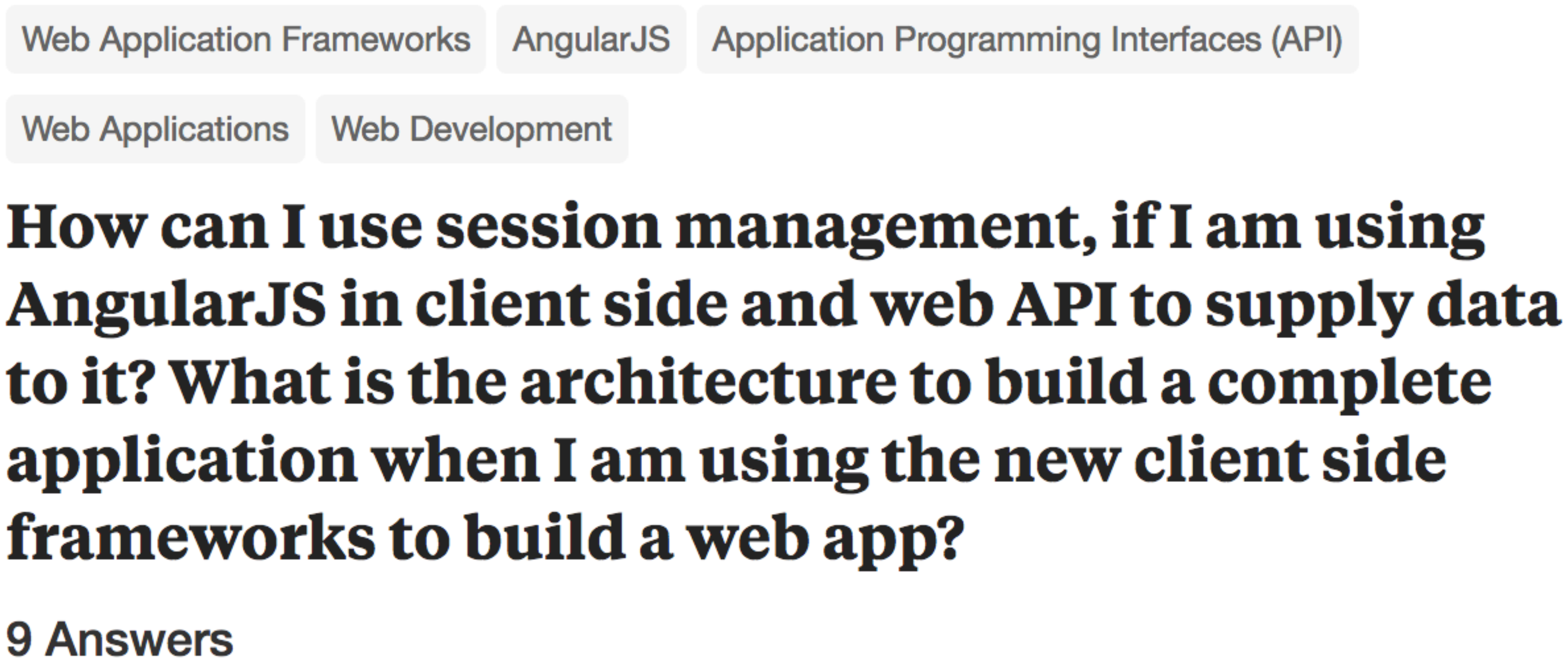
Pragmatic Web Security

Security training for developers

The truth about cookies, tokens and APIs

Philippe De Ryck

@PhilippeDeRyck – philippe@PragmaticWebSecurity.com

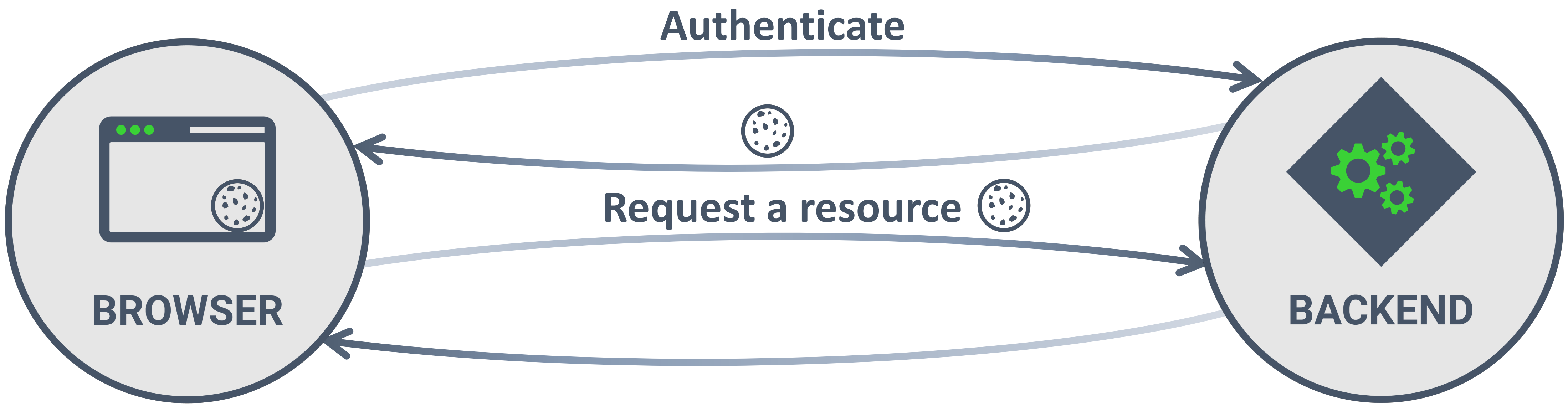


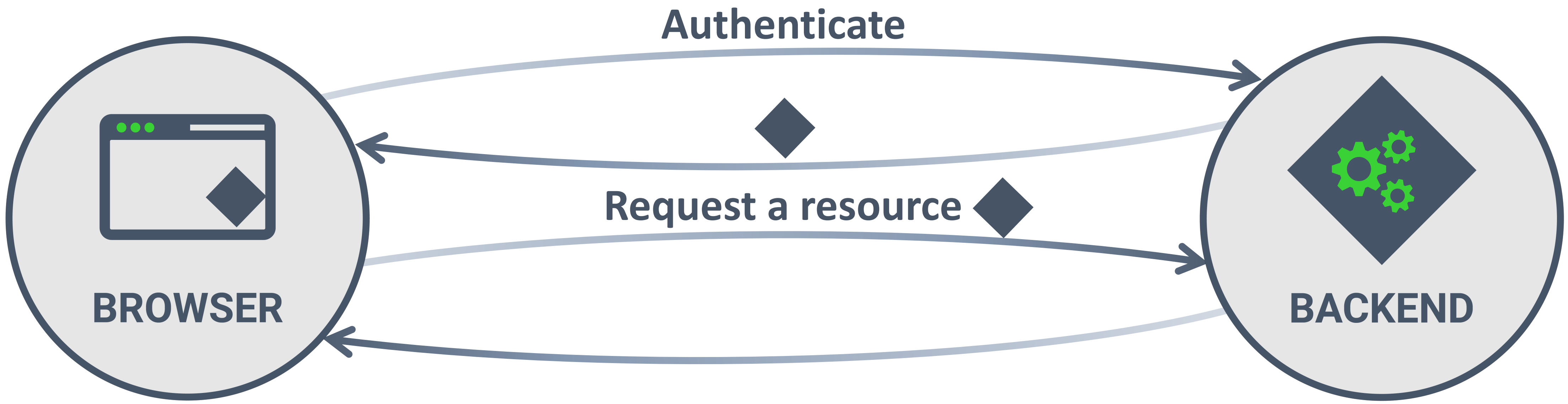
Web Application Frameworks AngularJS Application Programming Interfaces (API)

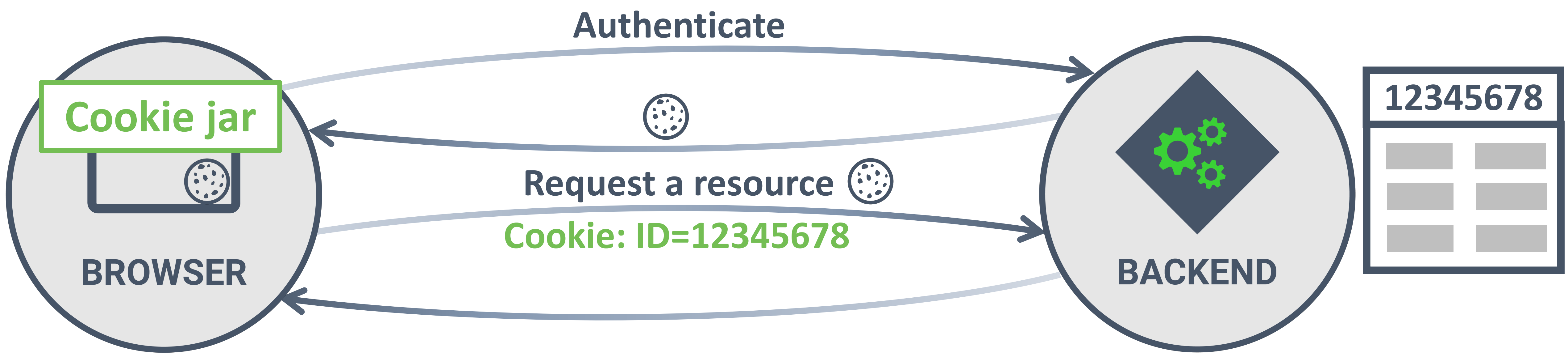
Web Applications Web Development

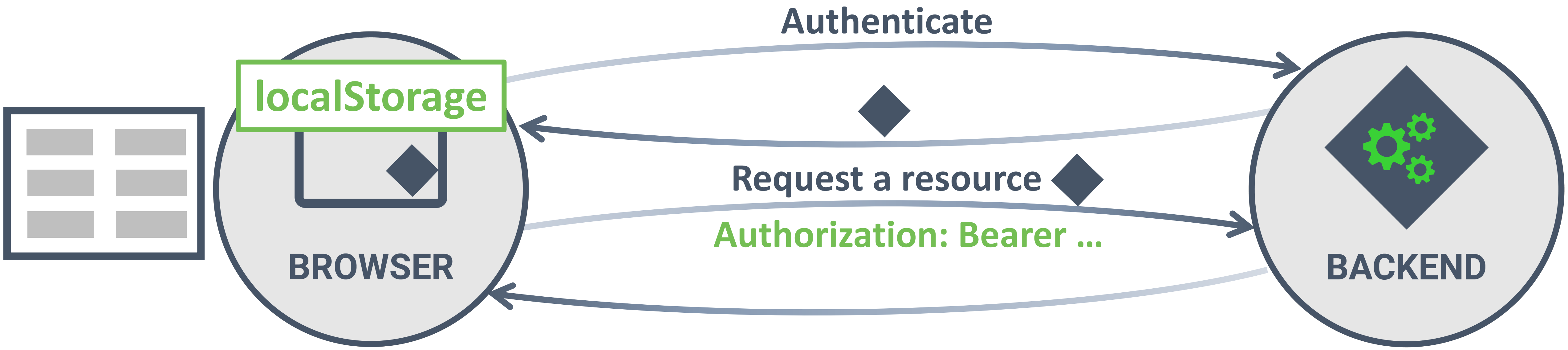
How can I use session management, if I am using AngularJS in client side and web API to supply data to it? What is the architecture to build a complete application when I am using the new client side frameworks to build a web app?

9 Answers


Authenticate
Request a resource
BROWSER
BACKEND


Authenticate
Request a resource
BROWSER
BACKEND


Authenticate
Cookie jar
Request a resource
Cookie: ID=12345678
BROWSER
BACKEND
12345678


Authenticate
localStorage
Request a resource
Authorization: Bearer …
BROWSER
BACKEND

DR. PHILIPPE DE RYCK

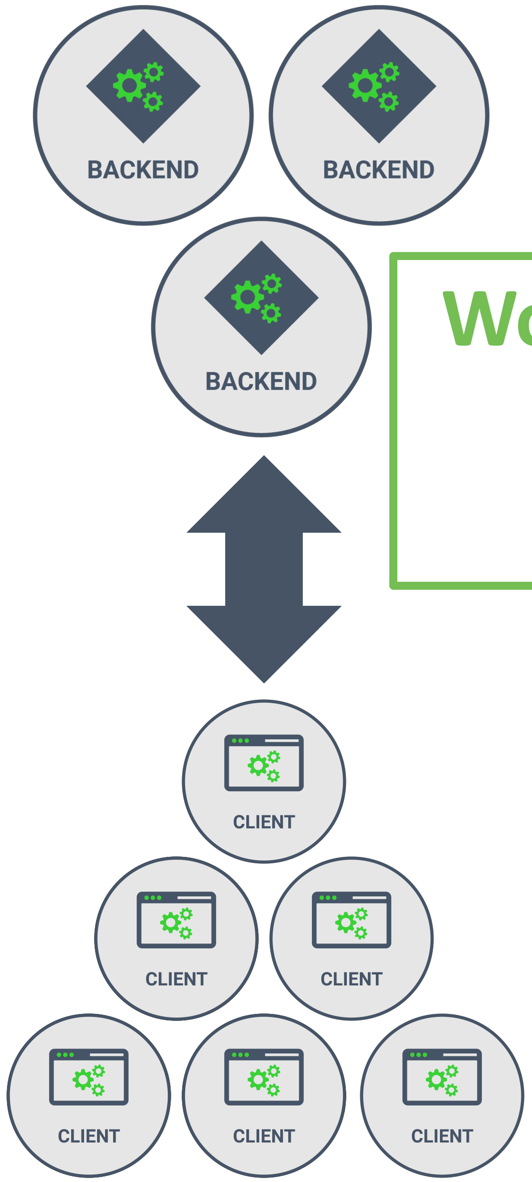
- Deep understanding of the web security landscape
- Google Developer Expert (not employed by Google)
- Author of the *primer on client-side web security*
- Course curator of the SecAppDev course (https://secappdev.org)

@PHILIPPEDERYCK
HTTPS://PRAGMATICWEBSECURITY.COM

Pragmatic Web Security

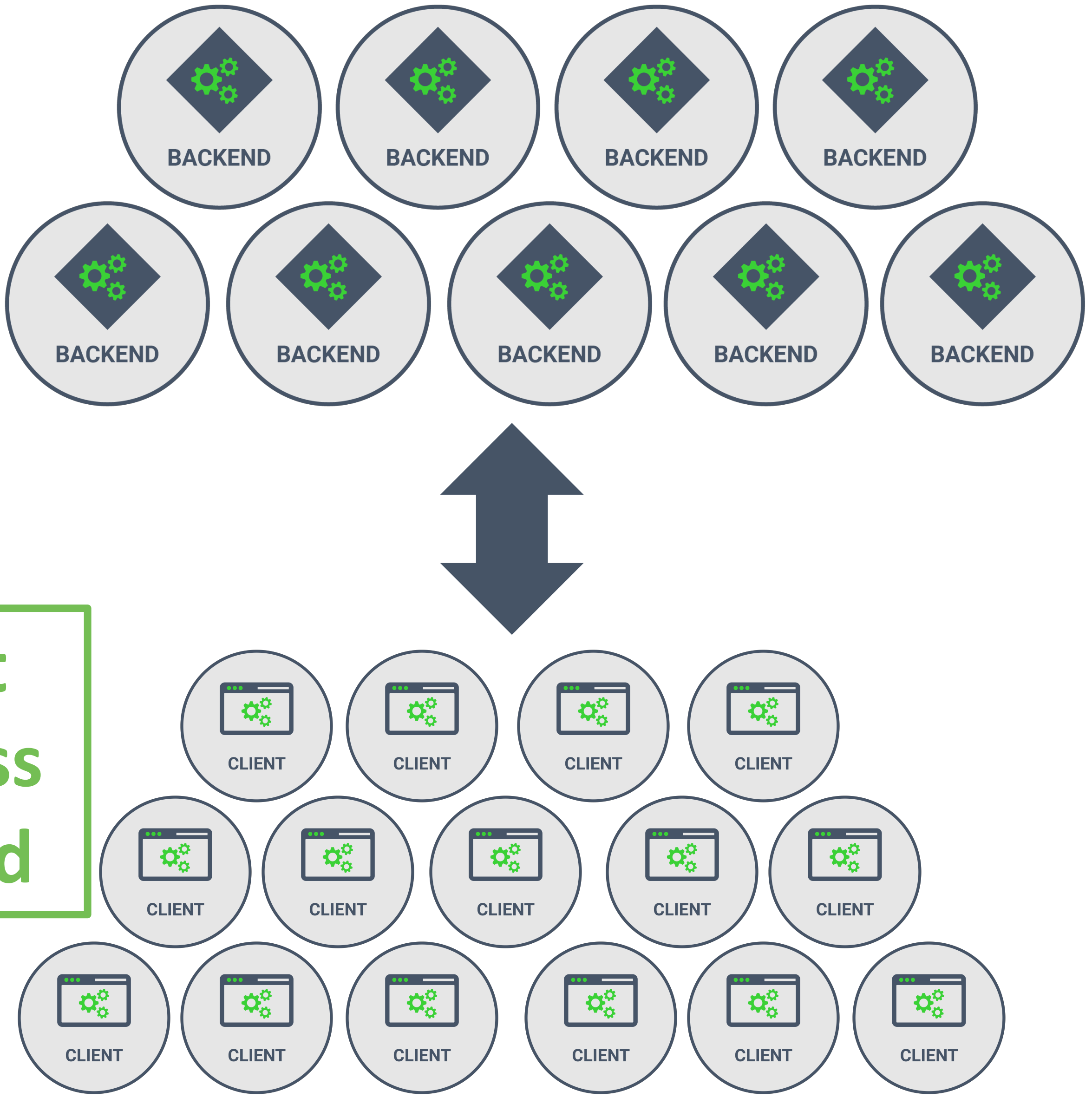
High-quality security training for developers and managers

Custom courses covering web security, API security, Angular security, …



Works fine with a stateful backend

Might benefit from a stateless (REST) backend



Do not overthink statelessness

There are various degrees of statelessness, each with its own use cases.

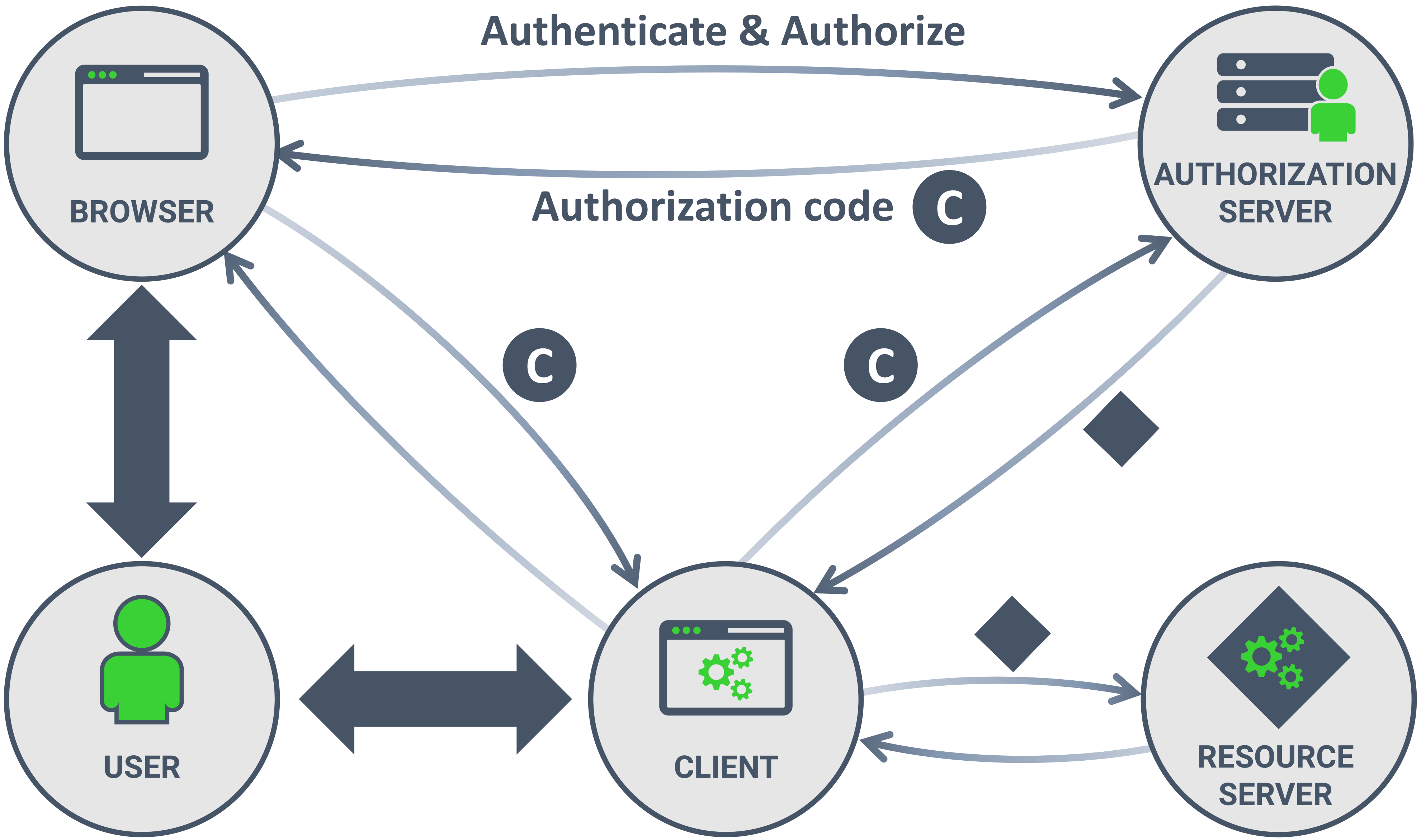
Build your API according to your requirements.

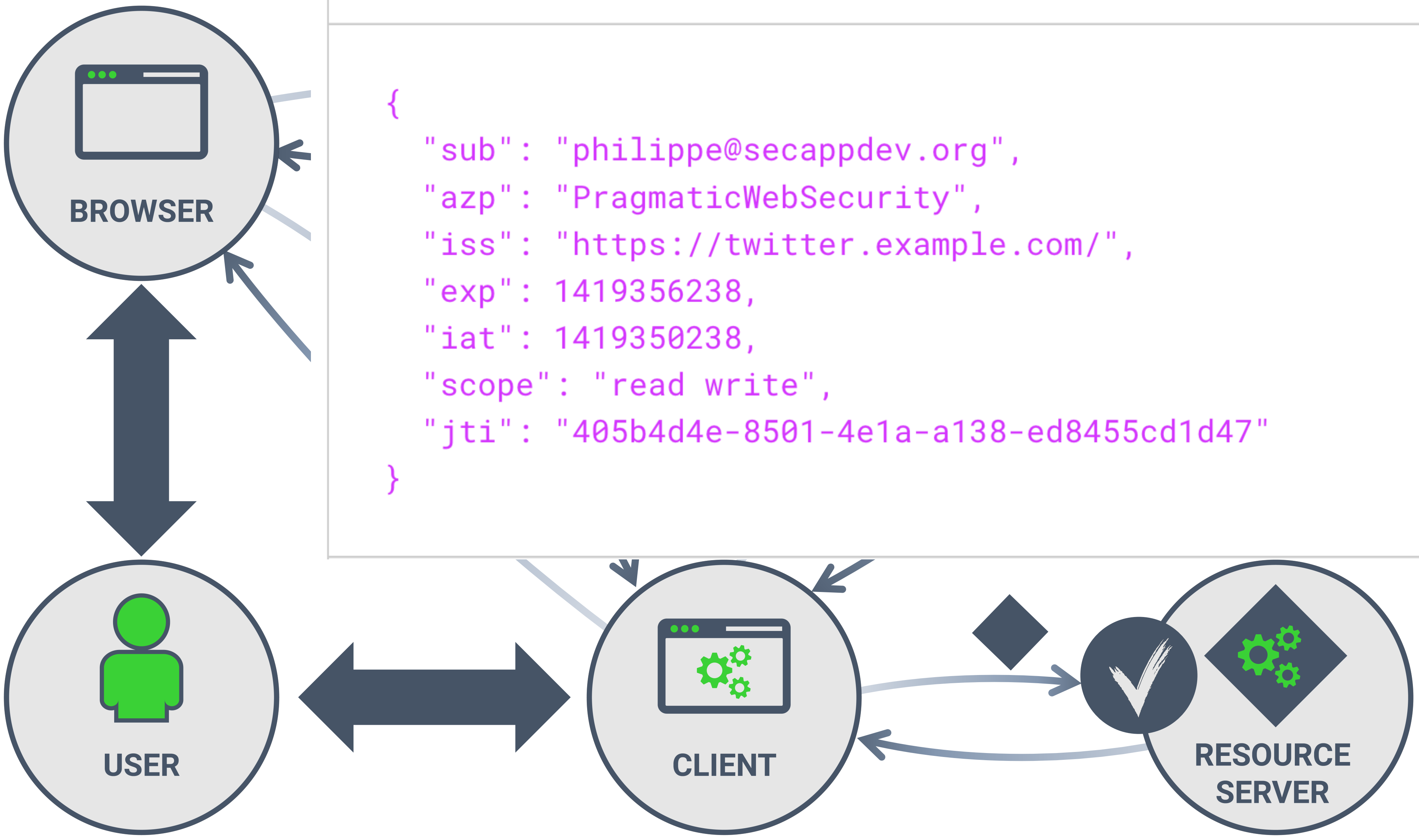
Cookie: **ID=12345678**

Authorization: **Bearer 12345678**

Cookie: **JWT=eyJhbGci…**

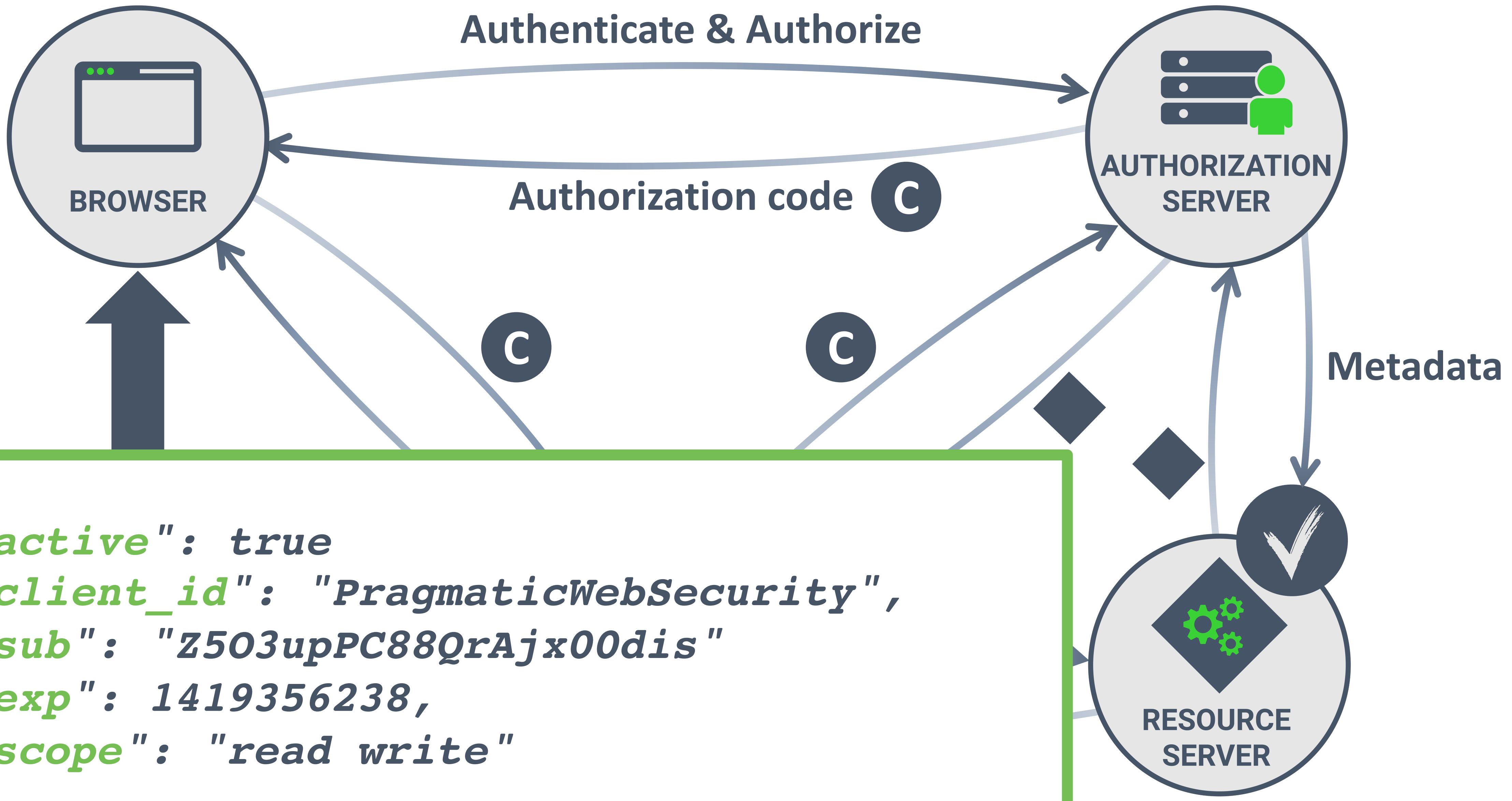
Authorization: **Bearer eyJhbGci…**


Authenticate & Authorize
Authorization code
C
BROWSER
AUTHORIZATION SERVER
C
C
USER
CLIENT
RESOURCE SERVER

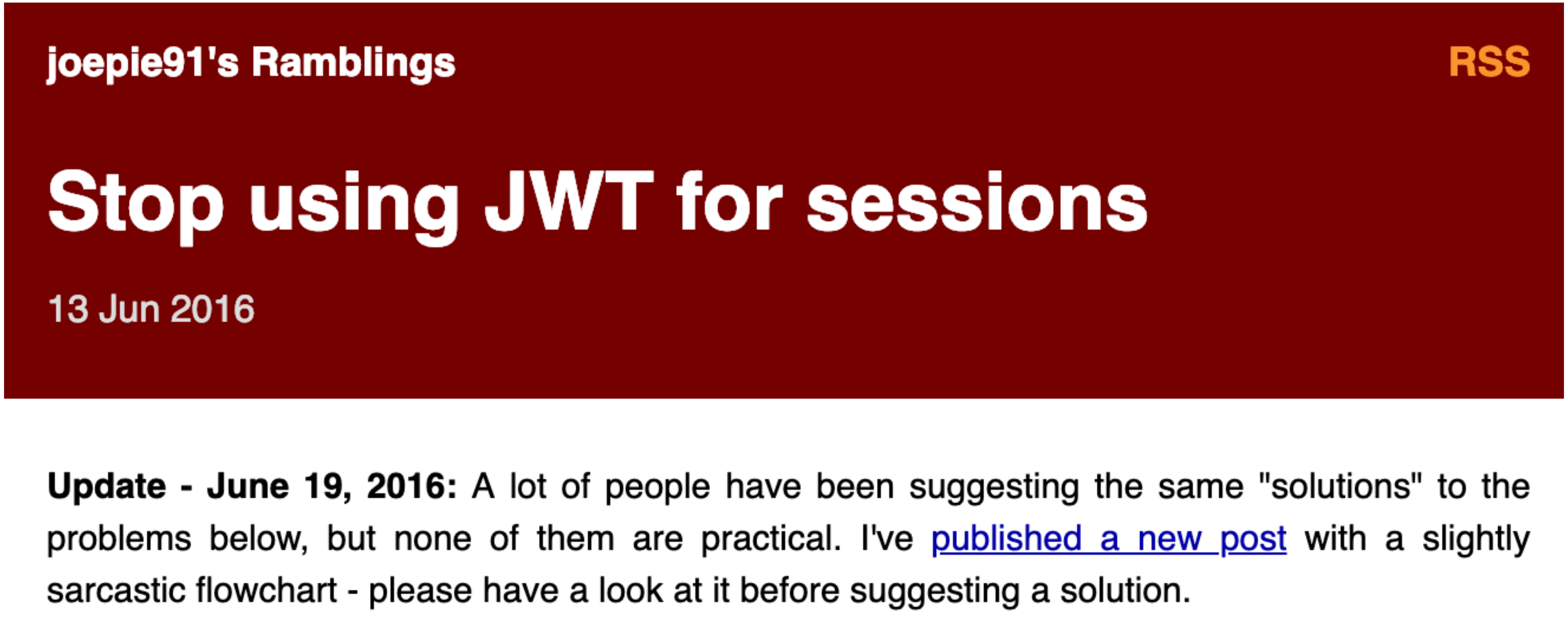


PAYLOAD: DATA

```
{
  "sub": "philippe@secappdev.org",
  "azp": "PragmaticWebSecurity",
  "iss": "https://twitter.example.com/",
  "exp": 1419356238,
  "iat": 1419350238,
  "scope": "read write",
  "jti": "405b4d4e-8501-4e1a-a138-ed8455cd1d47"
}
```



```
{
    "active": true
    "client_id": "PragmaticWebSecurity",
    "sub": "Z5O3upPC88QrAjx00dis"
    "exp": 1419356238,
    "scope": "read write"
}
```


joepie91's Ramblings RSS

Stop using JWT for sessions

13 Jun 2016

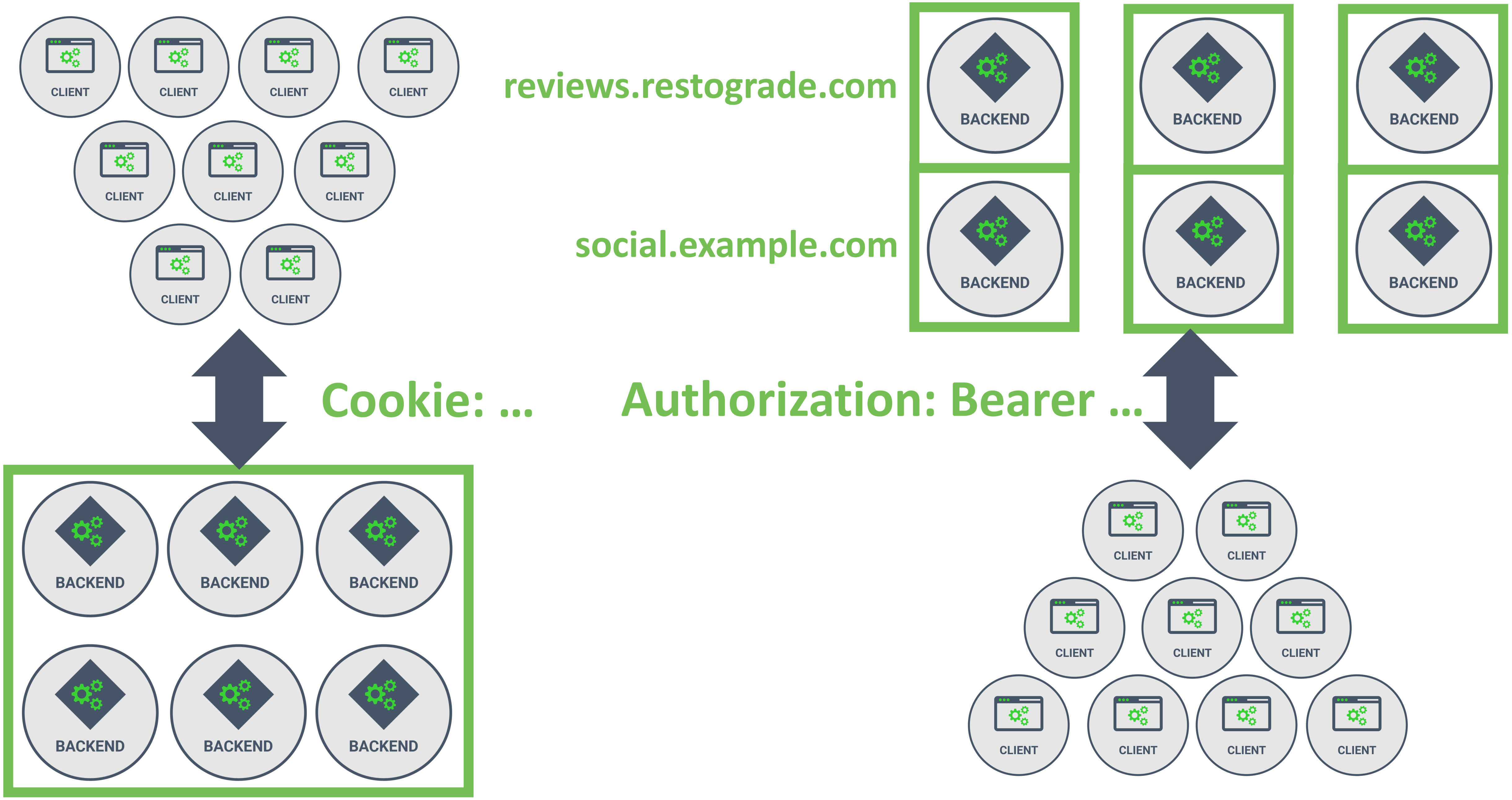
Update - June 19, 2016: A lot of people have been suggesting the same "solutions" to the problems below, but none of them are practical. I've published a new post with a slightly sarcastic flowchart - please have a look at it before suggesting a solution.

“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.”

Differentiate the mechanism from the value

Cookies or the Authorization header can be used to transport authorization state.

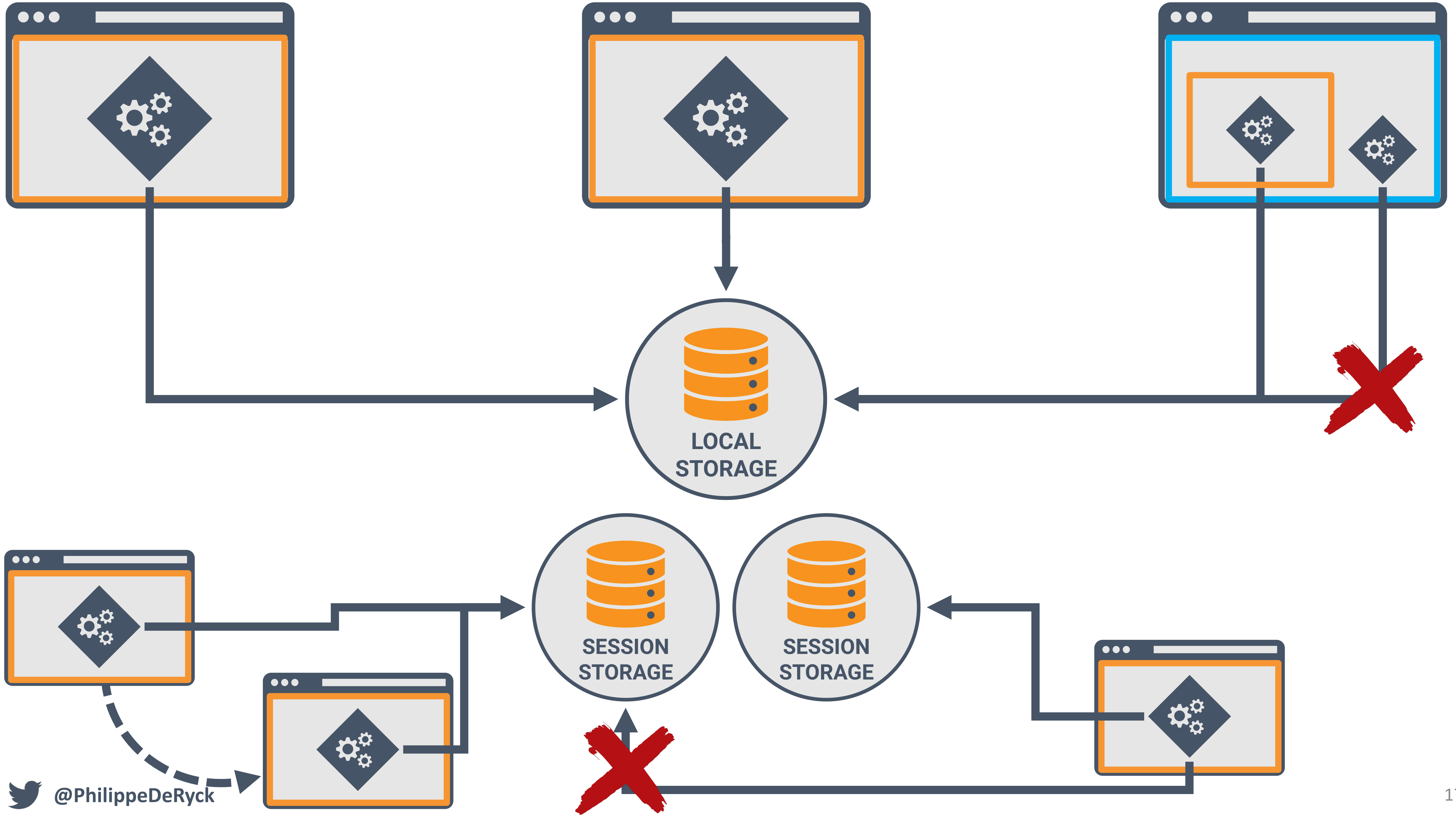
Both can contain any type of String-based value.

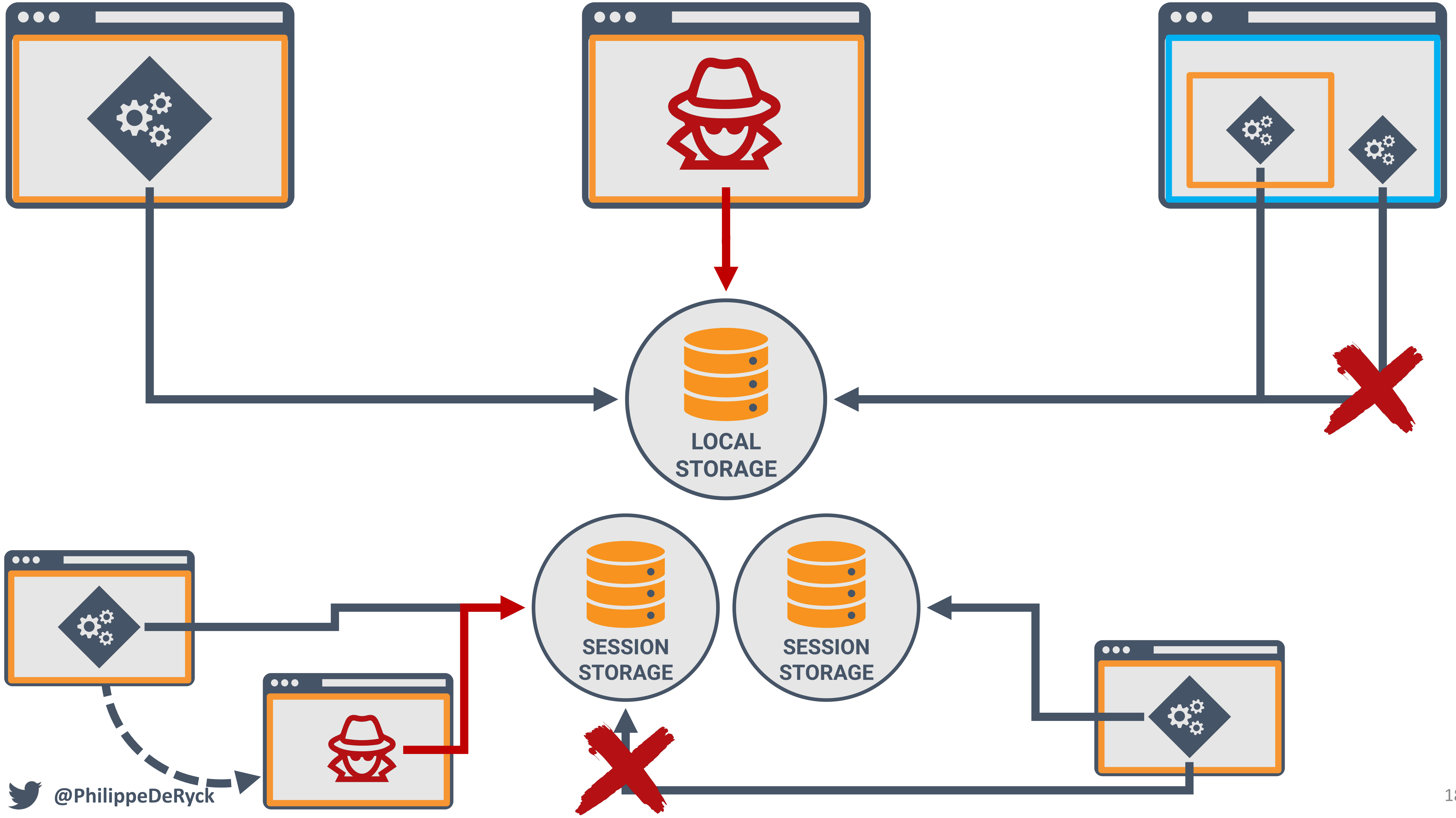

CLIENT
reviews.restogr ade.com
social.example.com
BACKEND
Cookie: …
Authorization: Bearer …
api.restogr762ade.com

TAKE YOUR DEPLOYMENT SCENARIO INTO ACCOUNT

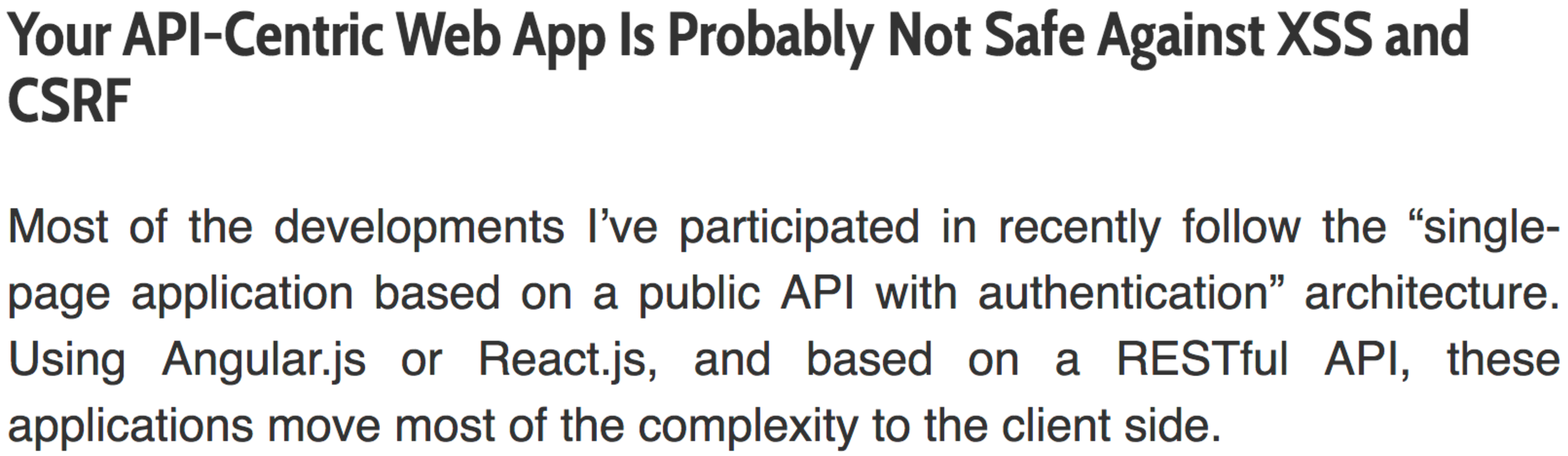
Cookies only work well with a single backend domain.

The Authorization header can be sent
to multiple domains.


LOCAL STORAGE
SESSION STORAGE
SESSION STORAGE



@PhilippeDeRyck



Your API-Centric Web App Is Probably Not Safe Against XSS and CSRF

Most of the developments I've participated in recently follow the "single-page application based on a public API with authentication" architecture. Using Angular.js or React.js, and based on a RESTful API, these applications move most of the complexity to the client side.

"The browser offers a storage that can't be read by JavaScript: HttpOnly cookies. It's a good way to identify a requester without risking XSS attacks."

HttpOnly cookies

HttpOnly is useful, but not as an XSS defense

The deal with HttpOnly

- The ***HttpOnly*** flag resolves a consequence of an XSS attack
  - Stealing the session identifier becomes a lot harder
  - **But you still have an XSS vulnerability in your application**
    - XSS allows the attacker to execute arbitrary code
    - That code can trigger authenticated requests, modify the DOM, ...

- ***HttpOnly*** is still recommended, because it raises the bar
  - XSS attacks become a little bit harder to execute and to persist
  - XSS attacks from subdomains become less powerful (with domain-based cookies)

- In Chrome, ***HttpOnly*** prevents cookies from entering the rendering process
  - Useful to reduce the impact of CPU-based ***Spectre*** and ***Meltdown*** attacks

COMPARING CLIENT-SIDE STORAGE MECHANISMS

LOCALSTORAGE	SESSIONSTORAGE	IN-MEMORY	COOKIES
Survives a page reload	Survives a page reload	Does not survive a page reload	Survives a page reload
Available to the entire origin	Available to the window and children	Available to running code only	Available on outgoing requests
Cannot be shielded from malicious code	Can be a bit shielded from malicious code	Can be shielded from malicious code	Can be shielded from malicious code
Can be abused in case of XSS	**Can be abused in case of XSS**	**Can be abused in case of XSS**	**Can be abused in case of XSS**

Don't underestimate XSS

Contrary to custom storage areas, cookies can be fully hidden from JavaScript, preventing theft through XSS.

XSS is the problem here, and HttpOnly *will not save you*

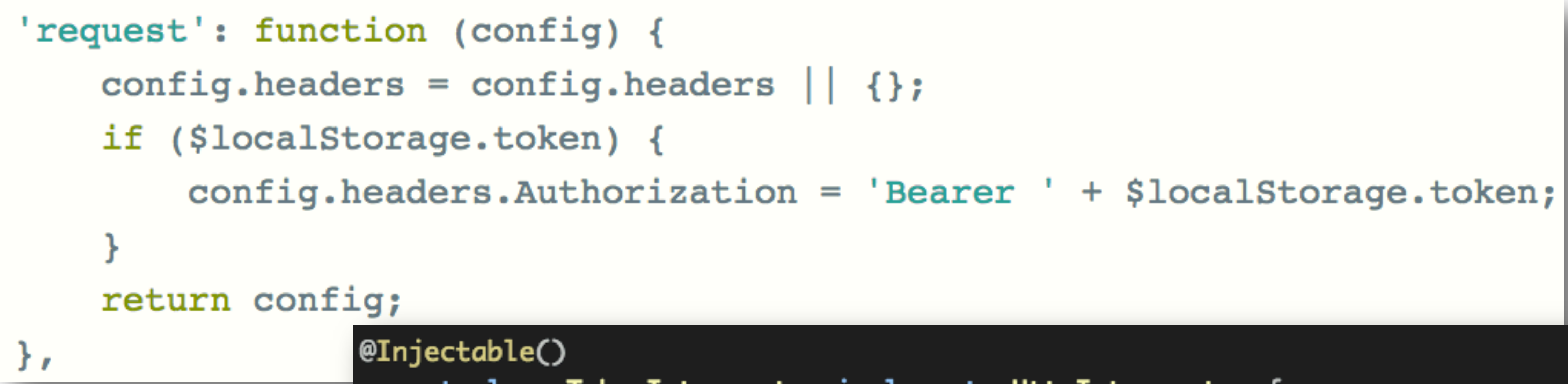
```
Set-Cookie: name=value
```

```
Set-Cookie: name=value; Secure
```

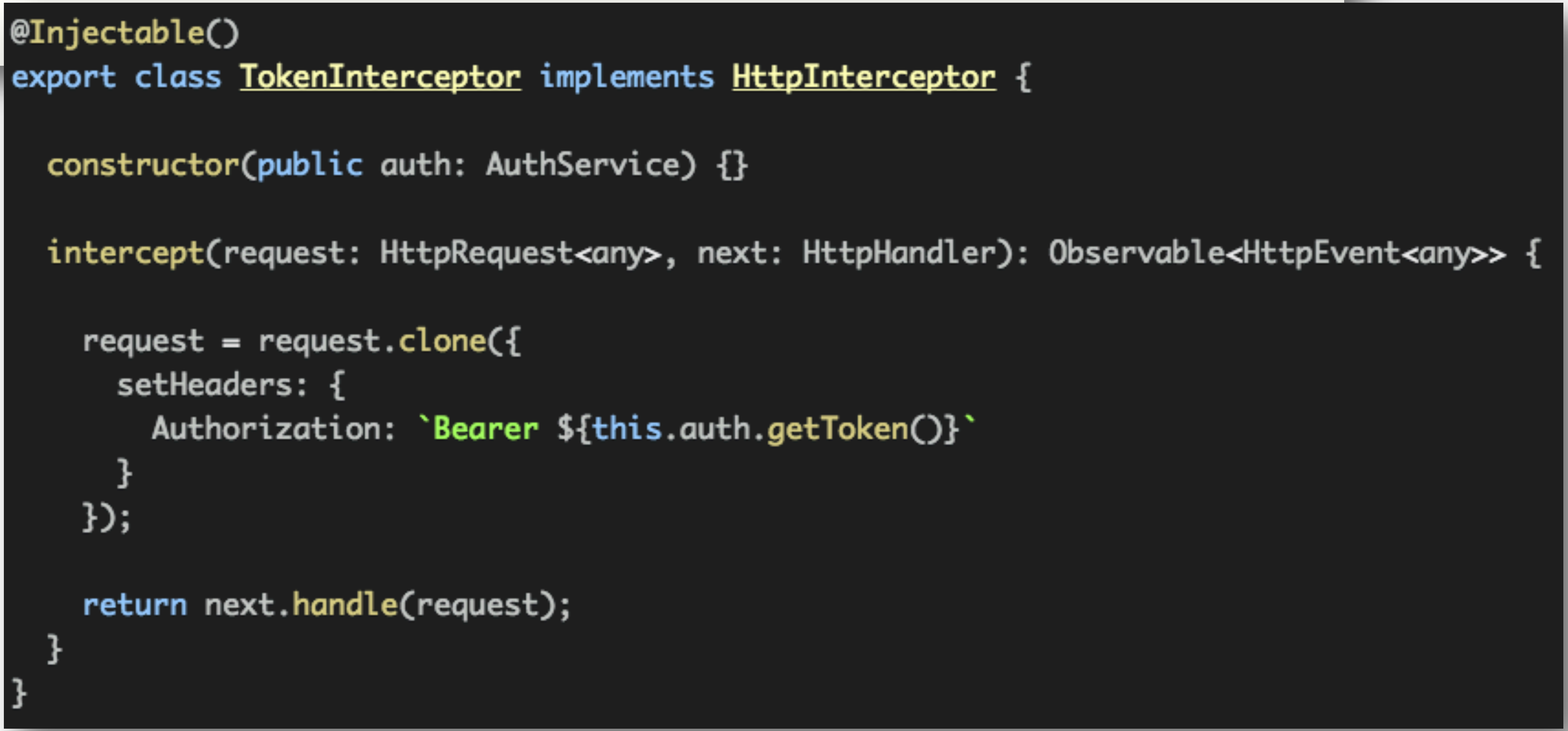
```
Set-Cookie: name=value; Secure; HttpOnly
```

```
Set-Cookie: __Secure-name=value; Secure; HttpOnly
```

```
Set-Cookie: __Host-name=value; Secure; HttpOnly
```



```
'request': function (config) {
    config.headers = config.headers || {};
    if ($localStorage.token) {
        config.headers.Authorization = 'Bearer ' + $localStorage.token;
    }
    return config;
},
```



```
@Injectable()
export class TokenInterceptor implements HttpInterceptor {

  constructor(public auth: AuthService) {}

  intercept(request: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {

    request = request.clone({
      setHeaders: {
        Authorization: `Bearer ${this.auth.getToken()}`
      }
    });

    return next.handle(request);
  }
}
```

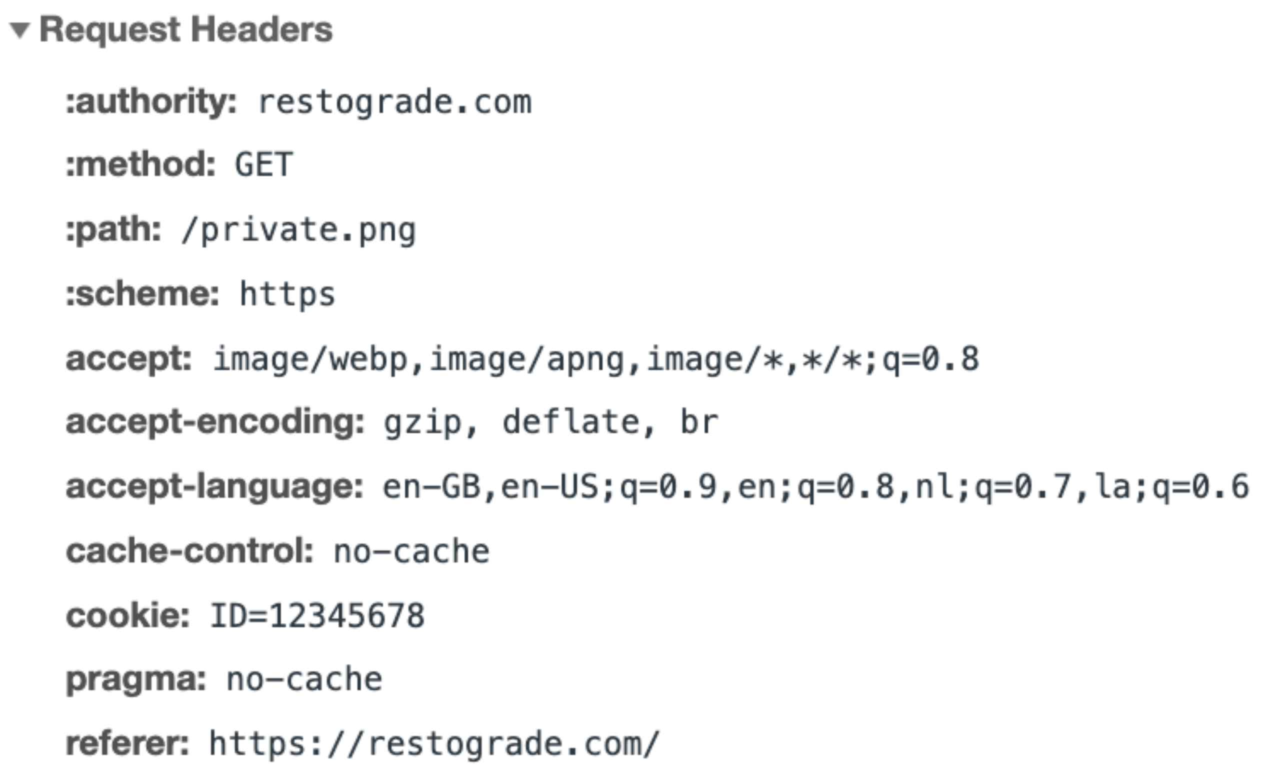
```
import { JwtModule } from '@auth0/angular-jwt';
import { HttpClientModule } from '@angular/common/http';

export function tokenGetter() {
  return localStorage.getItem('access_token');
}

@NgModule({
  bootstrap: [AppComponent],
  imports: [
    // ...
    HttpClientModule,
    JwtModule.forRoot({
      config: {
        tokenGetter: tokenGetter,
        whitelistedDomains: ['localhost:3001'],
        blacklistedRoutes: ['localhost:3001/auth/']
      }
    })
  ]
})
export class AppModule {}
```

There is no free lunch

Both mechanisms require effort to secure.

Cookies need flags and prefixes, and the Authorization header needs to be controlled in code

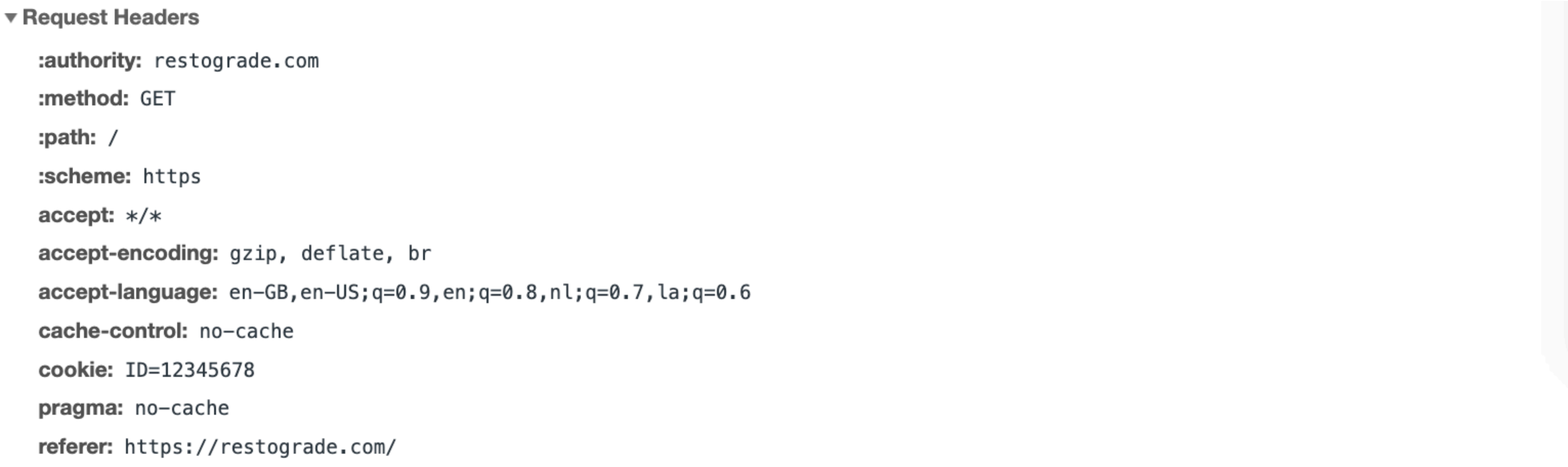
```
<img src="https://restograde.com/private.png">
<script src="admin/allTheCode.js"></script>
```



▾ Request Headers

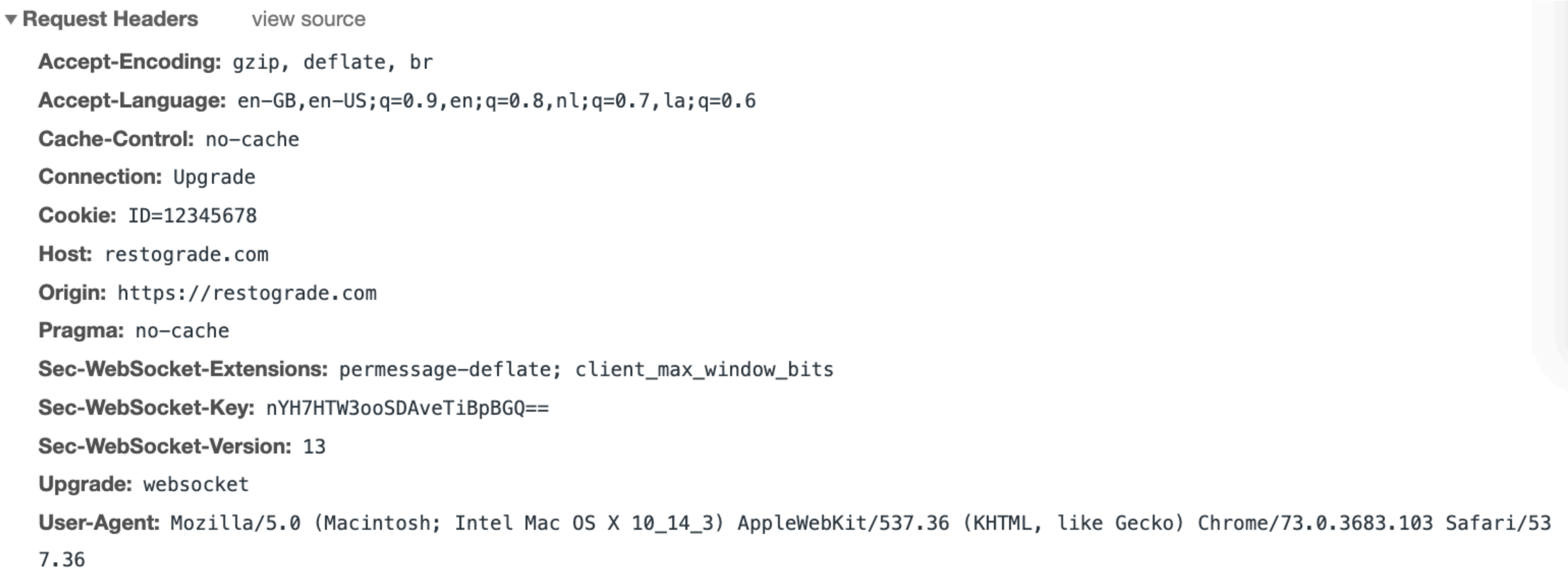
```
:authority: restograde.com
:method: GET
:path: /private.png
:scheme: https
accept: image/webp,image/apng,image/*,*/*;q=0.8
accept-encoding: gzip, deflate, br
accept-language: en-GB,en-US;q=0.9,en;q=0.8,nl;q=0.7,la;q=0.6
cache-control: no-cache
cookie: ID=12345678
pragma: no-cache
referer: https://restograde.com/
```

```
xhr = new XMLHttpRequest();
xhr.open("GET", "https://restograde.com");
xhr.withCredentials = true;
xhr.send();
```



```
▾ Request Headers
    :authority: restograde.com
    :method: GET
    :path: /
    :scheme: https
    accept: */*
    accept-encoding: gzip, deflate, br
    accept-language: en-GB,en-US;q=0.9,en;q=0.8,nl;q=0.7,la;q=0.6
    cache-control: no-cache
    cookie: ID=12345678
    pragma: no-cache
    referer: https://restograde.com/
```

```
new WebSocket("wss://restograde.com");
```

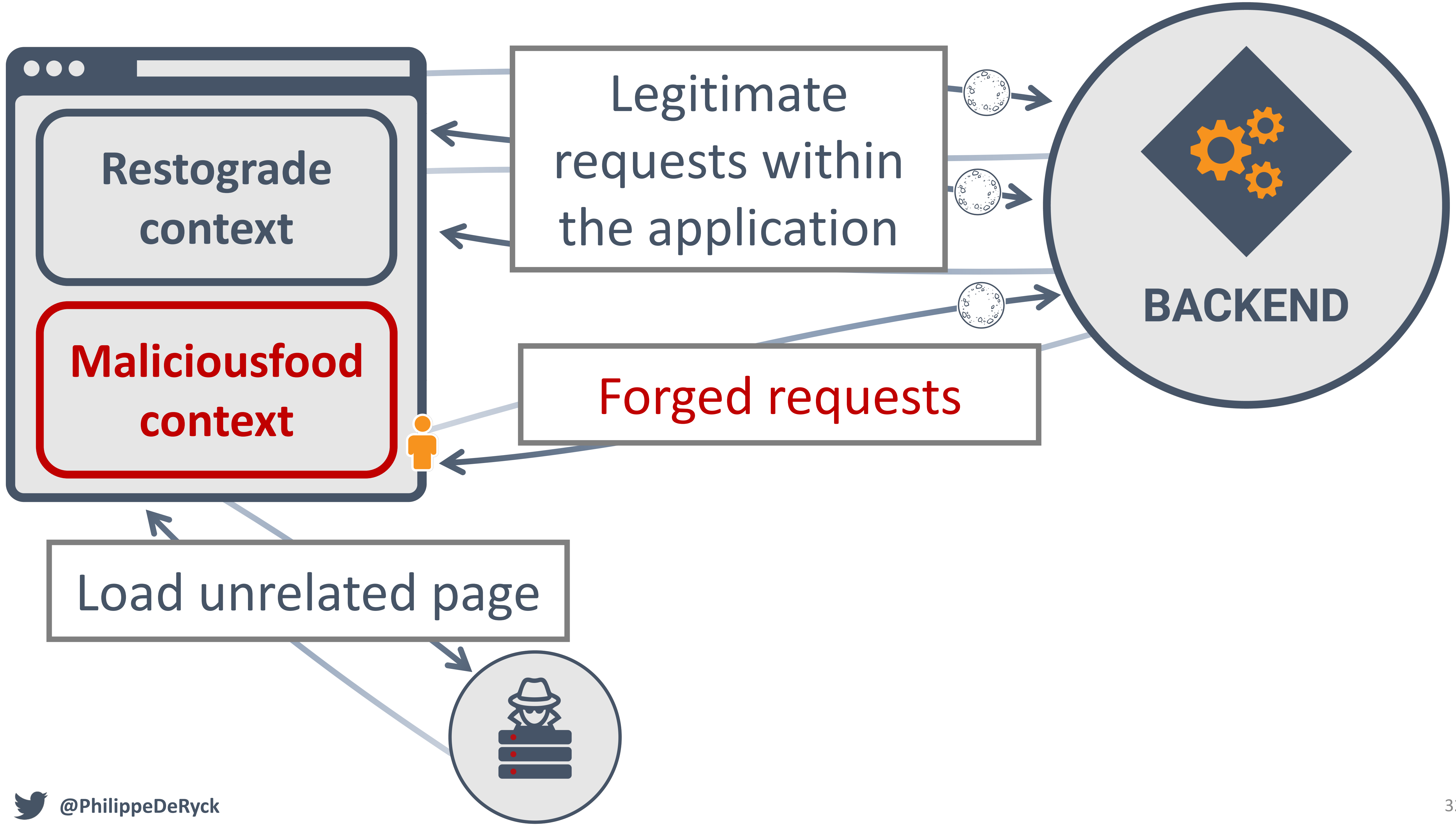


```
▾ Request Headers     view source
   Accept-Encoding: gzip, deflate, br
   Accept-Language: en-GB,en-US;q=0.9,en;q=0.8,nl;q=0.7,la;q=0.6
   Cache-Control: no-cache
   Connection: Upgrade
   Cookie: ID=12345678
   Host: restograde.com
   Origin: https://restograde.com
   Pragma: no-cache
   Sec-WebSocket-Extensions: permessage-deflate; client_max_window_bits
   Sec-WebSocket-Key: nYH7HTW3ooSDAveTiBpBGQ==
   Sec-WebSocket-Version: 13
   Upgrade: websocket
   User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_14_3) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/73.0.3683.103 Safari/53
   7.36
```

Cookies are always there

Cookies are present on every browser-initiated request, while the Authorization header is not.

If you depend on authorization for these features, consider using cookies.


Restograde context
Maliciousfood context
Legitimate requests within the application
Forged requests
BACKEND
Load unrelated page

CSRF

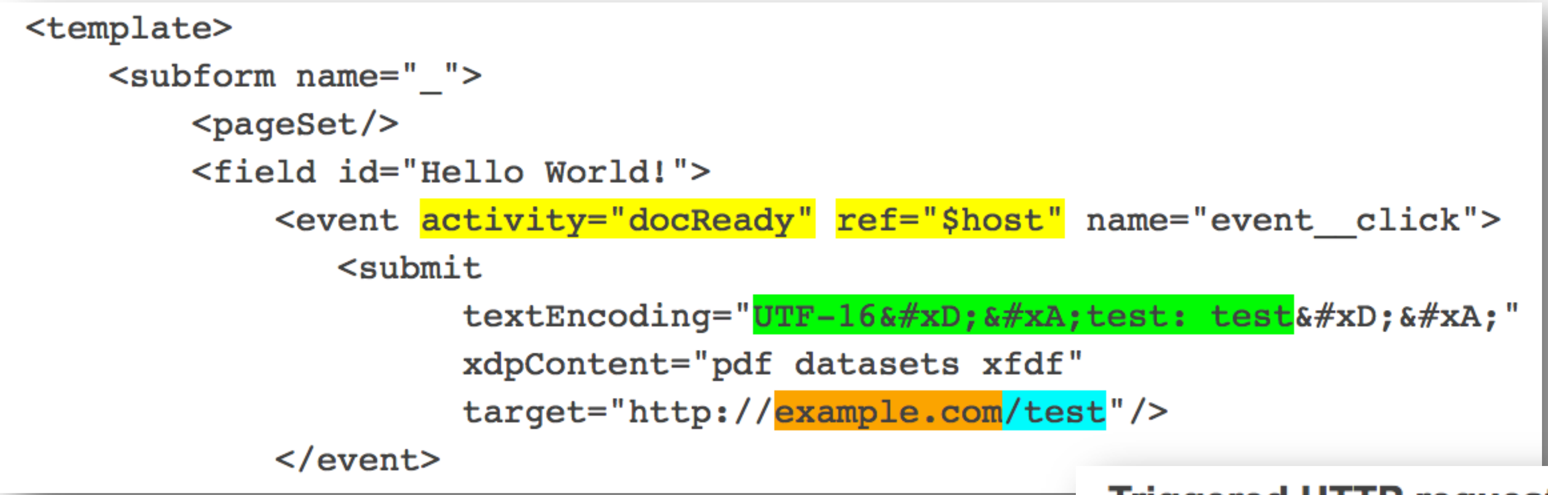
CSRF PROTECTION IN AN API WORLD

- If state-changing requests can only be sent from XHR, rely on CORS
  - Force the use of a non-form content type, so the requests fall within CORS
    - E.g., ***application/json***
  - For cross-origin requests, the browser always sends an ***Origin*** header

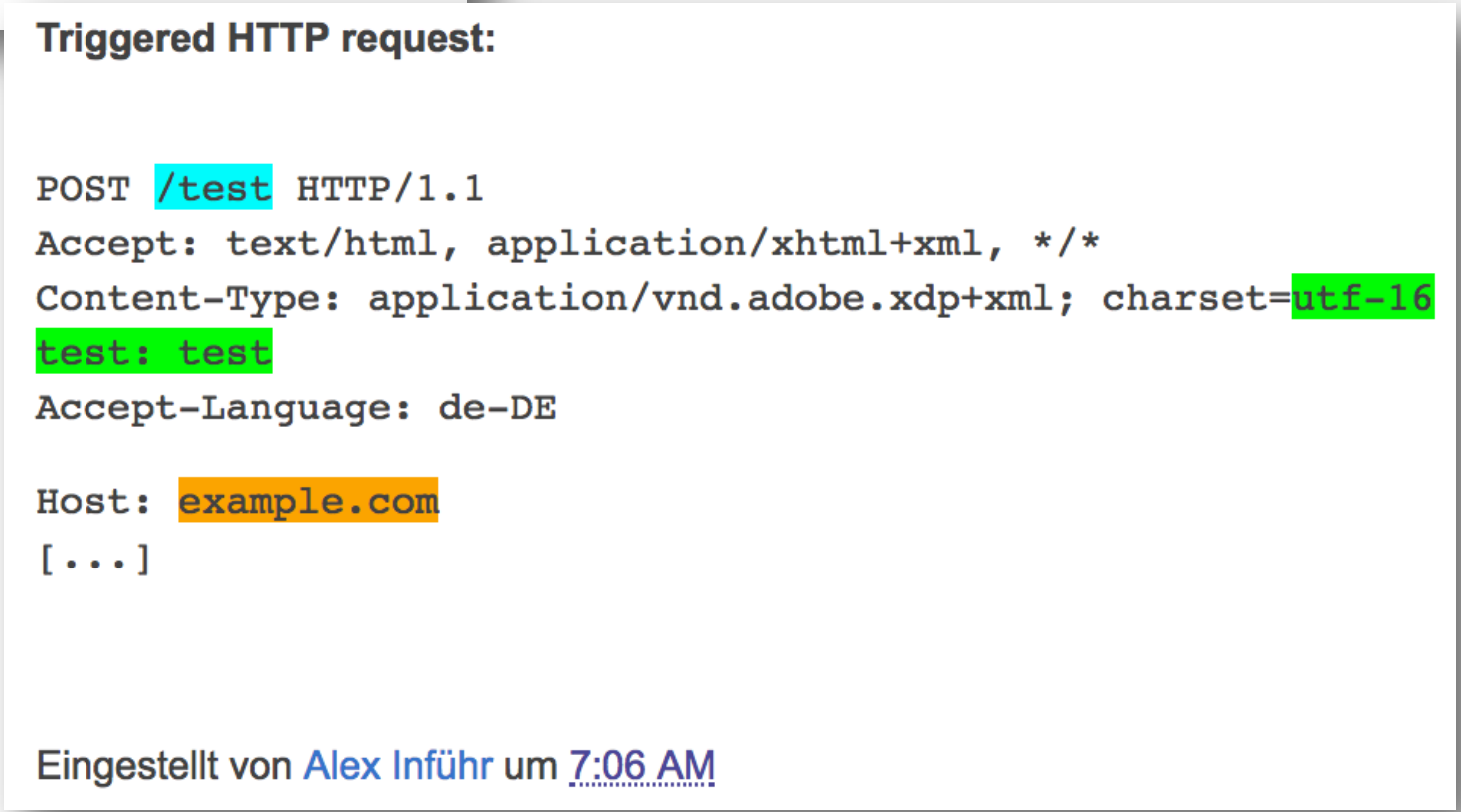
- If requests can be forged from HTML elements, use the double submit cookie
  - The server needs to set the cookie, and check incoming requests for the secret value
  - Angular supports this out of the box when frontend and backend run in the same domain

- API calls that can be forged from HTML should be avoided if possible
  - Double submit cookies only work well within a single domain
  - Plugins can forge requests across domains with arbitrary origin headers

EXOTIC WAYS TO FAKE THE *ORIGIN* HEADER WITH ADOBE PDF



```
<template>
    <subform name="_">
        <pageSet/>
        <field id="Hello World!">
            <event activity="docReady" ref="$host" name="event__click">
               <submit
                     textEncoding="UTF-16&#xD;&#xA;test: test&#xD;&#xA;"
                     xdpContent="pdf datasets xfdf"
                     target="http://example.com/test"/>
            </event>
```



Triggered HTTP request:

```
POST /test HTTP/1.1
Accept: text/html, application/xhtml+xml, */*
Content-Type: application/vnd.adobe.xdp+xml; charset=utf-16
test: test
Accept-Language: de-DE

Host: example.com
[...]
```

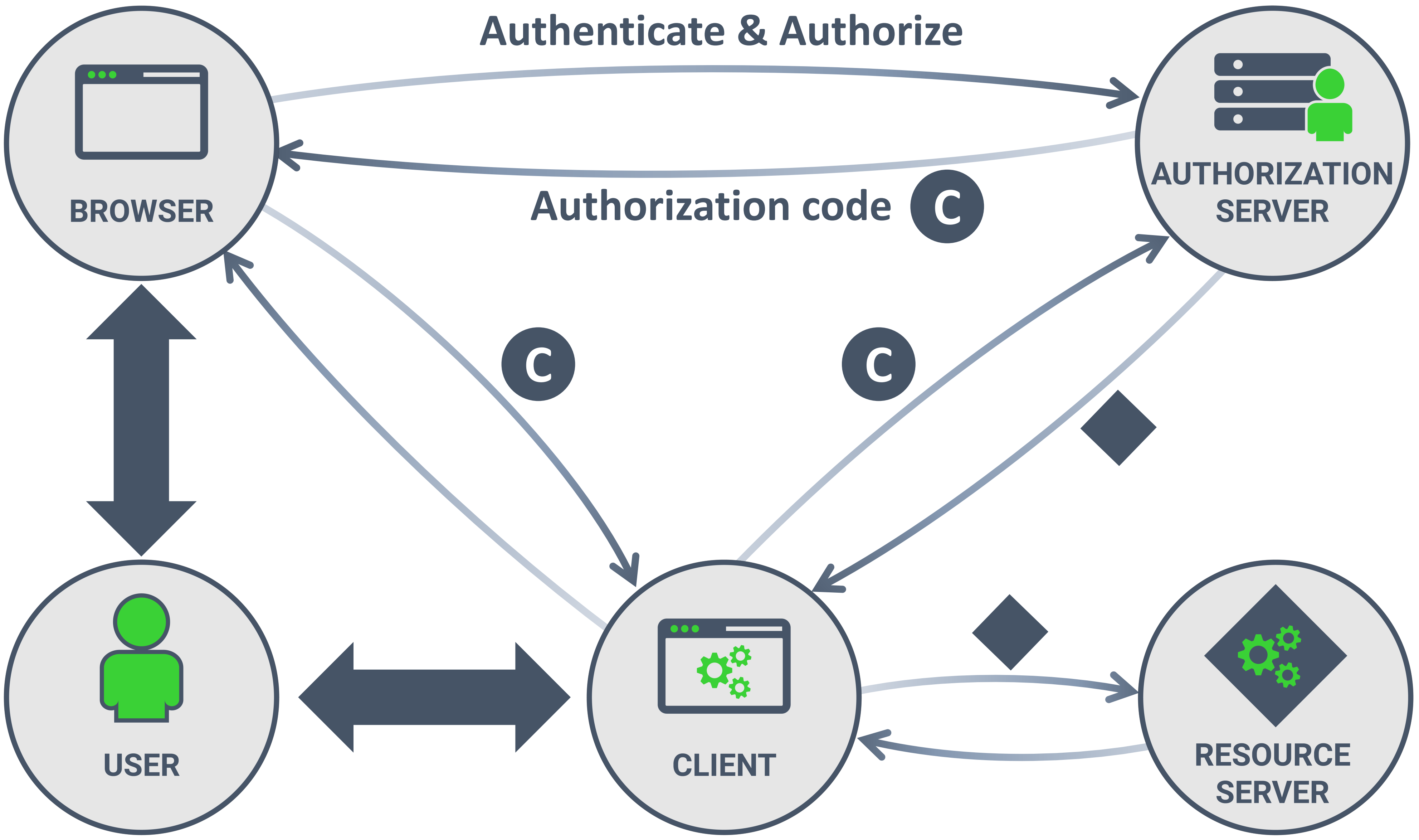
Eingestellt von Alex Inführ um 7:06 AM

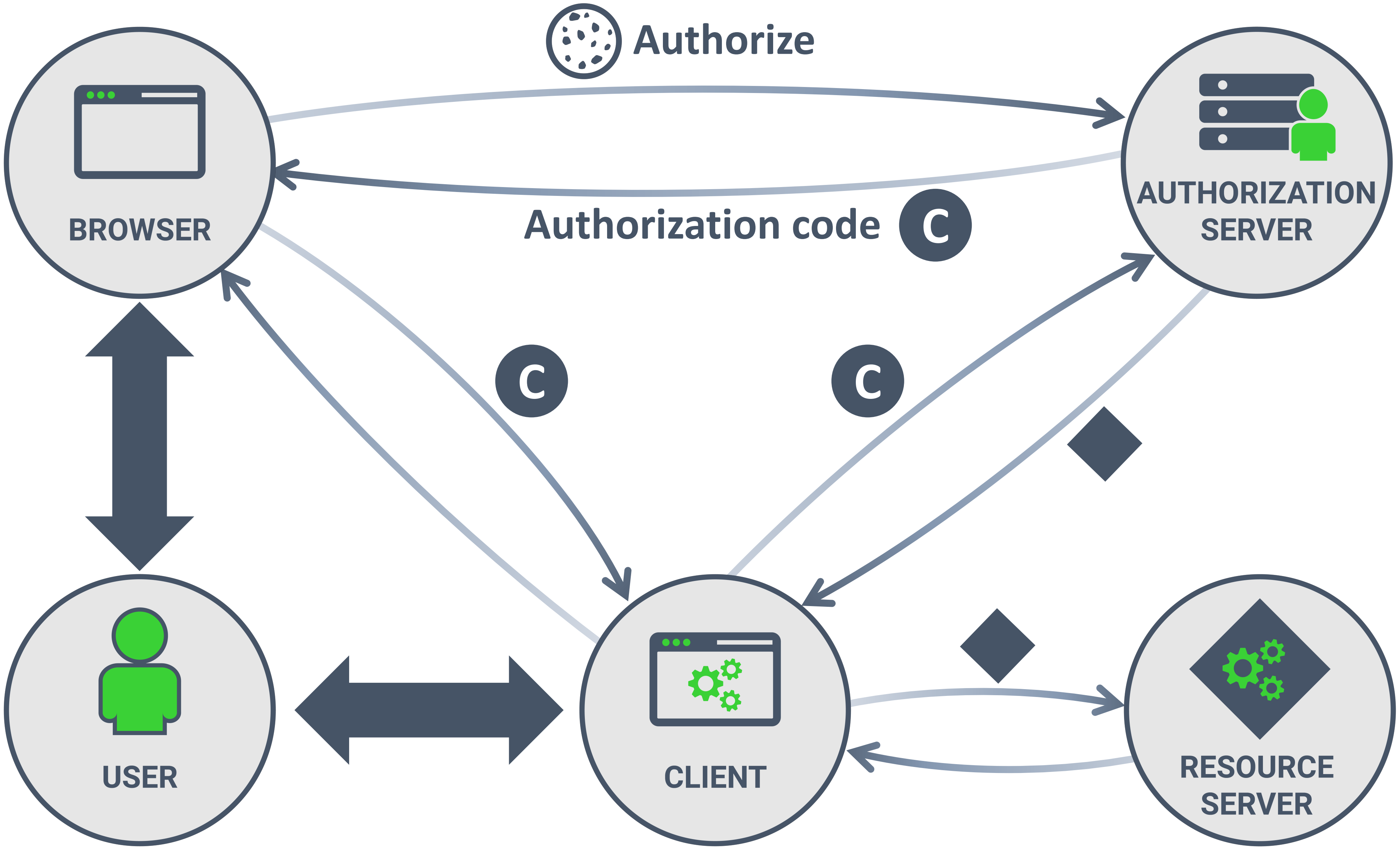
@PhilippeDeRyck

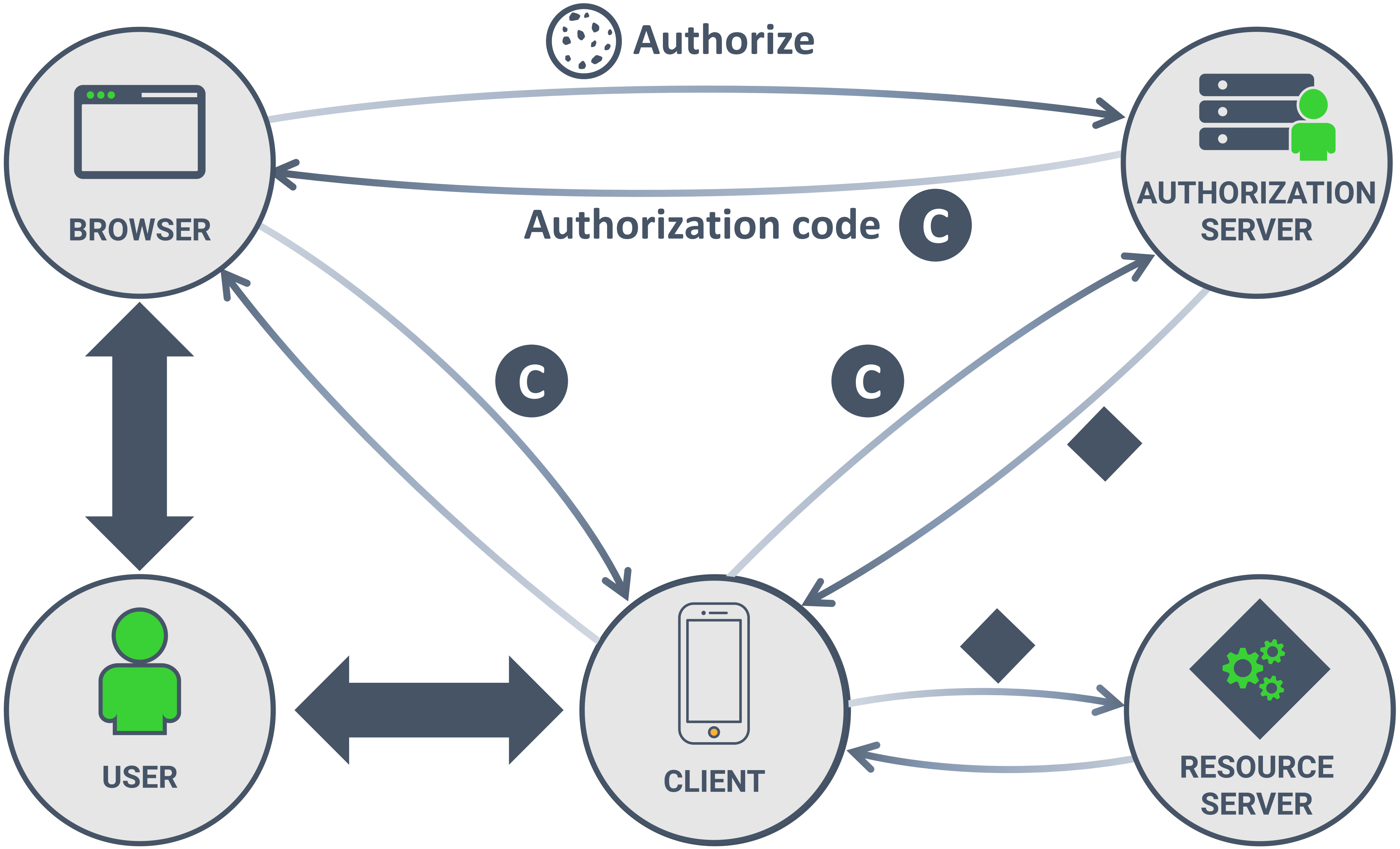
Cookies are always there

Cookies are present on every browser-initiated request, even when it originates from an attacker's page.

If you use cookies, implement CSRF protection.


Authenticate & Authorize
Authorization code C
BROWSER
AUTHORIZATION SERVER
C
C
USER
CLIENT
RESOURCE SERVER

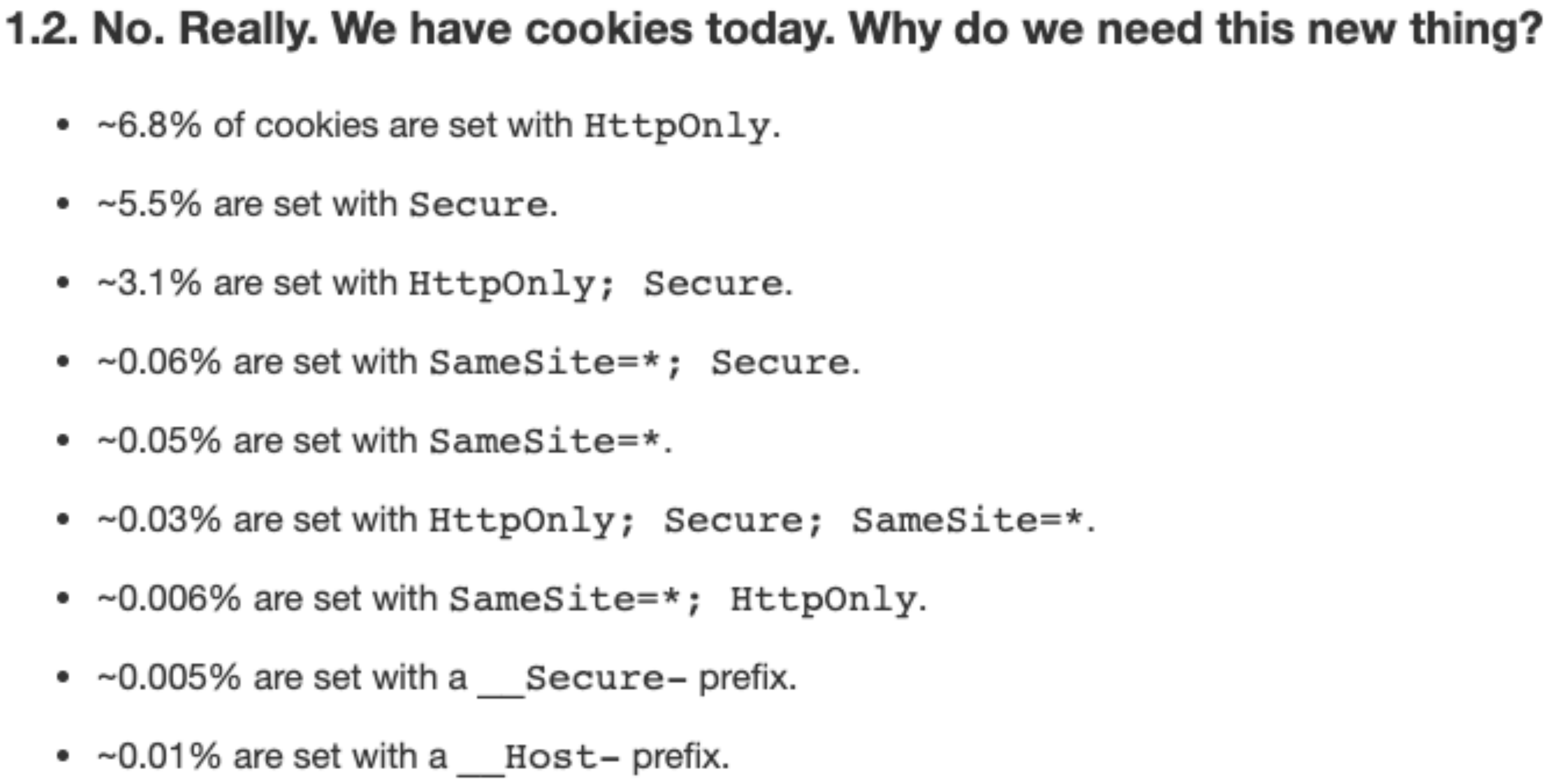

Authorize
Authorization code
C
C
C
BROWSER
AUTHORIZATION SERVER
USER
CLIENT
RESOURCE SERVER


Authorize
Authorization code
C
C
C
BROWSER
AUTHORIZATION SERVER
USER
CLIENT
RESOURCE SERVER

Cookies are inherent to the web (for now)

Cookies are inherent to the web and are the only reliable way to propagate state.

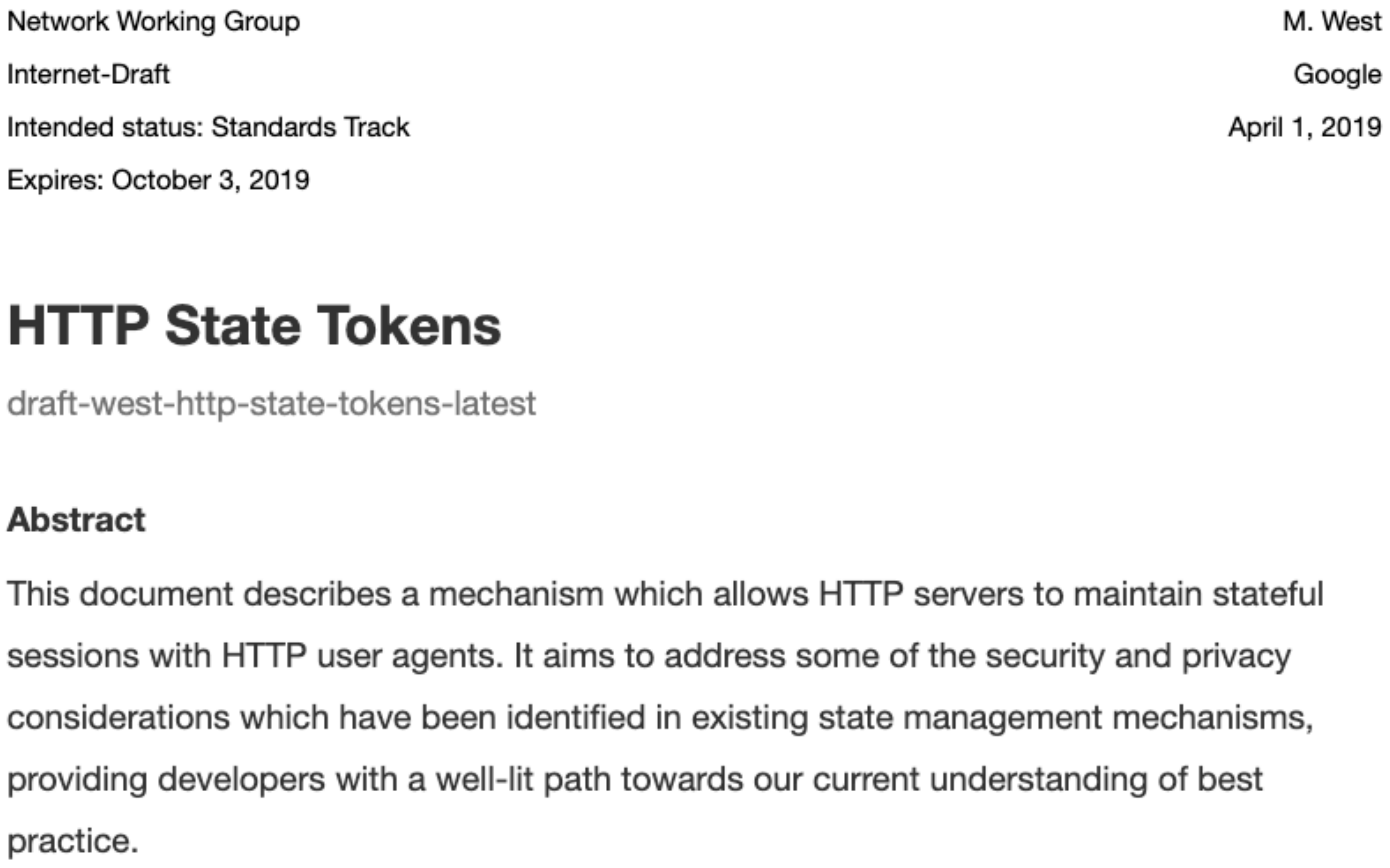
For now, secure your cookies, and lookout for the future.



1.2. No. Really. We have cookies today. Why do we need this new thing?

- ~6.8% of cookies are set with `HttpOnly`.
- ~5.5% are set with `Secure`.
- ~3.1% are set with `HttpOnly; Secure`.
- ~0.06% are set with `SameSite=*; Secure`.
- ~0.05% are set with `SameSite=*`.
- ~0.03% are set with `HttpOnly; Secure; SameSite=*`.
- ~0.006% are set with `SameSite=*; HttpOnly`.
- ~0.005% are set with a `__Secure-` prefix.
- ~0.01% are set with a `__Host-` prefix.

https://mikewest.github.io/http-state-tokens/draft-west-http-state-tokens.html



Network Working Group M. West

Internet-Draft Google

Intended status: Standards Track April 1, 2019

Expires: October 3, 2019

HTTP State Tokens

draft-west-http-state-tokens-latest

Abstract

This document describes a mechanism which allows HTTP servers to maintain stateful sessions with HTTP user agents. It aims to address some of the security and privacy considerations which have been identified in existing state management mechanisms, providing developers with a well-lit path towards our current understanding of best practice.

[1] https://mikewest.github.io/http-state-tokens/draft-west-http-state-tokens.html

Cookies are part of the web, whether you like it or not

They work well with a single domain, for all types of requests

They require flags and prefixes to lock 'em down

The *Authorization* header with bearer tokens is flexible

They work well, even in multi-domain scenarios

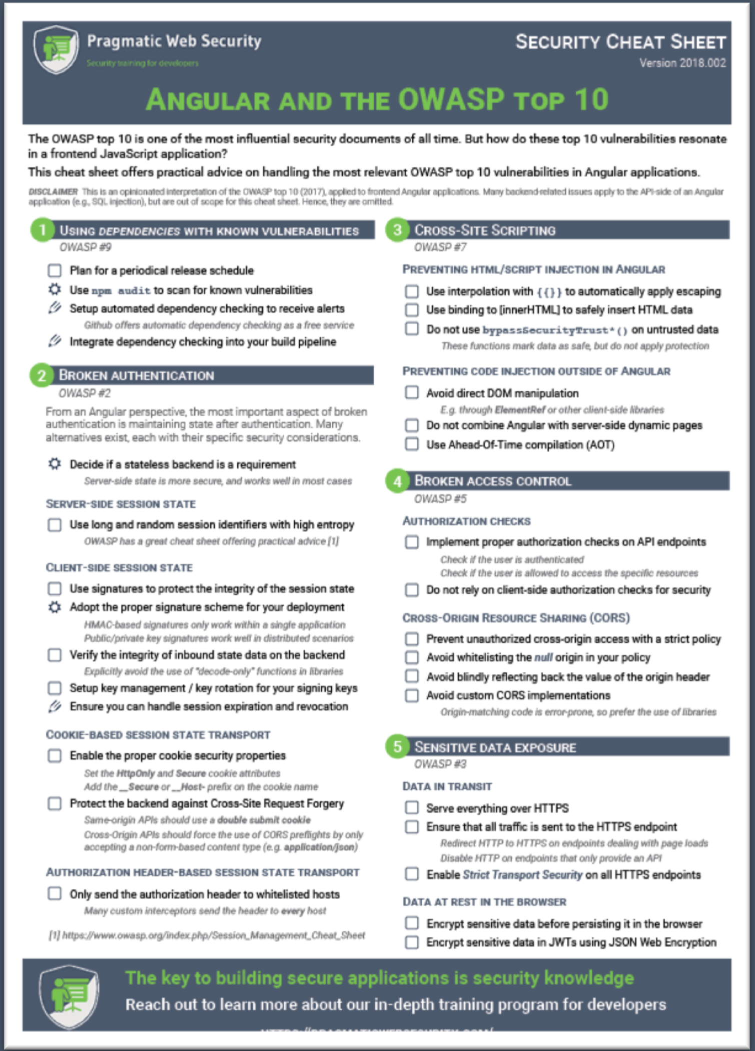
They require application code and are not always there

Proposal to replace cookies with *HTTP state tokens*

Client generates the value

Server offers additional security features (e.g., signing key)

Free security cheat sheets for modern applications



Pragmatic Web Security
Security training for developers

Security Cheat Sheet
Version 2018.002

Angular and the OWASP top 10

The OWASP top 10 is one of the most influential security documents of all time. But how do these top 10 vulnerabilities resonate in a frontend JavaScript application?

This cheat sheet offers practical advice on handling the most relevant OWASP top 10 vulnerabilities in Angular applications.

DISCLAIMER This is an opinionated interpretation of the OWASP top 10 (2017), applied to frontend Angular applications. Many backend-related issues apply to the API-side of an Angular application (e.g., SQL injection), but are out of scope for this cheat sheet. Hence, they are omitted.

1 Using *dependencies* with known vulnerabilities
OWASP #9

☐ Plan for a periodical release schedule
⚙ Use `npm audit` to scan for known vulnerabilities
✎ Setup automated dependency checking to receive alerts
Github offers automatic dependency checking as a free service
✎ Integrate dependency checking into your build pipeline

2 Broken authentication
OWASP #2

From an Angular perspective, the most important aspect of broken authentication is maintaining state after authentication. Many alternatives exist, each with their specific security considerations.

⚙ Decide if a stateless backend is a requirement
Server-side state is more secure, and works well in most cases

Server-side session state

☐ Use long and random session identifiers with high entropy
OWASP has a great cheat sheet offering practical advice [1]

Client-side session state

☐ Use signatures to protect the integrity of the session state
⚙ Adopt the proper signature scheme for your deployment
HMAC-based signatures only work within a single application
Public/private key signatures work well in distributed scenarios
☐ Verify the integrity of inbound state data on the backend
Explicitly avoid the use of "decode-only" functions in libraries
☐ Setup key management / key rotation for your signing keys
✎ Ensure you can handle session expiration and revocation

Cookie-based session state transport

☐ Enable the proper cookie security properties
*Set the **HttpOnly** and **Secure** cookie attributes*
*Add the **__Secure** or **__Host-** prefix on the cookie name*
☐ Protect the backend against Cross-Site Request Forgery
*Same-origin APIs should use a **double submit cookie***
*Cross-Origin APIs should force the use of CORS preflights by only accepting a non-form-based content type (e.g. **application/json**)*

Authorization header-based session state transport

☐ Only send the authorization header to whitelisted hosts
*Many custom interceptors send the header to **every** host*

[1] https://www.owasp.org/index.php/Session_Management_Cheat_Sheet

3 Cross-Site Scripting
OWASP #7

Preventing HTML/script injection in Angular

☐ Use interpolation with `{{}}` to automatically apply escaping
☐ Use binding to [innerHTML] to safely insert HTML data
☐ Do not use `bypassSecurityTrust*()` on untrusted data
These functions mark data as safe, but do not apply protection

Preventing code injection outside of Angular

☐ Avoid direct DOM manipulation
*E.g. through **ElementRef** or other client-side libraries*
☐ Do not combine Angular with server-side dynamic pages
☐ Use Ahead-Of-Time compilation (AOT)

4 Broken access control
OWASP #5

Authorization checks

☐ Implement proper authorization checks on API endpoints
Check if the user is authenticated
Check if the user is allowed to access the specific resources
☐ Do not rely on client-side authorization checks for security

Cross-Origin Resource Sharing (CORS)

☐ Prevent unauthorized cross-origin access with a strict policy
☐ Avoid whitelisting the *null* origin in your policy
☐ Avoid blindly reflecting back the value of the origin header
☐ Avoid custom CORS implementations
Origin-matching code is error-prone, so prefer the use of libraries

5 Sensitive data exposure
OWASP #3

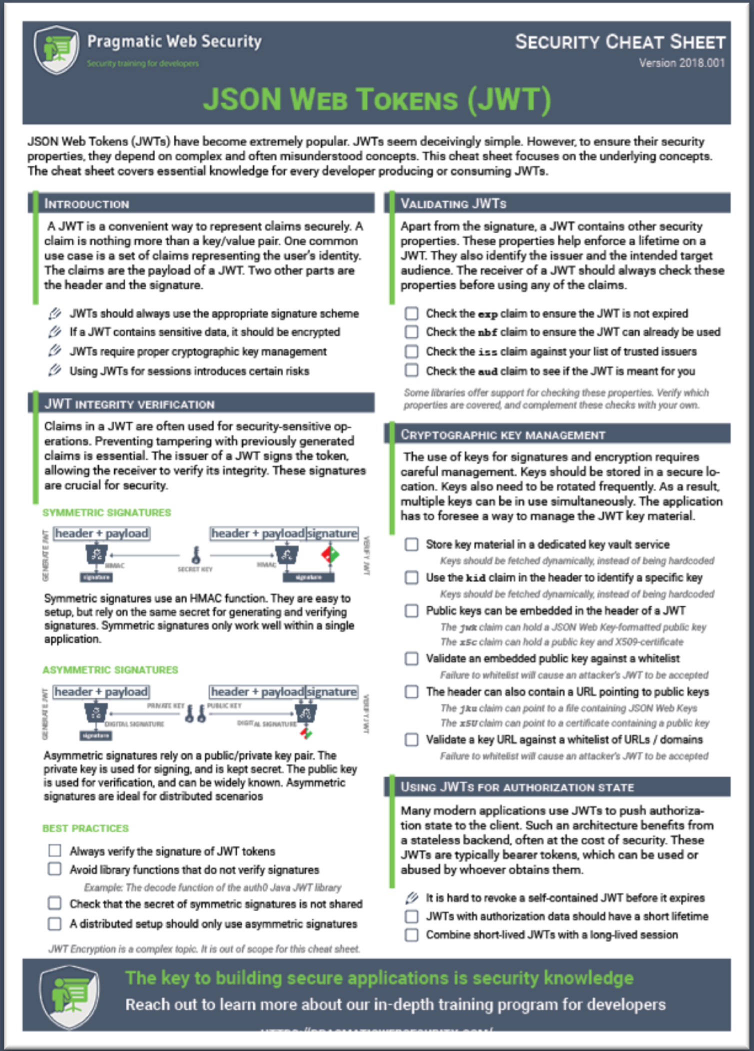
Data in transit

☐ Serve everything over HTTPS
☐ Ensure that all traffic is sent to the HTTPS endpoint
Redirect HTTP to HTTPS on endpoints dealing with page loads
Disable HTTP on endpoints that only provide an API
☐ Enable *Strict Transport Security* on all HTTPS endpoints

Data at rest in the browser

☐ Encrypt sensitive data before persisting it in the browser
☐ Encrypt sensitive data in JWTs using JSON Web Encryption

The key to building secure applications is security knowledge
Reach out to learn more about our in-depth training program for developers



Pragmatic Web Security
Security training for developers

Security Cheat Sheet
Version 2018.001

JSON Web Tokens (JWT)

JSON Web Tokens (JWTs) have become extremely popular. JWTs seem deceivingly simple. However, to ensure their security properties, they depend on complex and often misunderstood concepts. This cheat sheet focuses on the underlying concepts. The cheat sheet covers essential knowledge for every developer producing or consuming JWTs.

Introduction

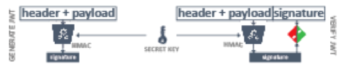
A JWT is a convenient way to represent claims securely. A claim is nothing more than a key/value pair. One common use case is a set of claims representing the user's identity. The claims are the payload of a JWT. Two other parts are the header and the signature.

✎ JWTs should always use the appropriate signature scheme
✎ If a JWT contains sensitive data, it should be encrypted
✎ JWTs require proper cryptographic key management
✎ Using JWTs for sessions introduces certain risks

JWT integrity verification

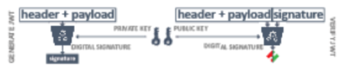
Claims in a JWT are often used for security-sensitive operations. Preventing tampering with previously generated claims is essential. The issuer of a JWT signs the token, allowing the receiver to verify its integrity. These signatures are crucial for security.

Symmetric signatures



Symmetric signatures use an HMAC function. They are easy to setup, but rely on the same secret for generating and verifying signatures. Symmetric signatures only work well within a single application.

Asymmetric signatures



Asymmetric signatures rely on a public/private key pair. The private key is used for signing, and is kept secret. The public key is used for verification, and can be widely known. Asymmetric signatures are ideal for distributed scenarios

Best practices

☐ Always verify the signature of JWT tokens
☐ Avoid library functions that do not verify signatures
Example: The decode function of the auth0 Java JWT library
☐ Check that the secret of symmetric signatures is not shared
☐ A distributed setup should only use asymmetric signatures

JWT Encryption is a complex topic. It is out of scope for this cheat sheet.

Validating JWTs

Apart from the signature, a JWT contains other security properties. These properties help enforce a lifetime on a JWT. They also identify the issuer and the intended target audience. The receiver of a JWT should always check these properties before using any of the claims.

☐ Check the `exp` claim to ensure the JWT is not expired
☐ Check the `nbf` claim to ensure the JWT can already be used
☐ Check the `iss` claim against your list of trusted issuers
☐ Check the `aud` claim to see if the JWT is meant for you

Some libraries offer support for checking these properties. Verify which properties are covered, and complement these checks with your own.

Cryptographic key management

The use of keys for signatures and encryption requires careful management. Keys should be stored in a secure location. Keys also need to be rotated frequently. As a result, multiple keys can be in use simultaneously. The application has to foresee a way to manage the JWT key material.

☐ Store key material in a dedicated key vault service
Keys should be fetched dynamically, instead of being hardcoded
☐ Use the `kid` claim in the header to identify a specific key
Keys should be fetched dynamically, instead of being hardcoded
☐ Public keys can be embedded in the header of a JWT
The `jwk` claim can hold a JSON Web Key-formatted public key
The `x5c` claim can hold a public key and X509-certificate
☐ Validate an embedded public key against a whitelist
Failure to whitelist will cause an attacker's JWT to be accepted
☐ The header can also contain a URL pointing to public keys
The `jku` claim can point to a file containing JSON Web Keys
The `x5u` claim can point to a certificate containing a public key
☐ Validate a key URL against a whitelist of URLs / domains
Failure to whitelist will cause an attacker's JWT to be accepted

Using JWTs for authorization state

Many modern applications use JWTs to push authorization state to the client. Such an architecture benefits from a stateless backend, often at the cost of security. These JWTs are typically bearer tokens, which can be used or abused by whoever obtains them.

✎ It is hard to revoke a self-contained JWT before it expires
☐ JWTs with authorization data should have a short lifetime
☐ Combine short-lived JWTs with a long-lived session

The key to building secure applications is security knowledge
Reach out to learn more about our in-depth training program for developers

https://cheatsheets.pragmaticwebsecurity.com/

Pragmatic Web Security

Security training for developers

/in/PhilippeDeRyck

@PhilippeDeRyck

philippe@pragmaticwebsecurity.com