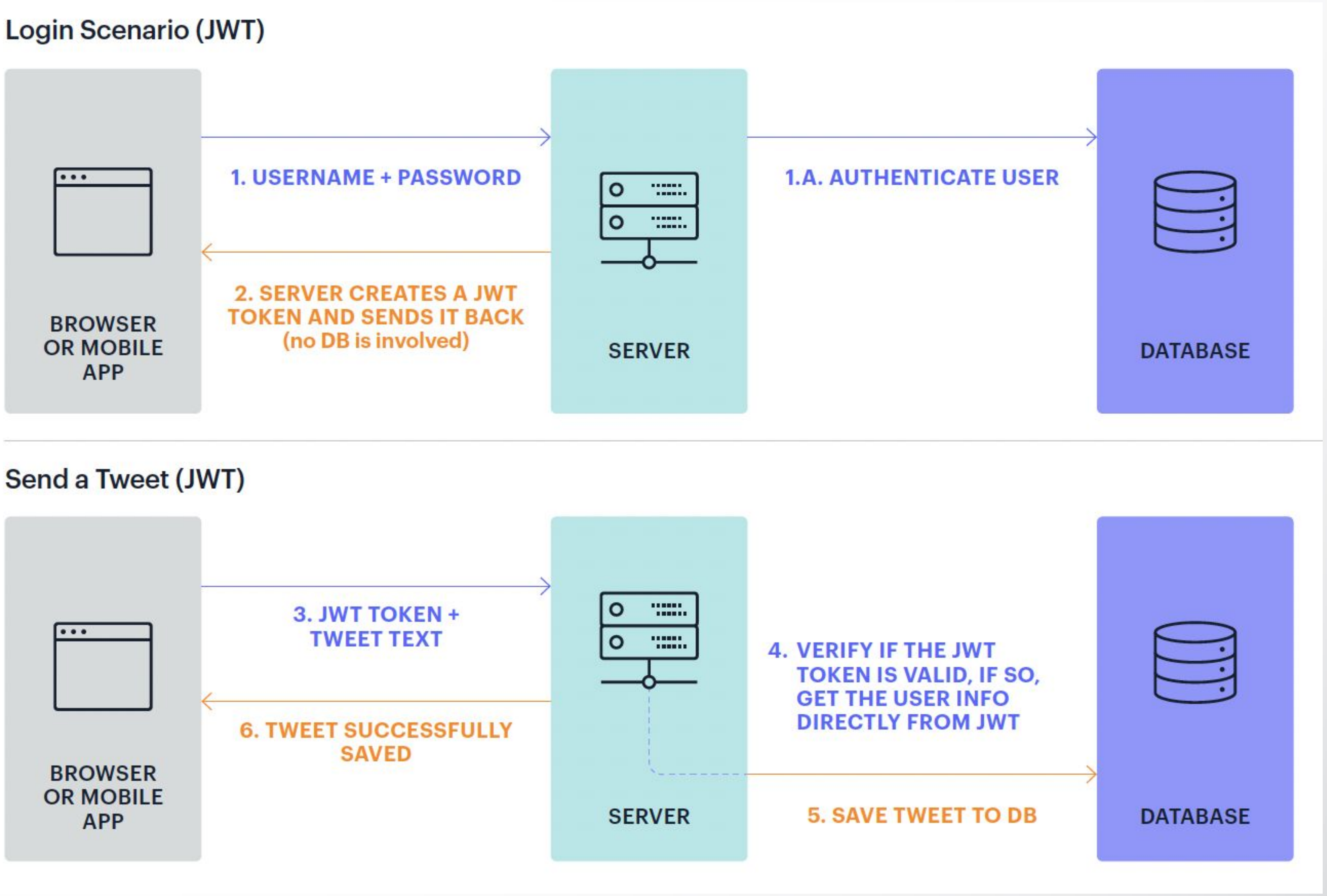
PAYLOAD: DATA

```
  "sub": "1234567890",
  "name": "John Doe",
  "iat": 1516239022
}
```

VERIFY SIGNATURE

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  your-256-sdfdsfdfsdf
) ☐ secret base64 encoded
```

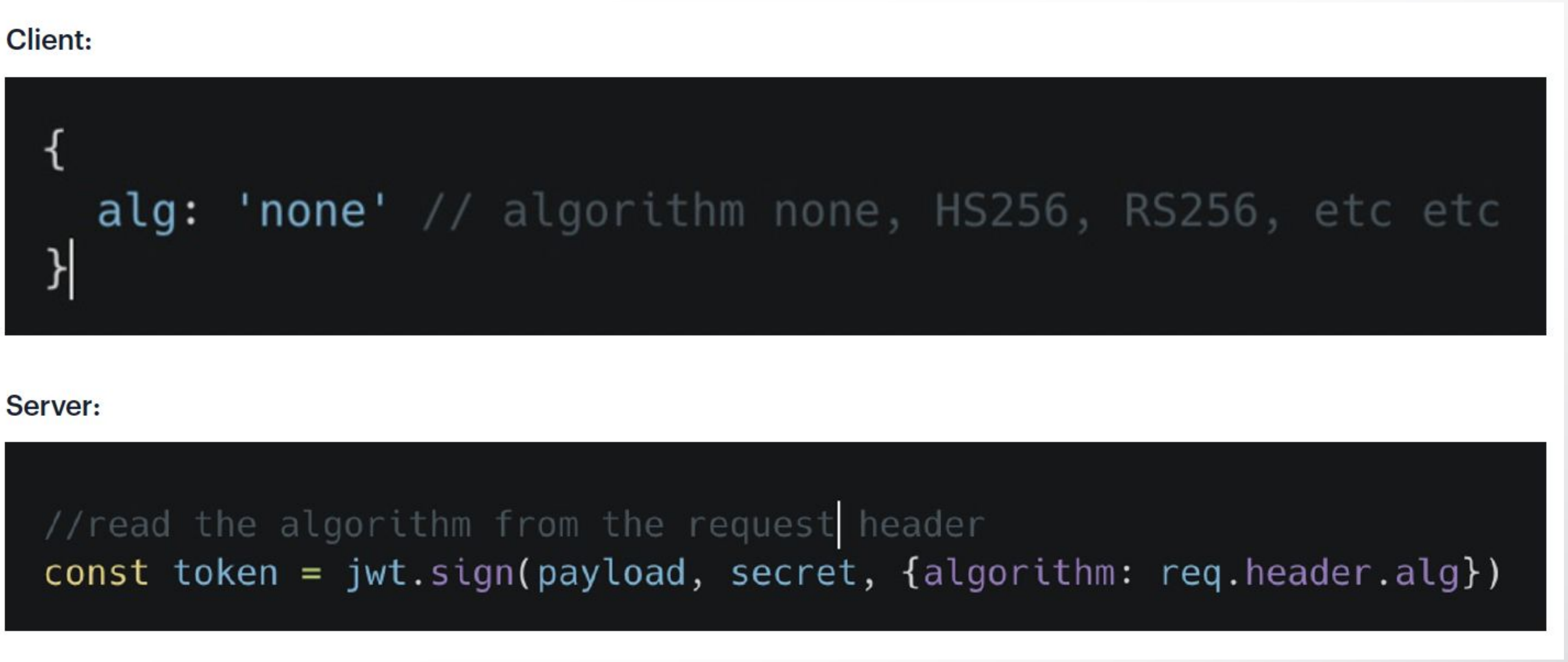
# Twitter with JWTs


Login Scenario (JWT)
BROWSER OR MOBILE APP
1. USERNAME + PASSWORD
2. SERVER CREATES A JWT TOKEN AND SENDS IT BACK (no DB is involved)
SERVER
1.A. AUTHENTICATE USER
DATABASE
Send a Tweet (JWT)
BROWSER OR MOBILE APP
3. JWT TOKEN + TWEET TEXT
6. TWEET SUCCESSFULLY SAVED
SERVER
4. VERIFY IF THE JWT TOKEN IS VALID, IF SO, GET THE USER INFO DIRECTLY FROM JWT
5. SAVE TWEET TO DB
DATABASE

## The JWT spec is too relaxed

- Written like HTML
- Provides workarounds to allow a variety of use cases (i.e. edge cases)
- Burdens engineers who must know how to avoid all the loopholes
- What might technically follow spec could also be insecure

## The “none” algorithm



Client:

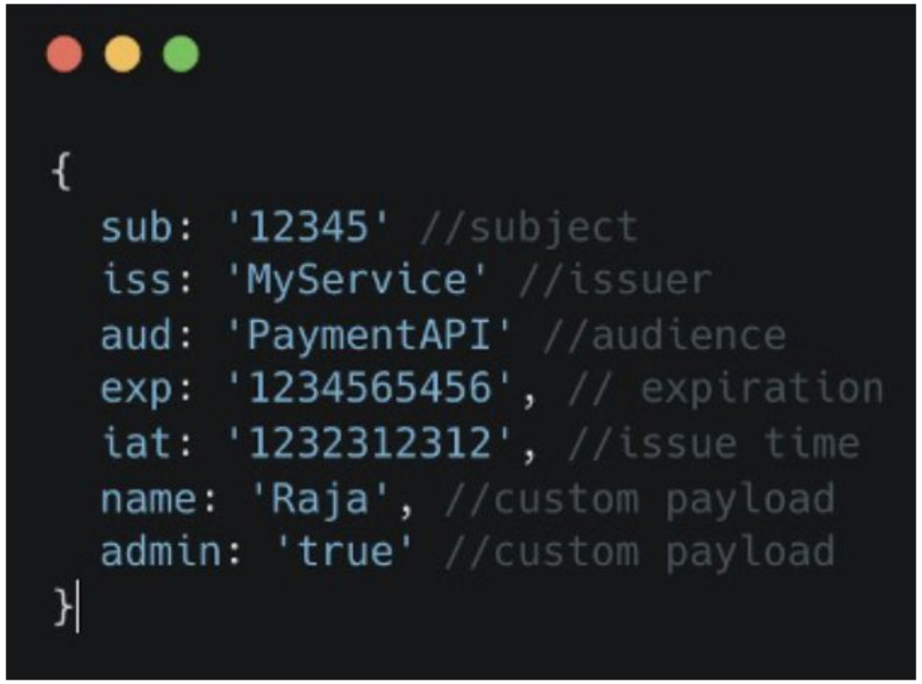
```
{
  alg: 'none' // algorithm none, HS256, RS256, etc etc
}
```

Server:

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

redis

## Claims are optional



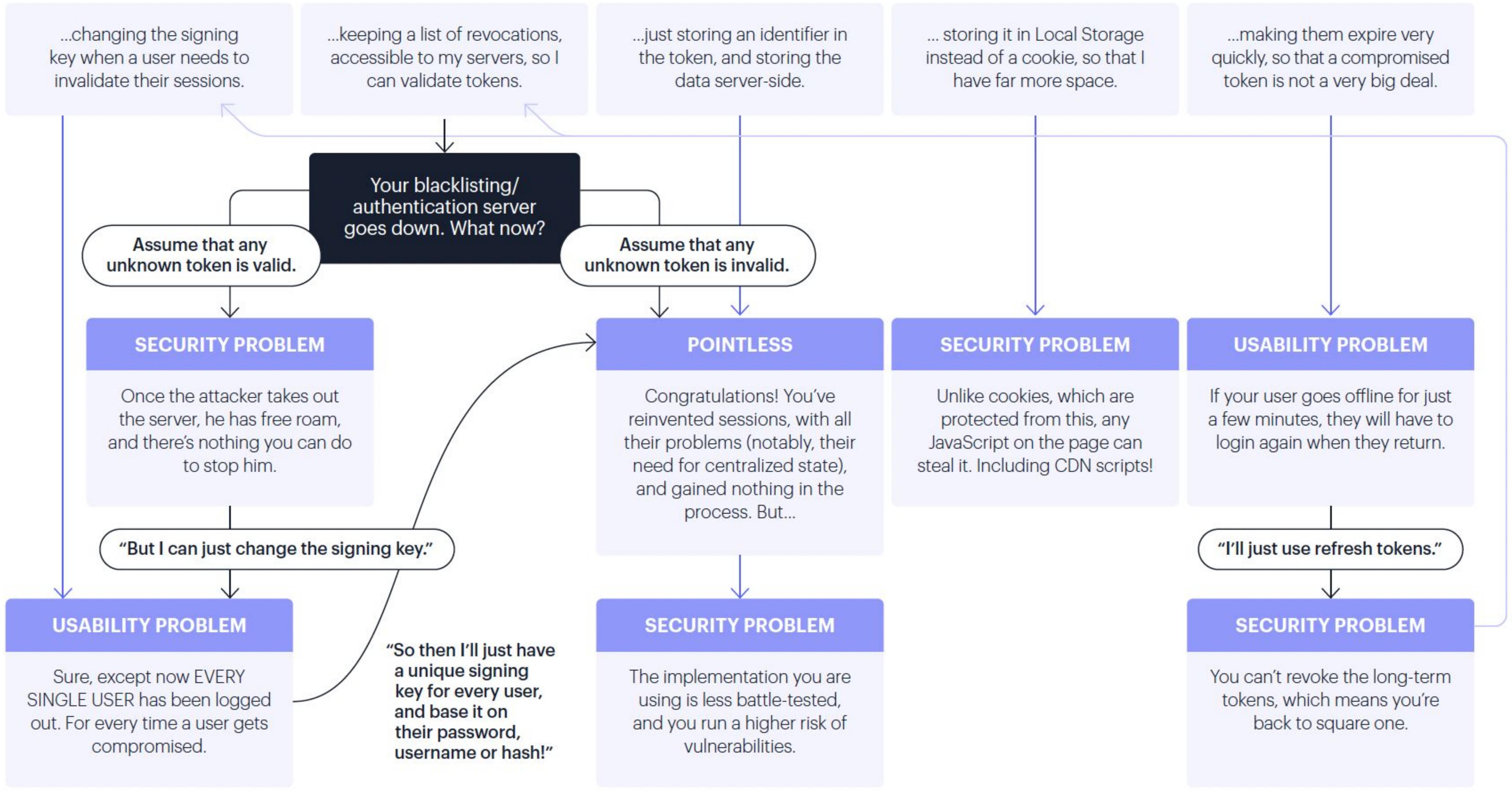
```
{
  sub: '12345' //subject
  iss: 'MyService' //issuer
  aud: 'PaymentAPI' //audience
  exp: '1234565456', // expiration
  iat: '1232312312', //issue time
  name: 'Raja', //custom payload
  admin: 'true' //custom payload
}
```

## Other considerations and issues

- Length of tokens
- Stateless? Not really

## Bottom line on why JWTs are dangerous

- Logout doesn’t really log you out
- Blocking users doesn’t immediately block them
- JWTs can contain stale data
- JWTs are often unencrypted



...changing the signing key when a user needs to invalidate their sessions.

...keeping a list of revocations, accessible to my servers, so I can validate tokens.

...just storing an identifier in the token, and storing the data server-side.

... storing it in Local Storage instead of a cookie, so that I have far more space.

...making them expire very quickly, so that a compromised token is not a very big deal.

Your blacklisting/ authentication server goes down. What now?

Assume that any unknown token is valid.

Assume that any unknown token is invalid.

**SECURITY PROBLEM**

Once the attacker takes out the server, he has free roam, and there's nothing you can do to stop him.

**POINTLESS**

Congratulations! You've reinvented sessions, with all their problems (notably, their need for centralized state), and gained nothing in the process. But...

**SECURITY PROBLEM**

Unlike cookies, which are protected from this, any JavaScript on the page can steal it. Including CDN scripts!

**USABILITY PROBLEM**

If your user goes offline for just a few minutes, they will have to login again when they return.

"But I can just change the signing key."

"I'll just use refresh tokens."

**USABILITY PROBLEM**

Sure, except now EVERY SINGLE USER has been logged out. For every time a user gets compromised.

"So then I'll just have a unique signing key for every user, and base it on their password, username or hash!"

**SECURITY PROBLEM**

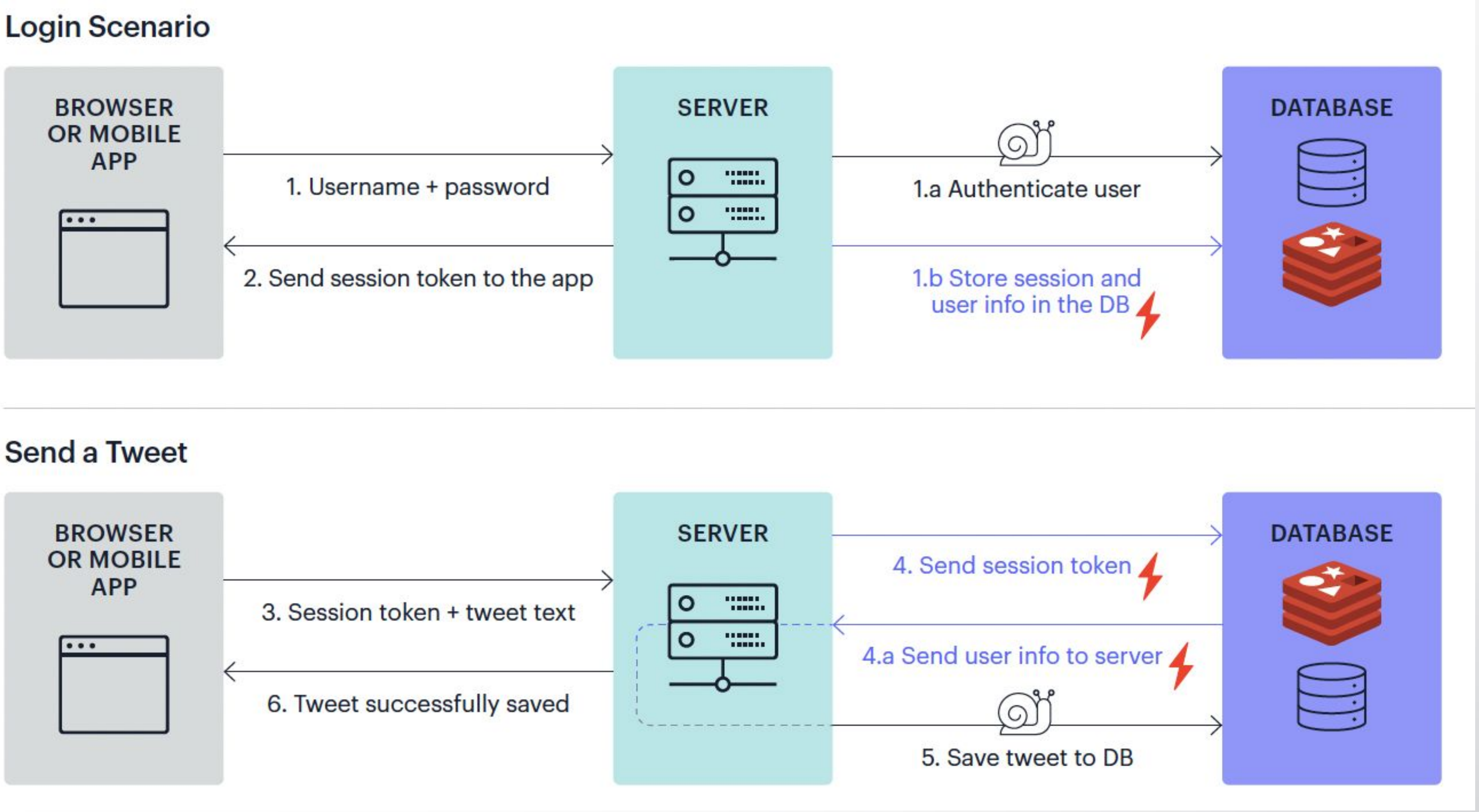
The implementation you are using is less battle-tested, and you run a higher risk of vulnerabilities.

**SECURITY PROBLEM**

You can't revoke the long-term tokens, which means you're back to square one.

Source: http://cryto.net/~joepie91/blog/2016/06/19/stop-using-jwt-for-sessions-part-2-why-your-solution-doesnt-work/

# Storing Sessions in Redis



redis

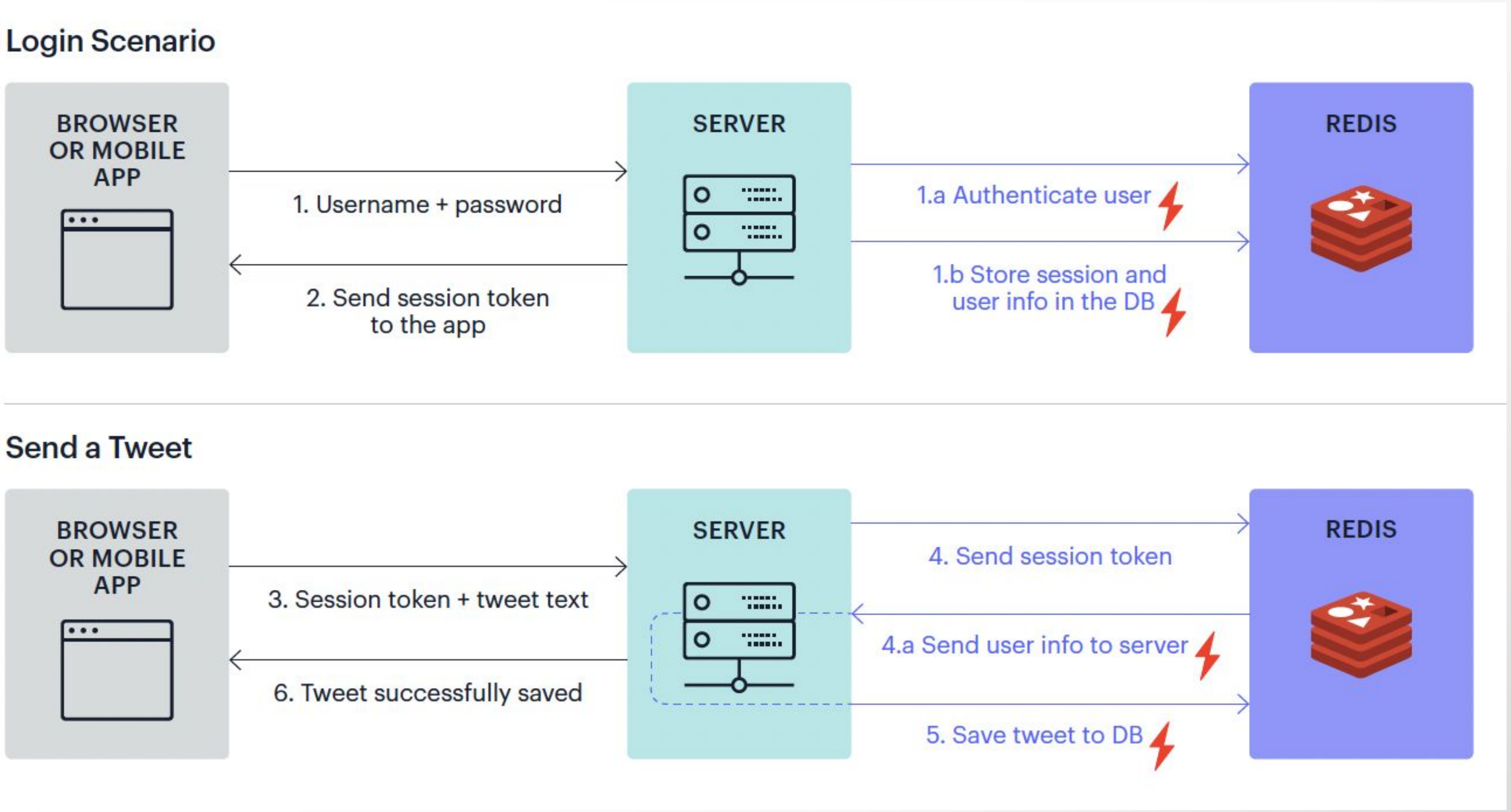
## Storing Sessions in Redis with Node and Express

```
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,
  })
)
```

```
// 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!')
  }
})
```

redis

# Sessions with Redis as the primary database



# Thank You

youtube.com/c/Redisinc

developer.redis.com

redis.com/try-free?coupon=JWT3

redis

# Q & A

youtube.com/c/Redisinc
developer.redis.com
redis.com/try-free?coupon=JWT3

redis