PDF of the slides: https://bit.ly/2Am8vl3

Introduction to OAuth 2.0 and OIDC

with Dr. Philippe De Ryck

Dr. Philippe De Ryck

- Ph.D. in web security
- Founder of Pragmatic Web Security
- Course curator of the SecAppDev course
  (https://secappdev.org)

Pragmatic Web Security

Providing practical security knowledge to developers and architects

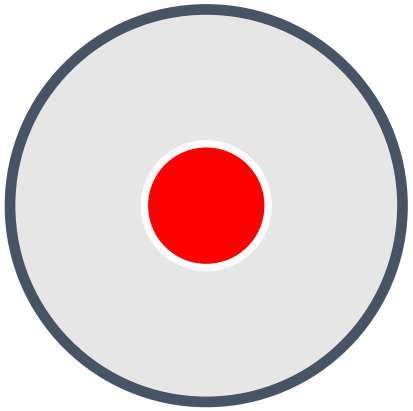
In-depth training courses and security advisory services

Security cheat sheets and detailed articles

@PhilippeDeRyck
https://PragmaticWebSecurity.com

PRACTICALITIES

Ask questions through the Zoom Q&A



Recording will be published online

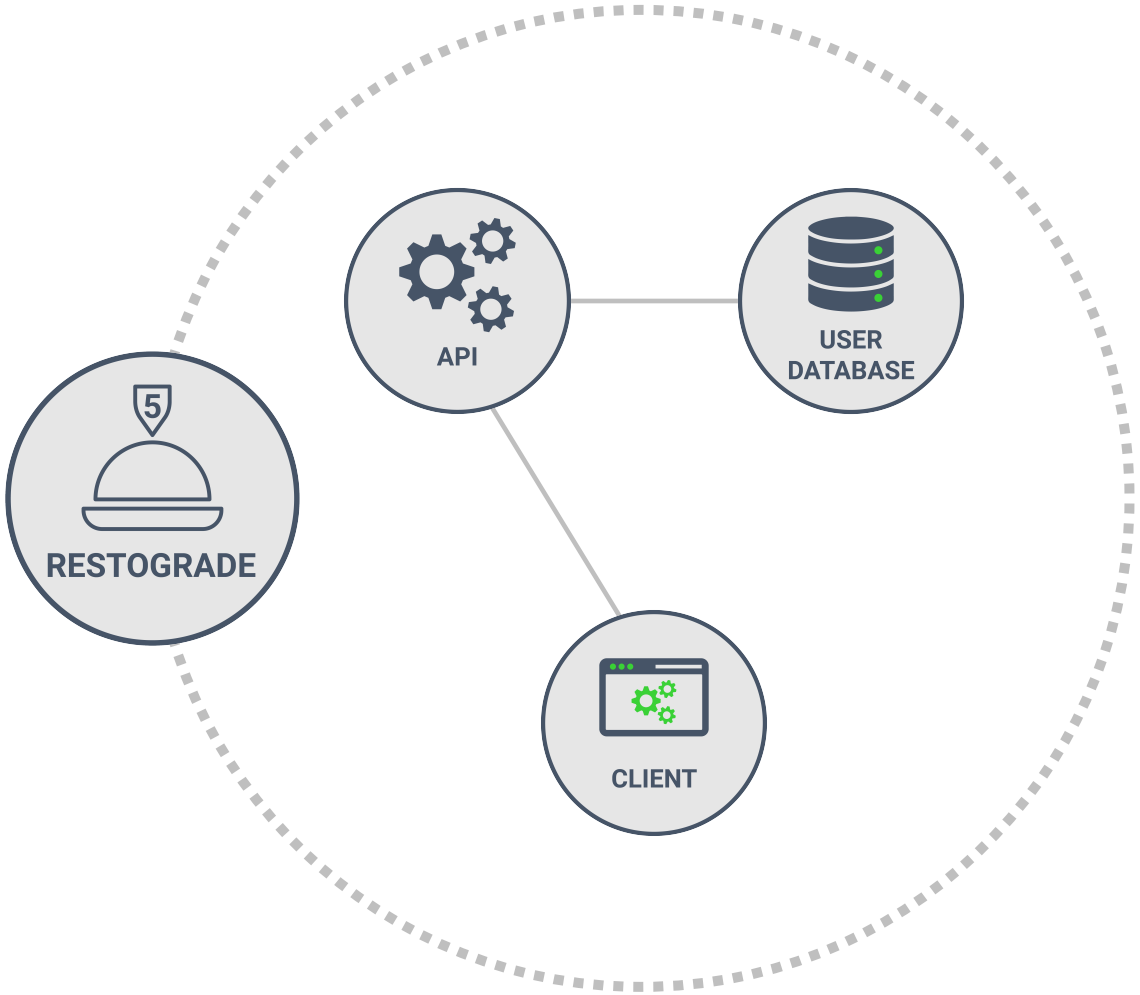
Use the Zoom chat if there is a problem

Introduction

RESTOGRADE

A RESTAURANT
REVIEW APPLICATION

FRONTEND AND
MOBILE APPLICATIONS

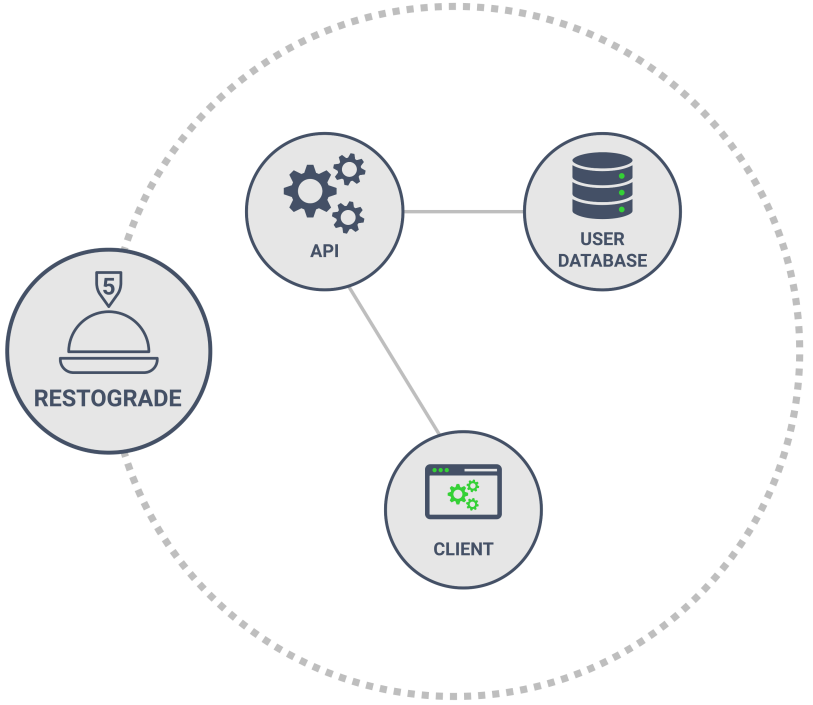

API
USER
DATABASE
RESTOGRADE
CLIENT

YOU DON'T NEED OAUTH 2.0 AND OPENID CONNECT

- Many applications are small and uncomplicated
  - Frontend and API share the same level of trust
  - There is no need for insane scalability

- There is nothing wrong with cookie-based sessions
  - Cookies are supported by every browser
  - Modern server frameworks all still support cookies
  - Strict cookie security settings address common issues
  - ***Sticky sessions*** or ***session replication*** support scalability

- Keep it simple when you can!



RESTOGRADE

A RESTAURANT REVIEW APPLICATION

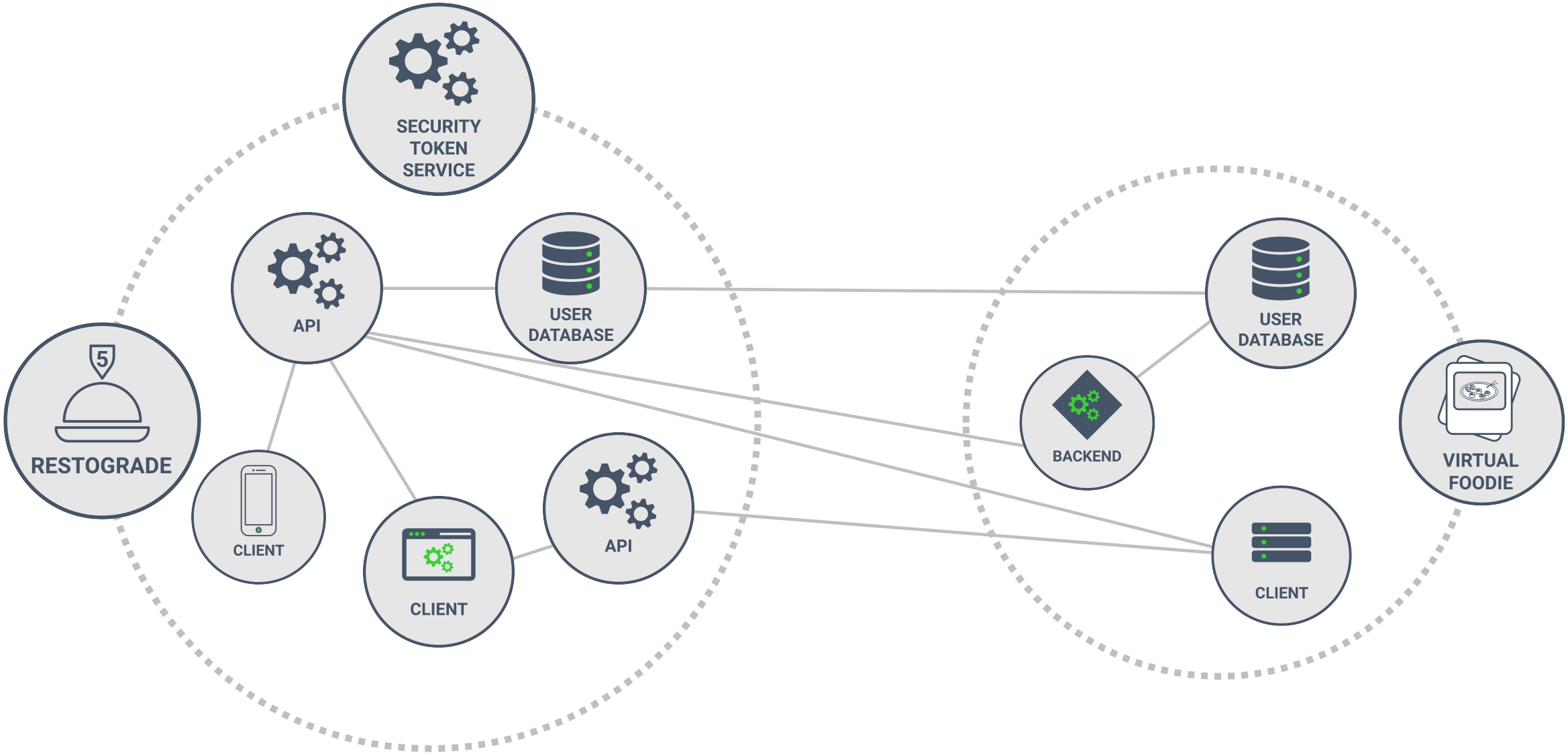
FRONTEND AND MOBILE APPLICATIONS

APIS TO EMPOWER PARTNER SITES

VIRTUAL FOODIE

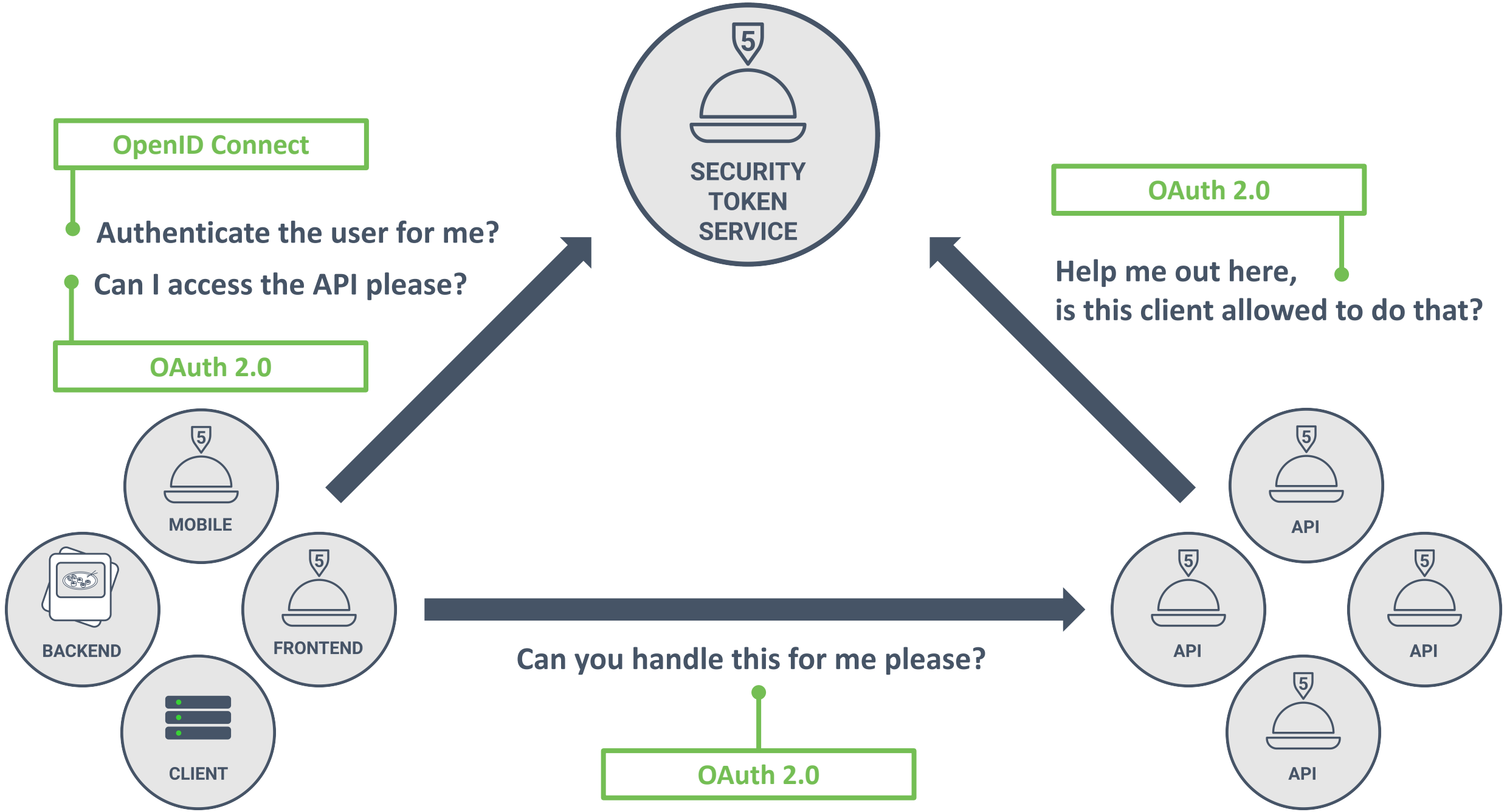
An application for foodies

Supports integration with Restograde

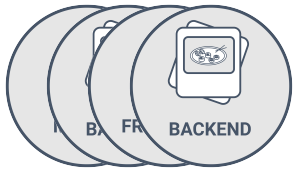

SECURITY TOKEN SERVICE
API
USER DATABASE
RESTOGRADE
CLIENT
CLIENT
API
USER DATABASE
BACKEND
VIRTUAL FOODIE
CLIENT

OAuth 2.0 offers an authorization framework to support complex applications


LIGHT ST.
STOPPING ANY TIME
FDC
VALET PARKING
VALET
Maryland
4DJ2760


OpenID Connect
Authenticate the user for me?
Can I access the API please?
OAuth 2.0
SECURITY TOKEN SERVICE
OAuth 2.0
Help me out here,
is this client allowed to do that?
MOBILE
BACKEND
FRONTEND
CLIENT
Can you handle this for me please?
OAuth 2.0
API
API
API
API

TERMINOLOGY

	This course	OAuth 2.0	OpenID Connect
	**User**	Resource Owner	End-User
	**API**	Resource Server	
	**Security Token Service (STS)**	Authorization Server	OpenID Provider
	**Client**	Client	Relying Party

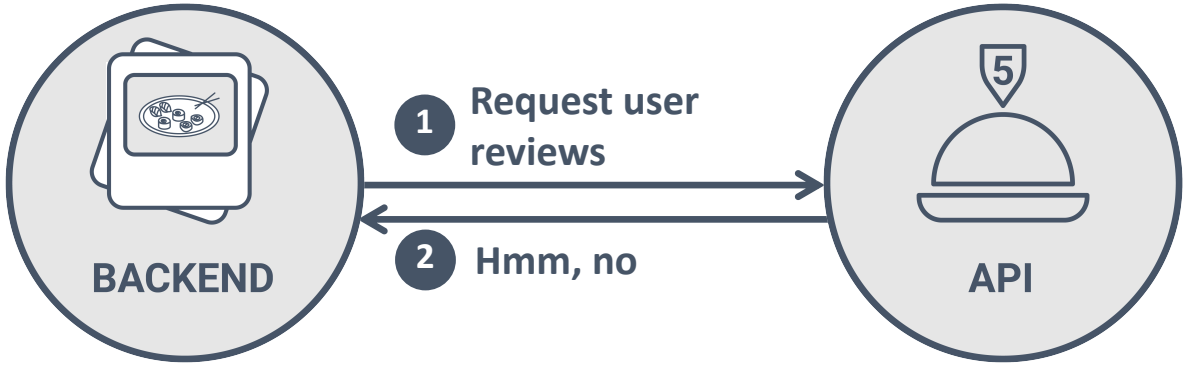
The value of OAuth 2.0 and OpenID Connect

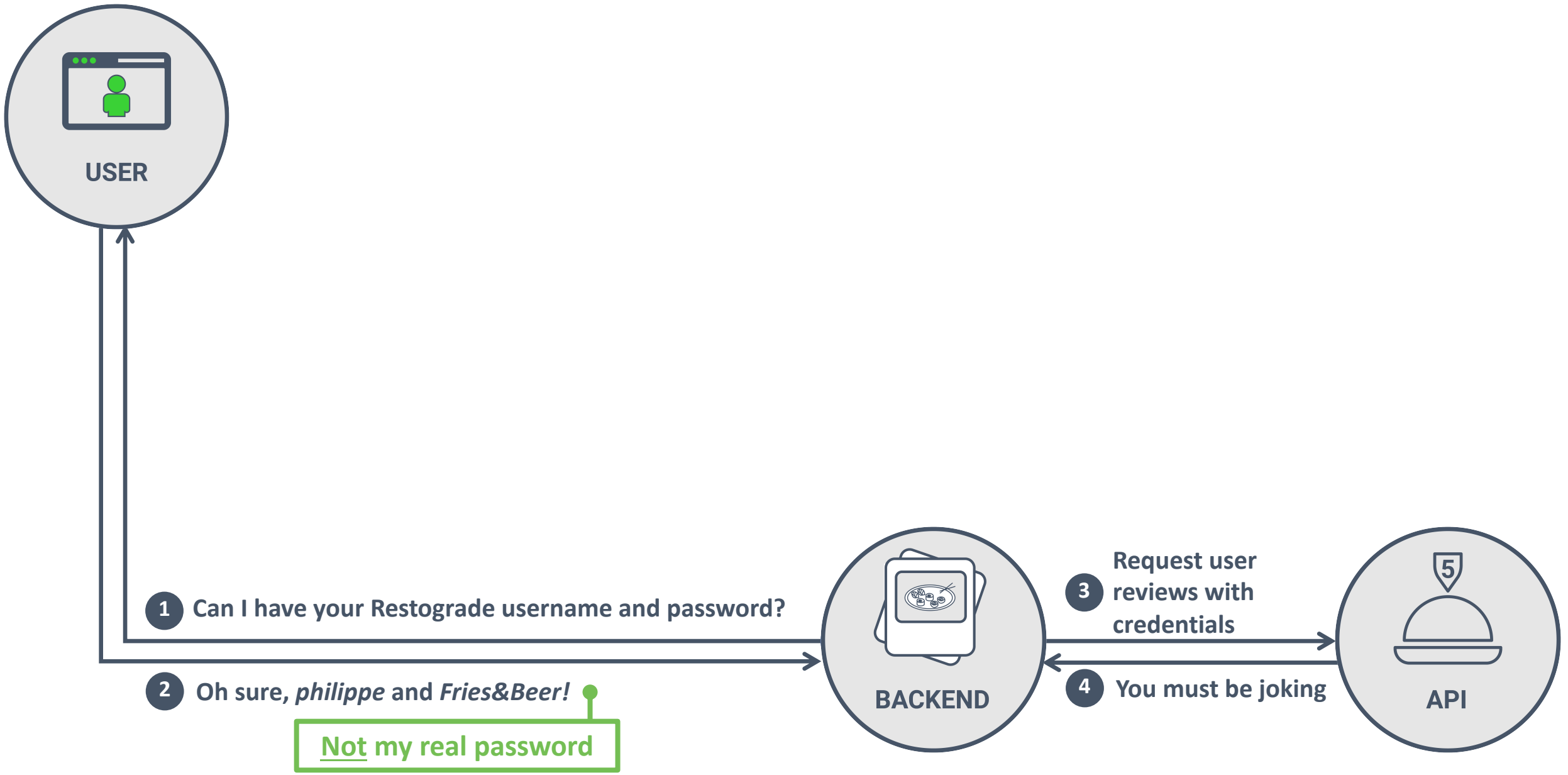
- Complex application architectures will benefit from OAuth 2.0 and OIDC
  - A centralized Security Token Service governs all interactions
  - Users, clients, and APIs all trust this Security Token Service
- The Security Token Service is the glue keeping it all together
  - The STS offers centralized user authentication with OIDC
  - The STS enables uniform authorization between clients and APIs with OAuth 2.0
- How do you know if you need OAuth 2.0 and OIDC?
  - If you want to implement a Single Sign-On system
  - If you have an elaborate ecosystem of clients and APIs that need to be decoupled
  - If your application has extensive scalability needs
  - If your application is likely to hit these needs in the near future

Session outline

- The conceptual idea of OAuth 2.0 and OIDC
- Backend web clients
- Mobile clients
- Frontend web clients
- Additional OAuth 2.0 flows
- Wrapping up

The conceptual idea of OAuth 2.0 and OIDC

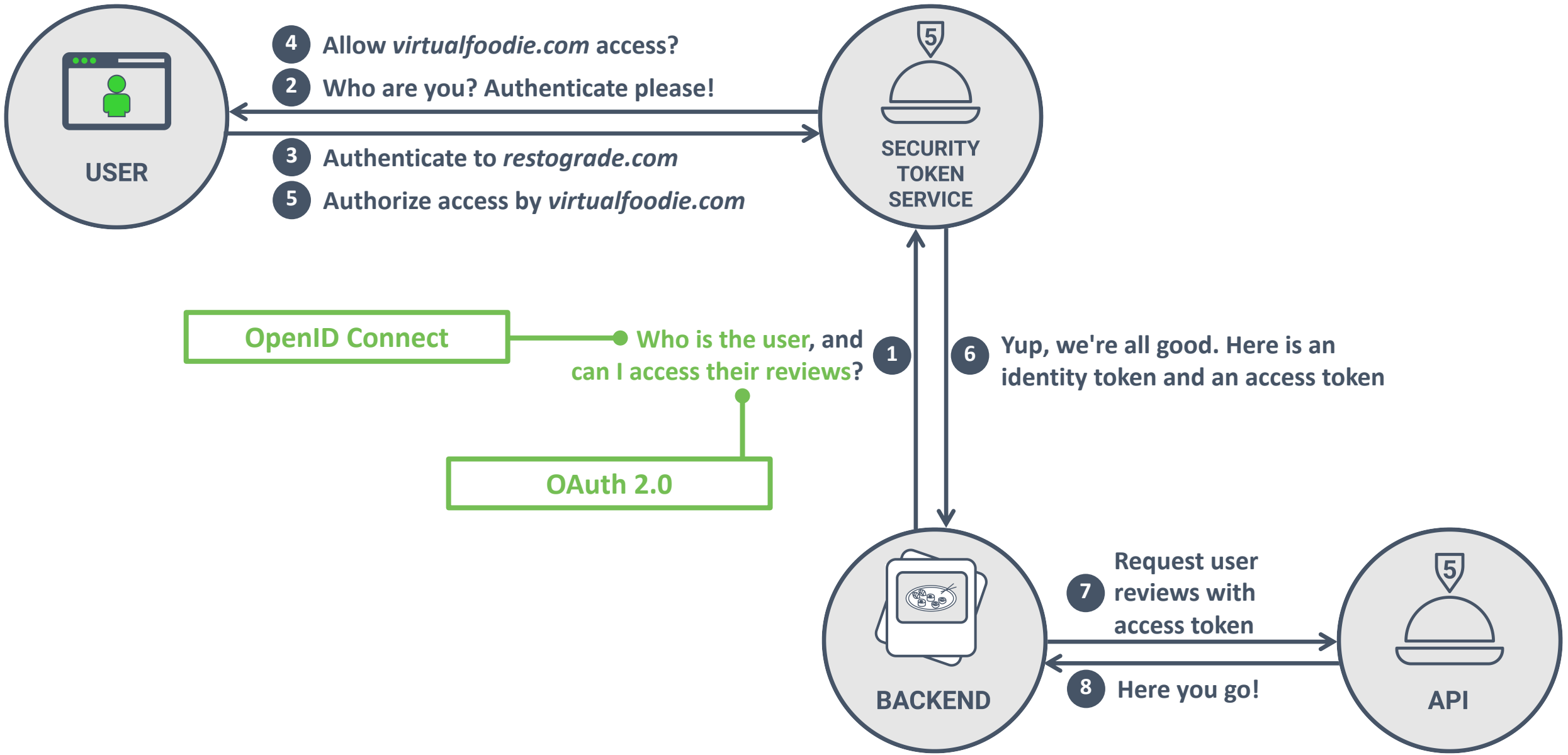

BACKEND
1
Request user reviews
2
Hmm, no
5
API


USER
1 Can I have your Restograde username and password?
2 Oh sure, philippe and Fries&Beer!
Not my real password
BACKEND
3 Request user reviews with credentials
4 You must be joking
5
API

Facebook under fire after firm is caught demanding new users hand over their email passwords in exchange for harvesting their contacts without their consent

- **Some users who attempt to sign up are required to give their email password**
- **The firm also appears to be harvesting their contacts after they provide the info**
- **Facebook now says it will no longer ask users to provide their email passwords**
- **Security experts called the move 'sleazy' and compared it to a phishing attack**

By ANNIE PALMER FOR DAILYMAIL.COM
PUBLISHED: 17:11 BST, 3 April 2019 | **UPDATED:** 17:13 BST, 3 April 2019

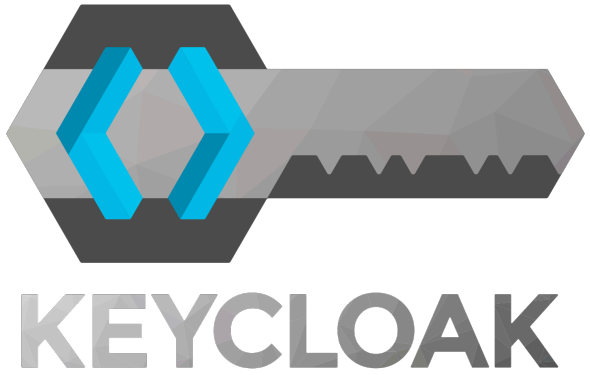

USER
4 Allow *virtualfoodie.com* access?
2 Who are you? Authenticate please!
3 Authenticate to *restograde.com*
5 Authorize access by *virtualfoodie.com*
SECURITY TOKEN SERVICE
OpenID Connect
Who is the user, and can I access their reviews?
OAuth 2.0
1
6 Yup, we're all good. Here is an identity token and an access token
BACKEND
7 Request user reviews with access token
8 Here you go!
API

OAuth 2.0 and OpenID Connect

- OAuth 2.0 enables a user to delegate access to a client application
  - OAuth 2.0 itself does not offer authentication or authorization
  - OAuth 2.0 specs only describe how to handle delegation and handle tokens
  - OAuth 2.0 has built-in mechanisms to support permissions and consent
- OpenID Connect allows a client to offload authentication to an identity provider
  - OIDC is a specific protocol that handles the delegation of authentication
  - The result of an OIDC flow is information about the user's authentication
    - Information contains a timestamp, a user ID, optional profile information, …
- Neither of the specifications defines how the user authenticates to the STS
  - The STS is free to decide how authentication should happen

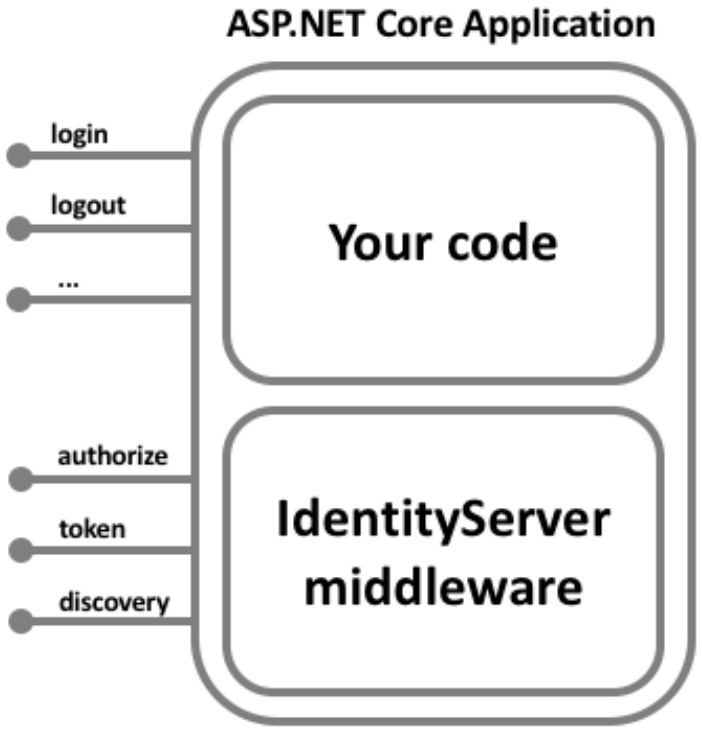
When using OAuth 2.0 & OIDC, a user should only authenticate to the Security Token Service, never anywhere else.

Deployment models for a Security Token Service



A standalone service handling OAuth 2.0, OIDC, and user management

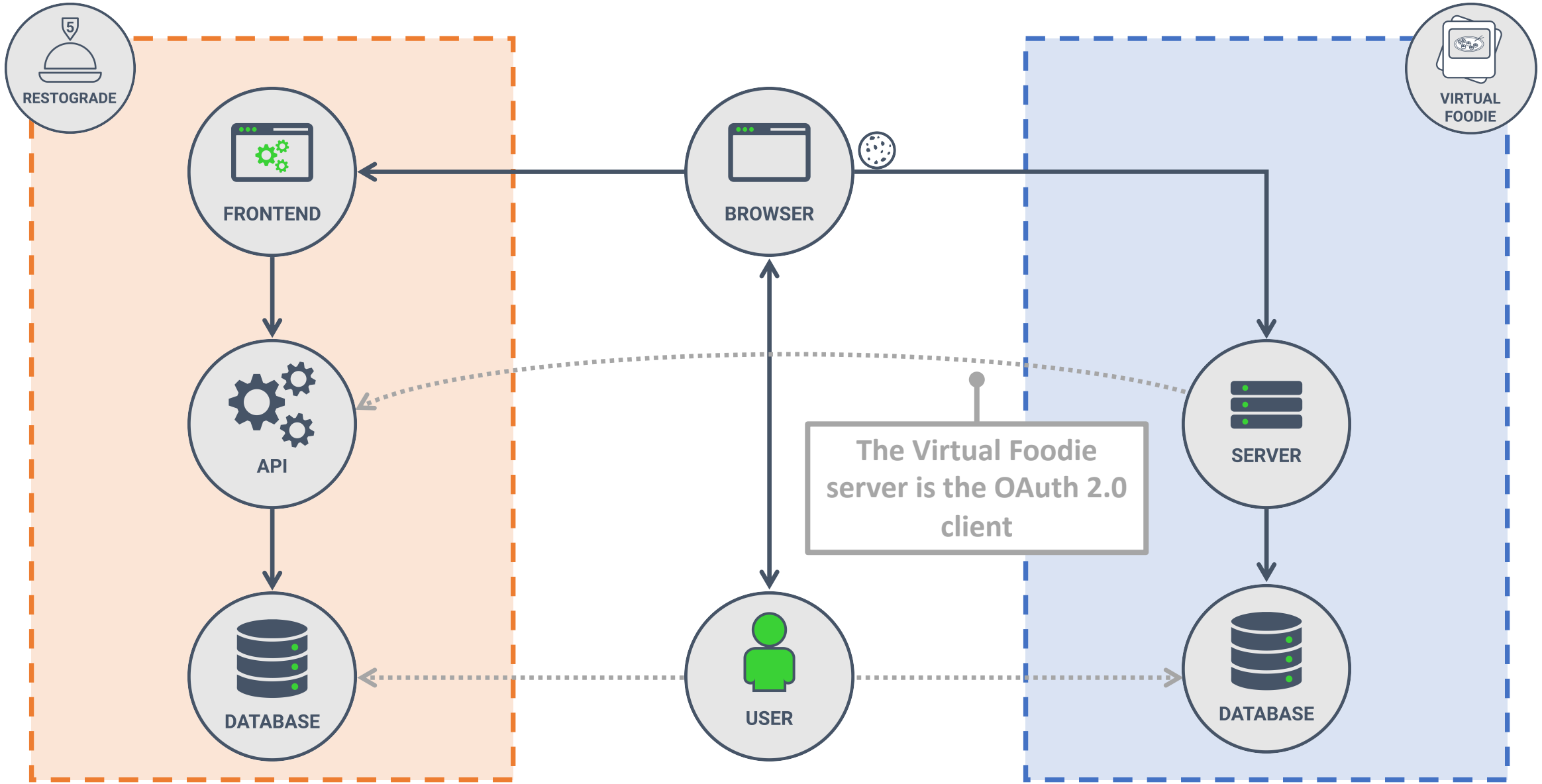
Offload all responsibilities to "Identity as a Service"



Augment your user management service with OAuth 2.0 & OIDC

Backend web clients

A BACKEND CLIENT SCENARIO

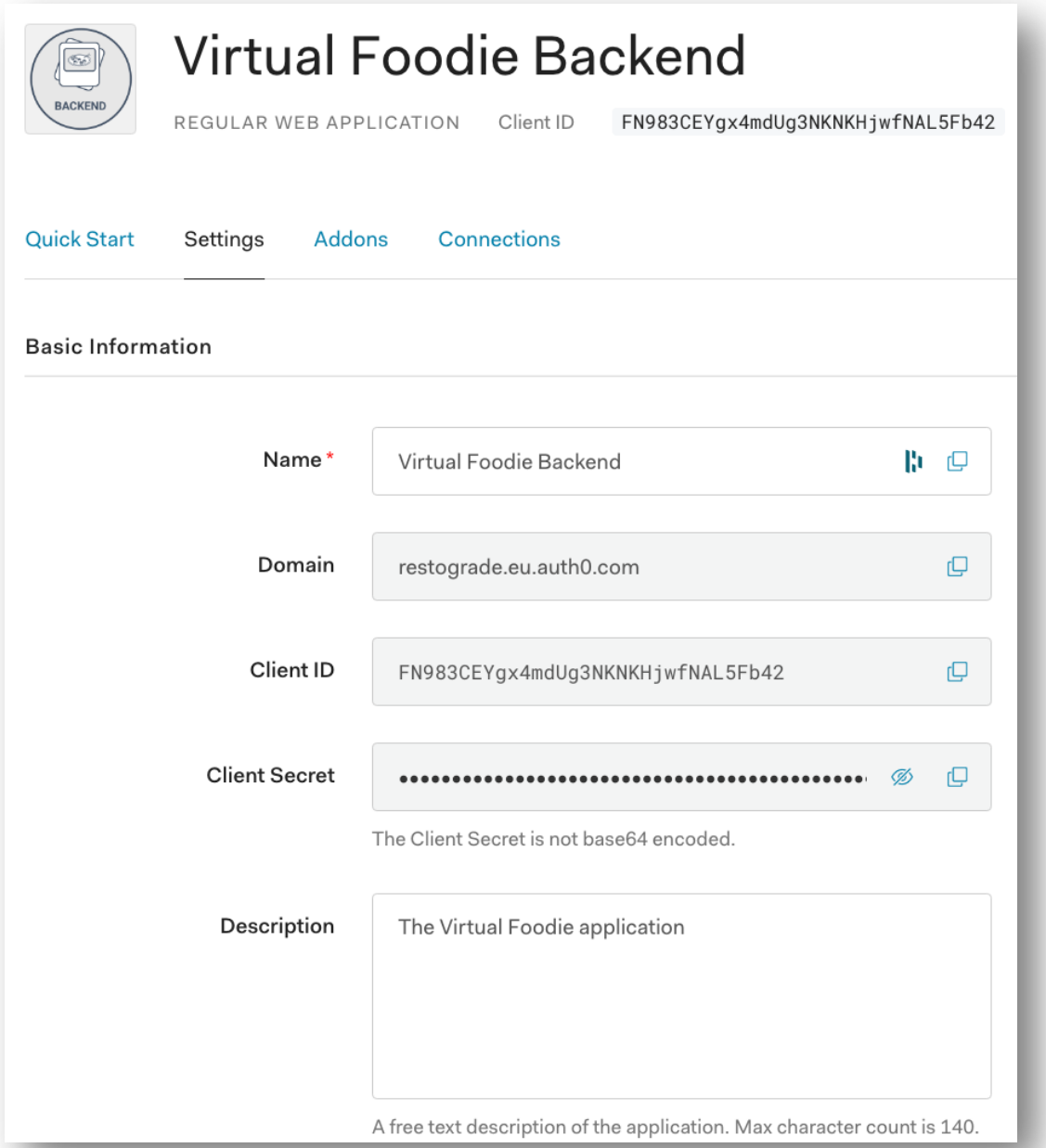


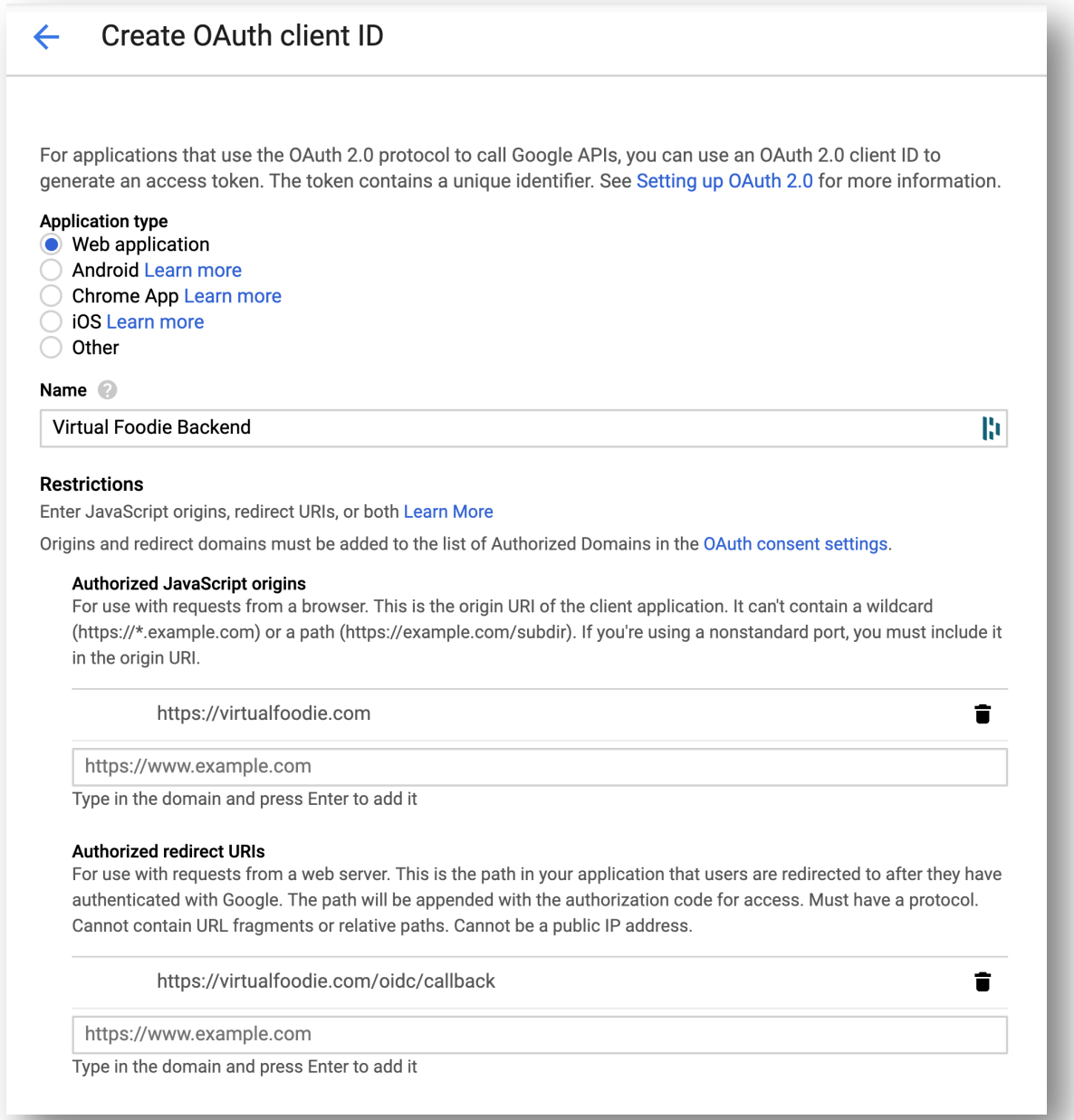
IT ALL STARTS WITH CLIENT REGISTRATION

- Client registration is essential to secure OAuth 2.0 flows
  - Configure the available flows for a particular client
  - Setup client permissions and advanced authorization mechanisms
  - Register the URIs that are used to send sensitive information to
    - Essential to prevent an attacker from running a flow with a malicious redirect URI
    - Oauth 2.1 only allows exact matching and prohibits the use of wildcards in redirect URIs

- Confidential clients need to register to setup client credentials
  - Simplest form is a shared secret string that acts as a password
  - Advanced authentication mechanisms rely on cryptographic keys
    - E.g., mTLS or JWT-based authentication by proving possession of a private key

CLIENT REGISTRATION



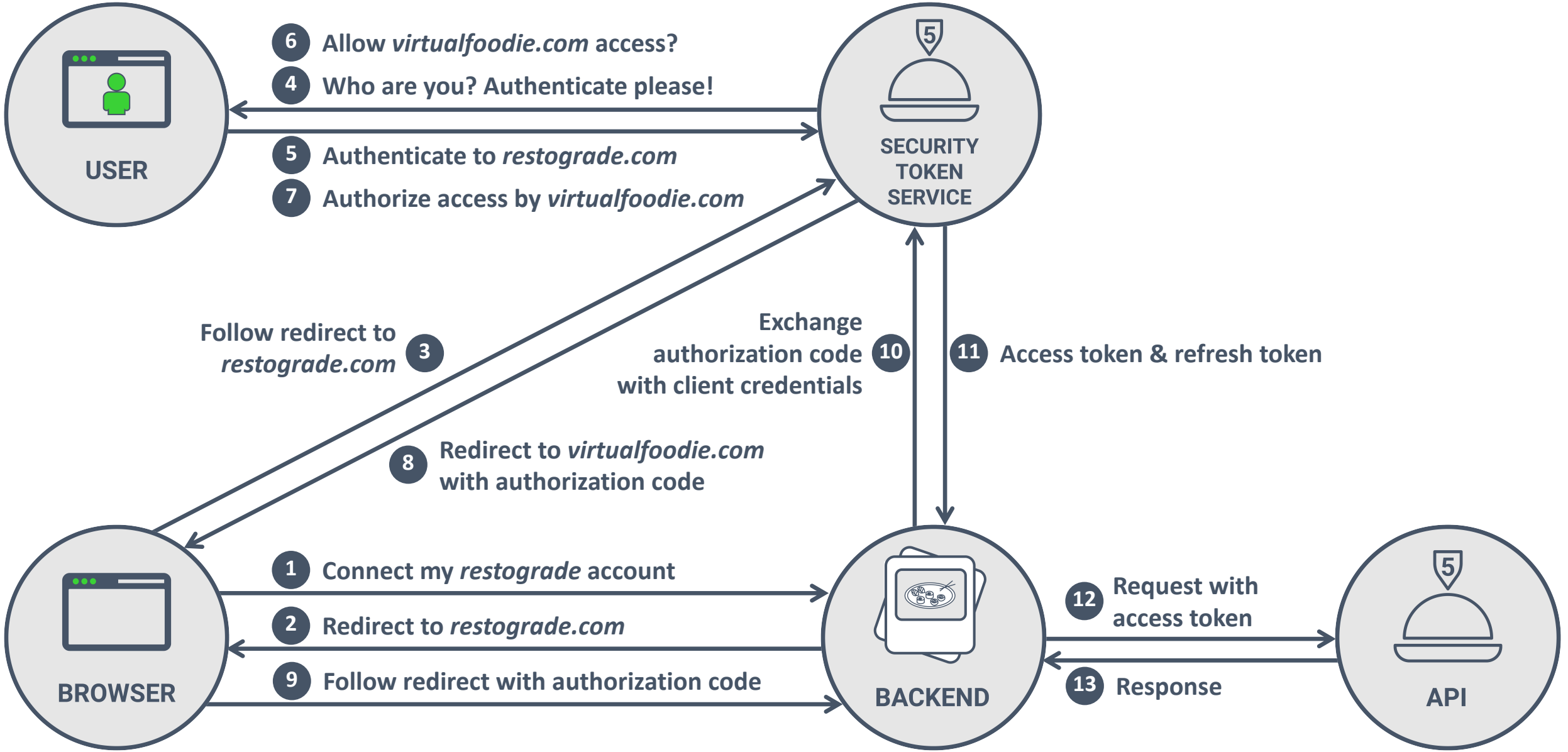


CLIENT REGISTRATION

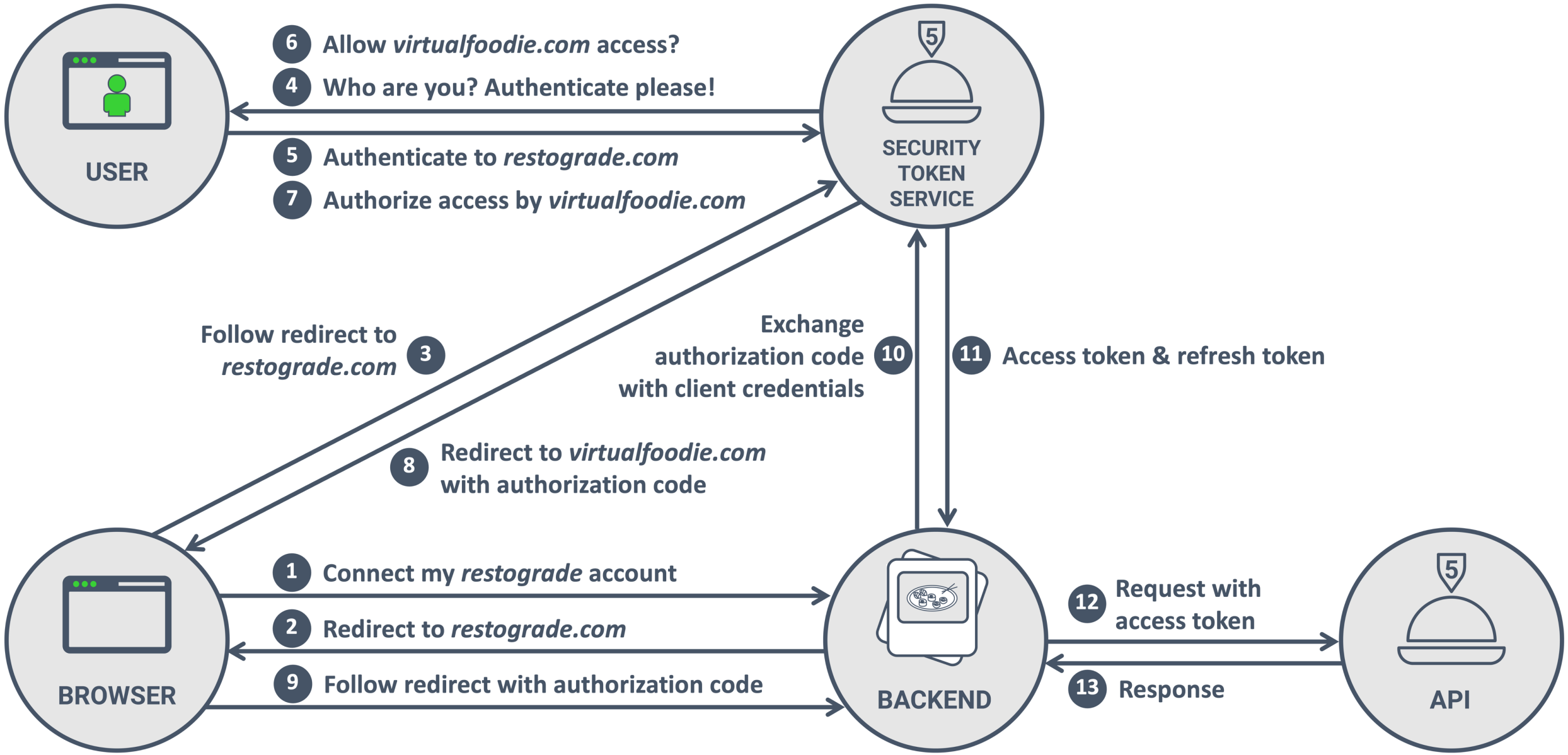
- Clients need to be registered with the STS
  - Who is responsible for registering a client depends on the type of system
    - Open systems (e.g., Google, Github, …) can be self-managed by client developers
    - Closed systems (e.g., enterprise STS) are managed by an admin team
  - During client registration, the client receives a unique client ID
  - Client registration also requires specifying client redirect URIs

- Backend clients also receive client credentials for authentication
  - These clients are also known as ***confidential clients***
    - Confidential clients run in an environment where they can securely store sensitive credentials
  - Typical client credentials are the ***client ID*** and ***client secret***
    - Basically a username and password for the client
    - Stronger key-based authentication mechanisms are also supported by the specifications

THE *AUTHORIZATION CODE* FLOW



THE *AUTHORIZATION CODE* FLOW



2 3 *The redirect URI*

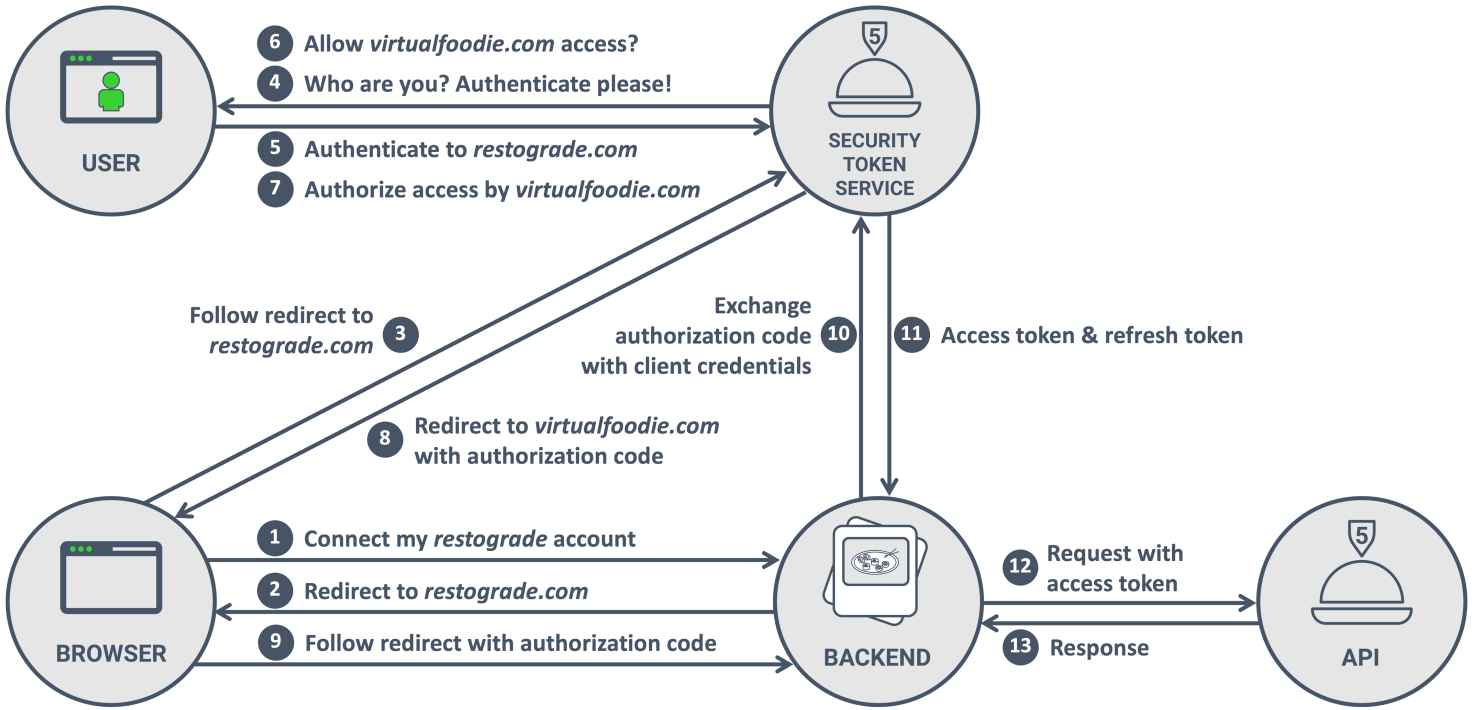
```
1  https://sts.restograde.com/authorize
2    ?response_type=code
3    &client_id=lY5g0BKB7Mow4yDlb6rdGPsO2i1g7Osv
4    &scope=read
5    &redirect_uri=https://virtualfoodie.com/callback
6    &state=s0wzojm2w8c23xzprkk6
```

Indicates the *authorization code flow*

The client requesting access

Where the STS should send the token

THE *AUTHORIZATION CODE* FLOW



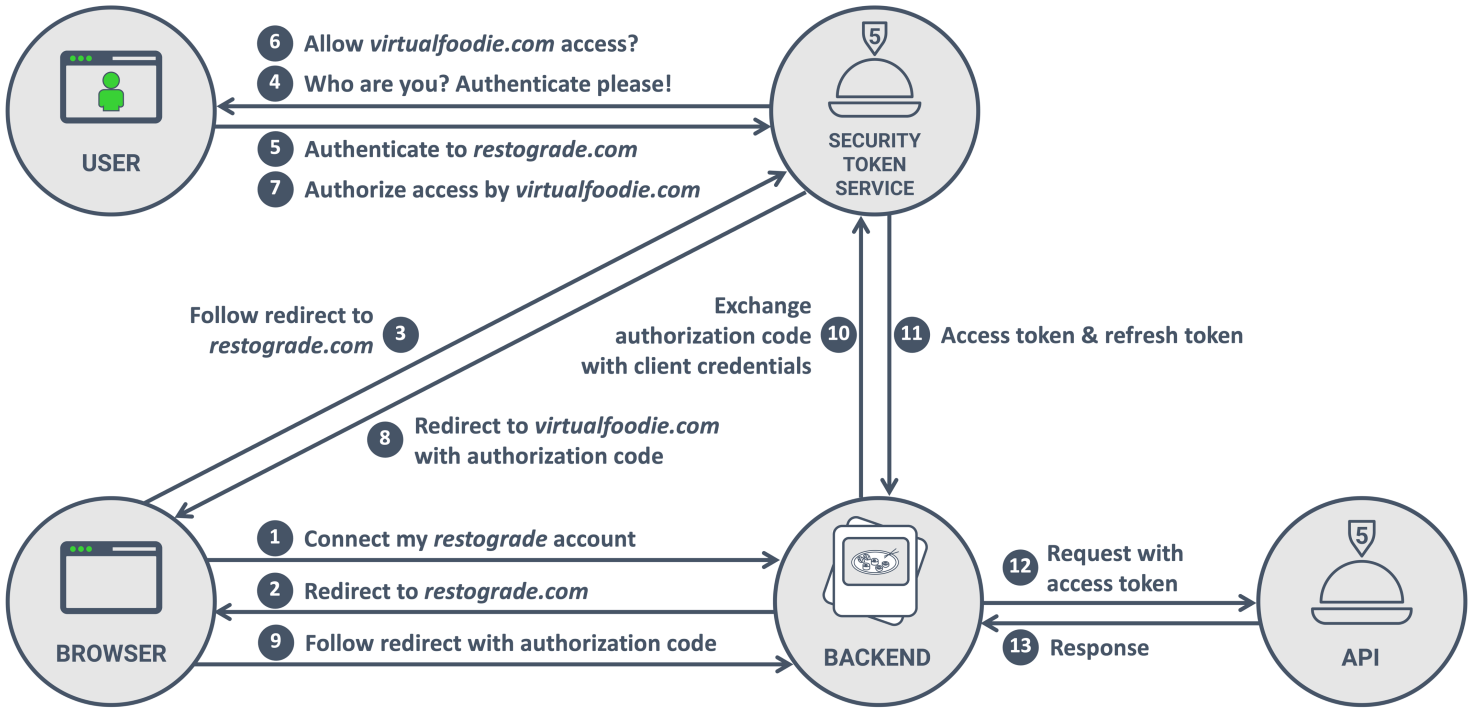
8 9 *The callback URI*

```
1 https://virtualfoodie.com/callback
2   ?code=SplxlOBeZQQYbYS6WxSbIA
3   &state=s0wzojm2w8c23xzprkk6
```

The callback URI from before

The authorization code

THE *AUTHORIZATION CODE* FLOW

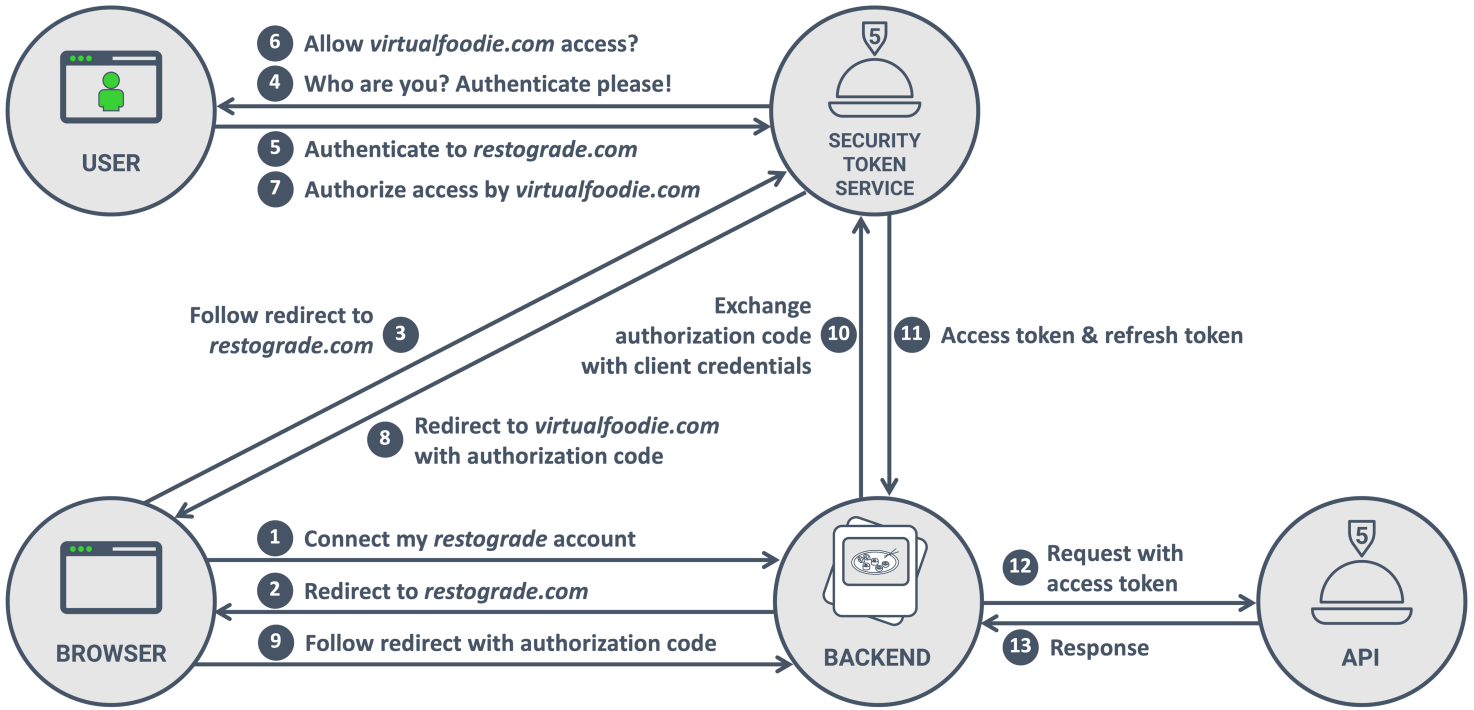


10 *The request to exchange the authorization code*

```
1  POST https://sts.restograde.com/oauth/token
2
3   grant_type=authorization_code
4  &client_id=lY5g0BKB7Mow4yDlb6rdGPsO2i1g7Osv
5  &client_secret=6ODRv0g…OVOSWI
7  &redirect_uri=https://virtualfoodie.com/callback
8  &code=SplxlOBeZQQYbYS6WxSbIA
```

- `grant_type=authorization_code` — Indicates the code exchange request
- `&client_id=lY5g0BKB7Mow4yDlb6rdGPsO2i1g7Osv` — The client exchanging the code
- `&client_secret=6ODRv0g…OVOSWI` — The client needs to authenticate to the STS
- `&redirect_uri=https://virtualfoodie.com/callback` — The redirect URI used before
- `&code=SplxlOBeZQQYbYS6WxSbIA` — The code received in step 9

THE *AUTHORIZATION CODE* FLOW



11 *The response from the Security Token Service*

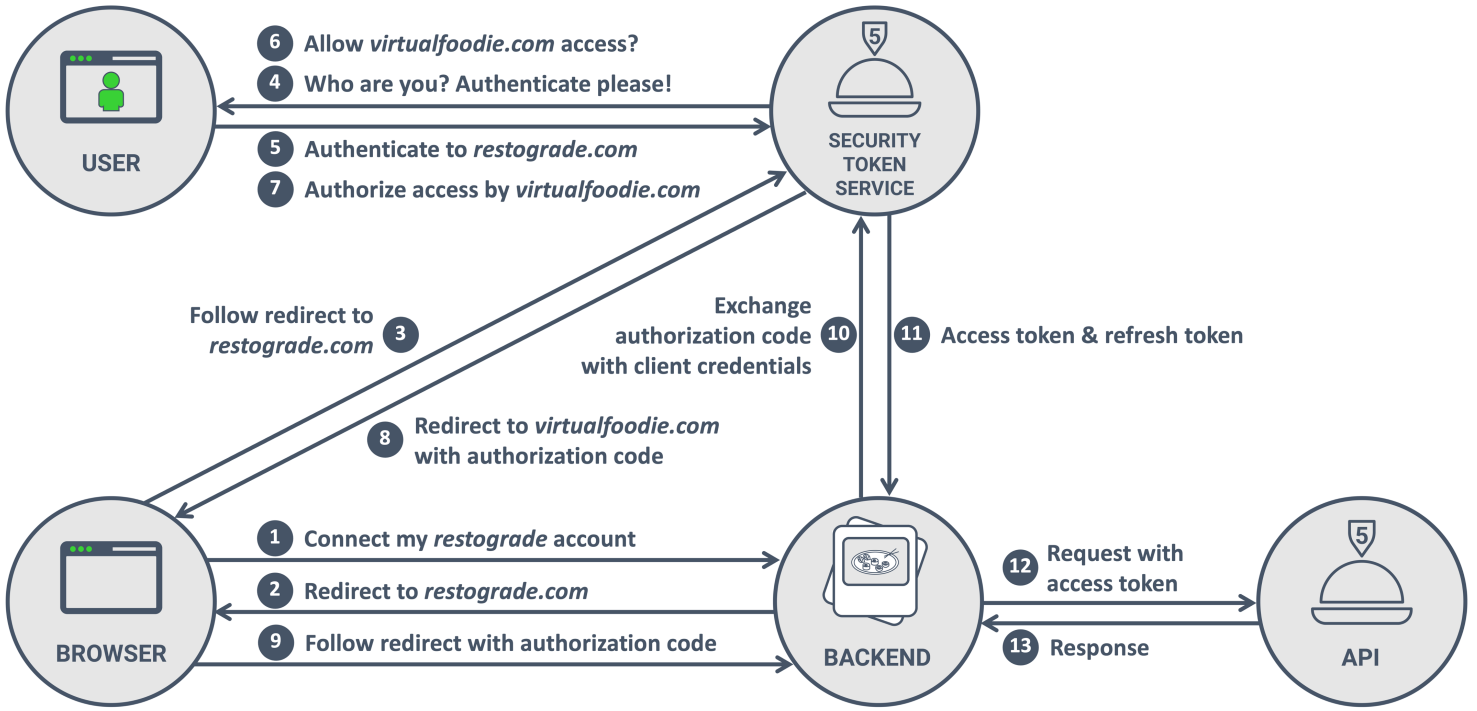
```
{
  "access_token": "eyJhbGci0...du6TY9w",
  "token_type": "Bearer",
  "expires_in": 3600,
  "refresh_token": "8xL0xBtZp8"
}
```

The access token to access APIs

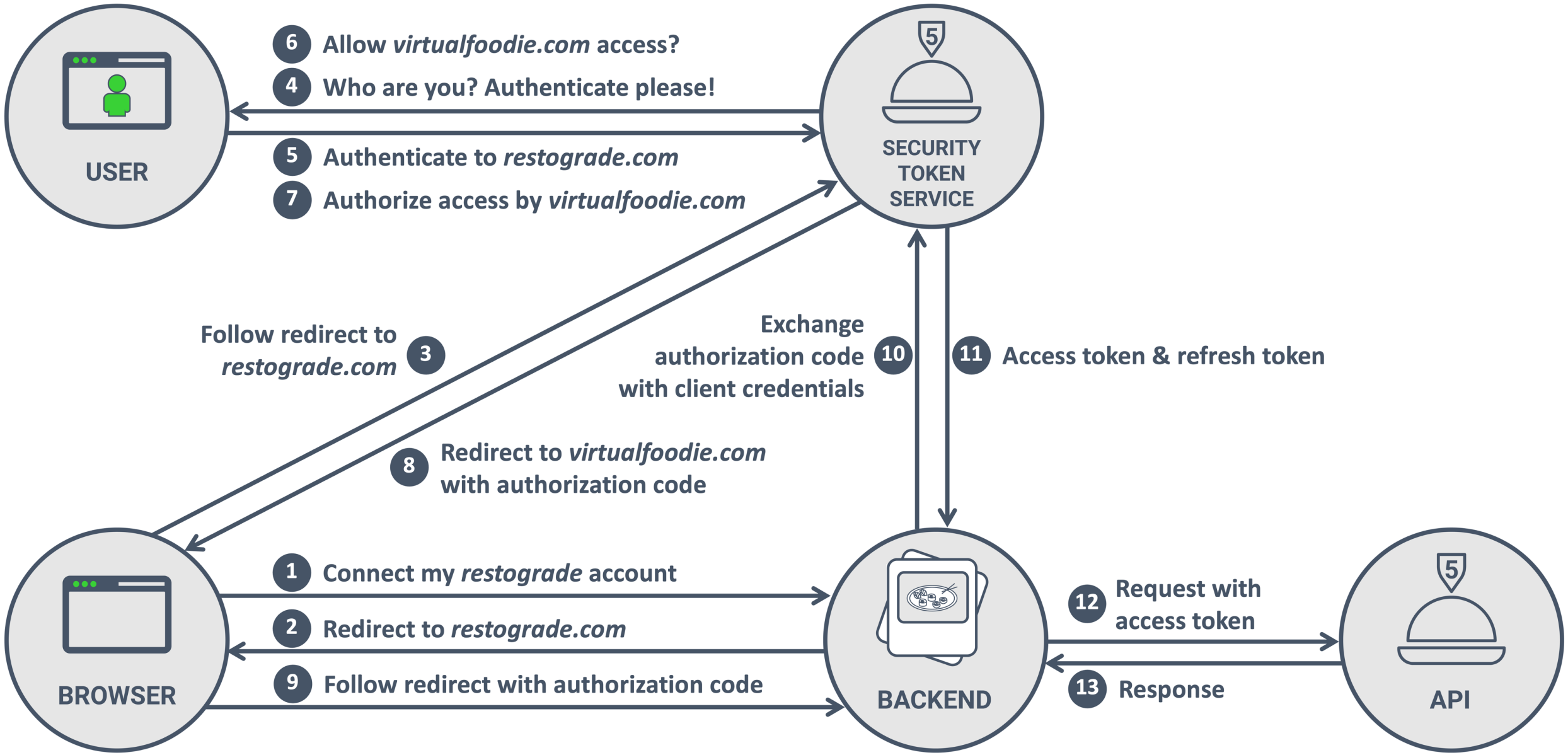
The expiration time of the access token

The refresh token

THE *AUTHORIZATION CODE* FLOW



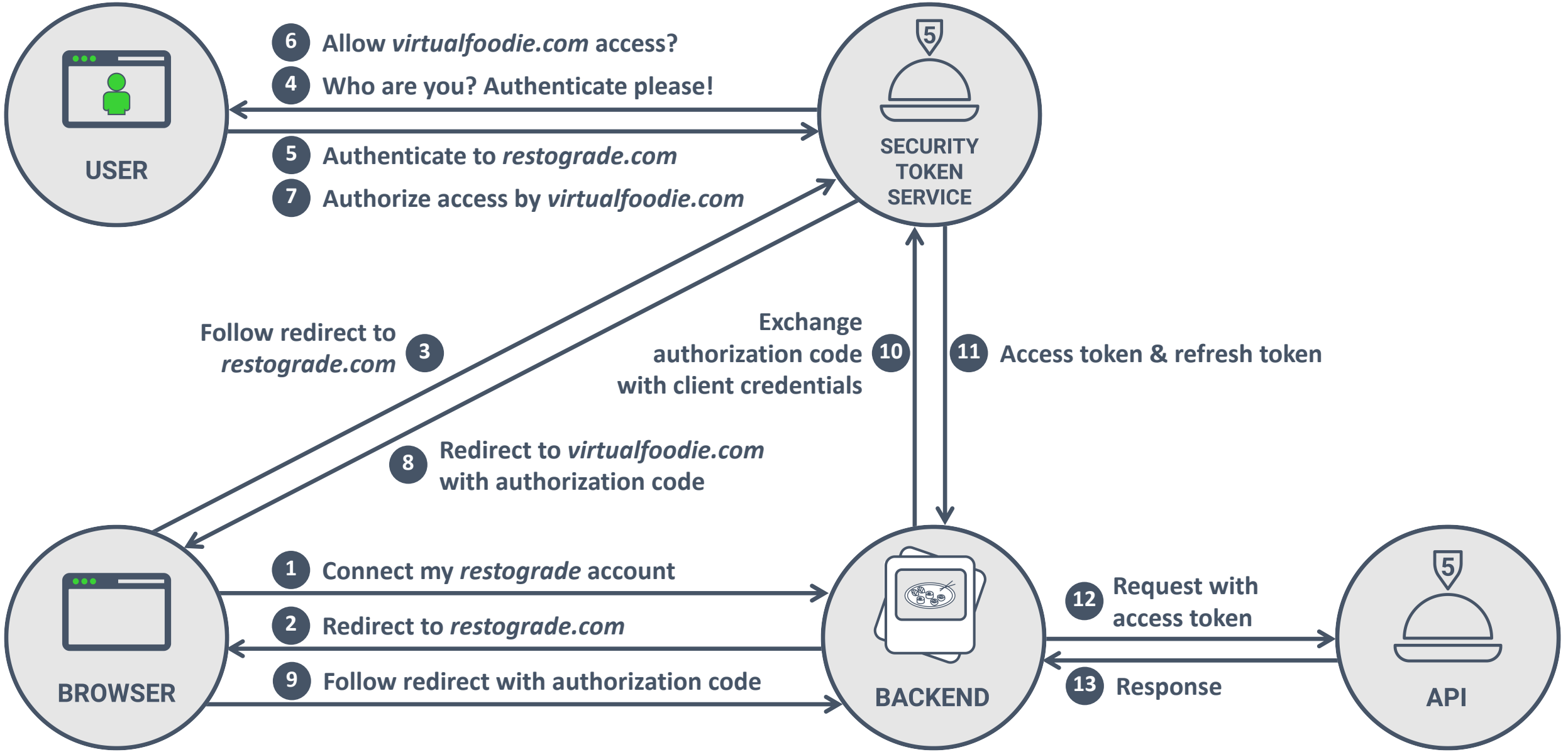
THE *AUTHORIZATION CODE* FLOW



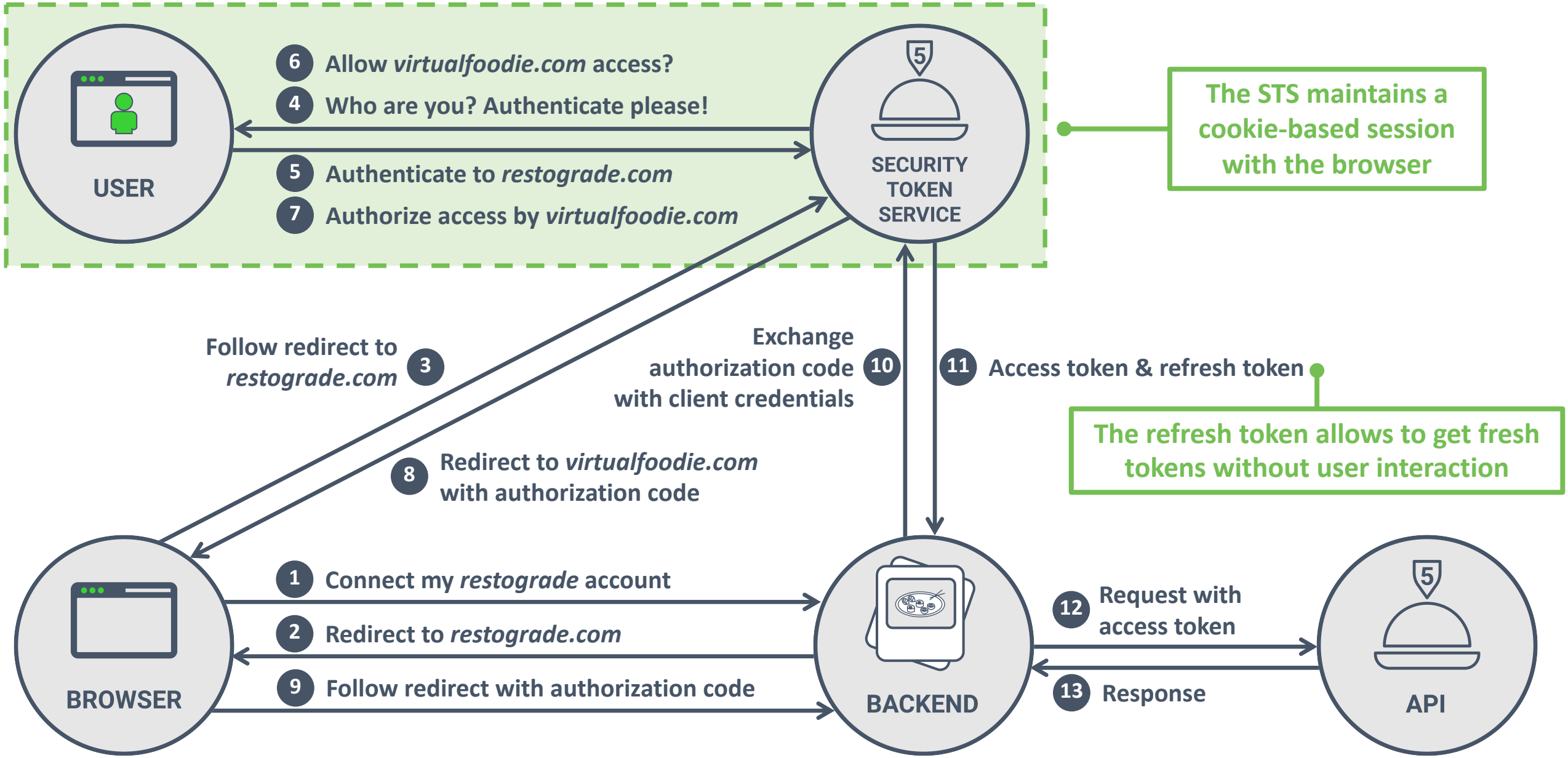
Full disclosure: I am a happy Auth0 user. I am also working closely with the Auth0 developer advocates as an Ambassador and Expert. I am not paid by Auth0, nor do I benefit from recommending Auth0 to others.

Running an *Authorization Code* flow

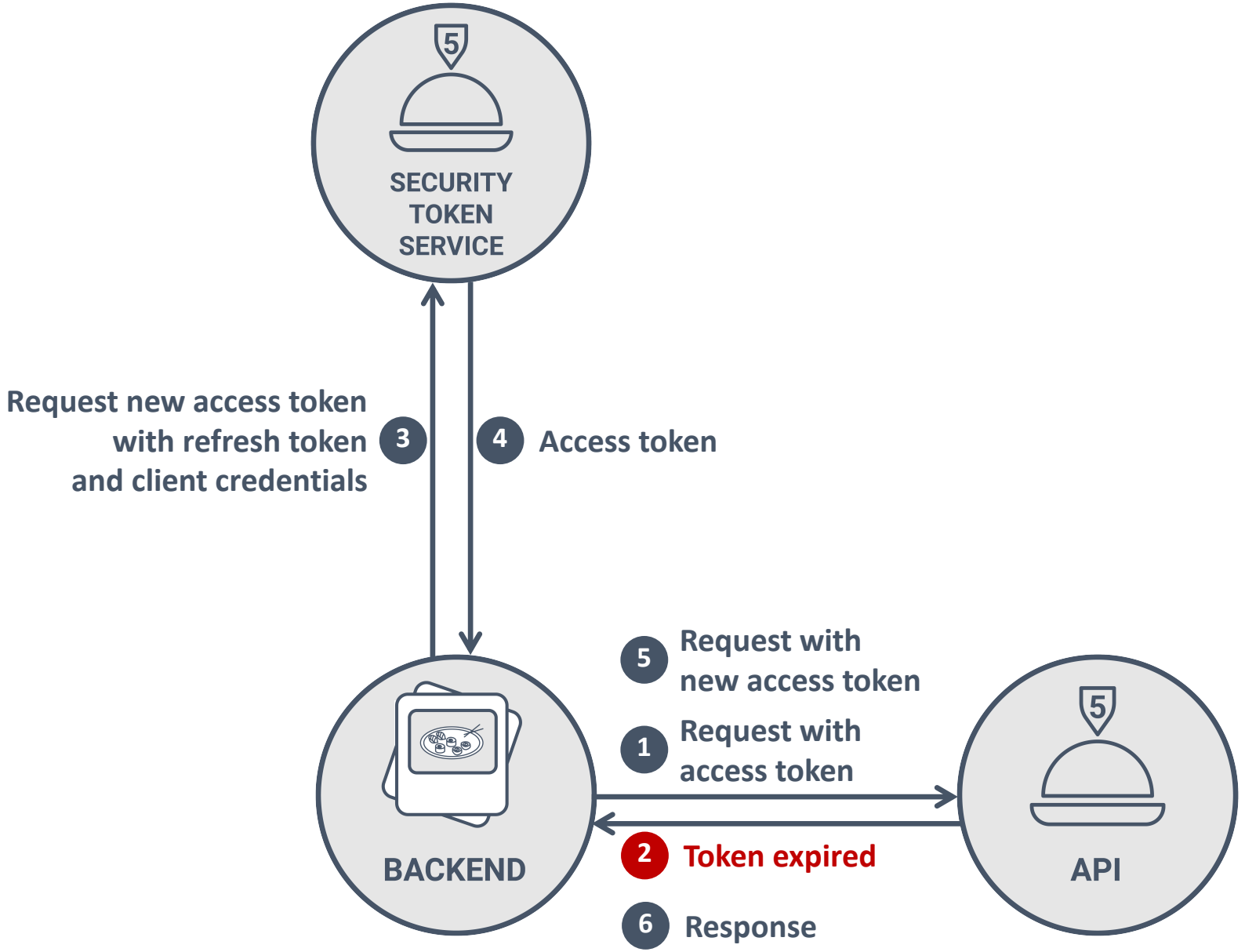
THE *AUTHORIZATION CODE* FLOW



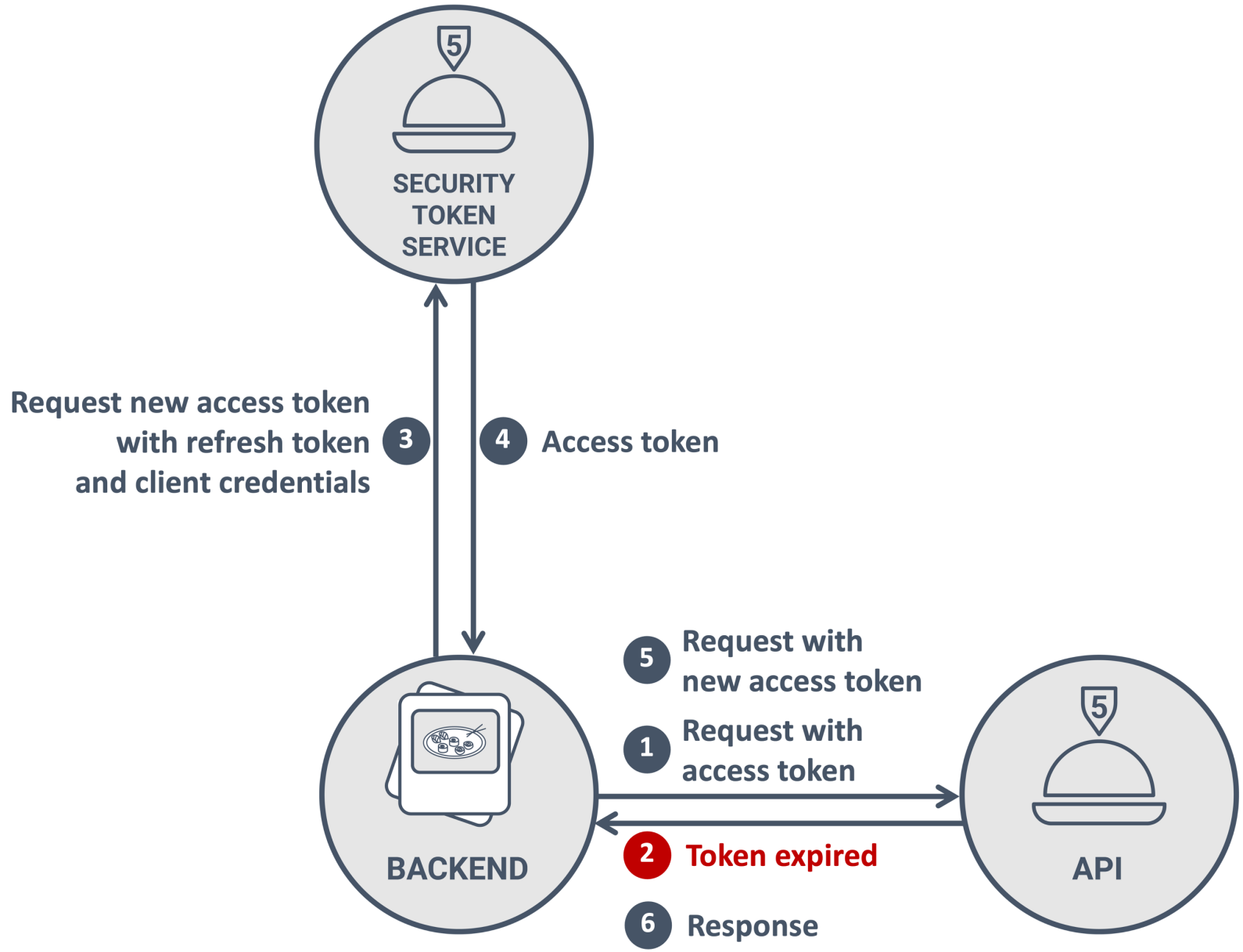
LONG-TERM ACCESS WITH THE *AUTHORIZATION CODE* FLOW



THE *REFRESH TOKEN* FLOW



THE *REFRESH TOKEN* FLOW



3 *The request to exchange a refresh token for an access token*

```
1  POST https://sts.restograde.com/oauth/token
2
3   grant_type=refresh_token
4  &client_id=lY5g0BKB7Mow4yDlb6rdGPsO2i1g7Osv
5  &client_secret=6ODRv0g…OVOSWI
7  &refresh_token=8xLOxBtZp8
```

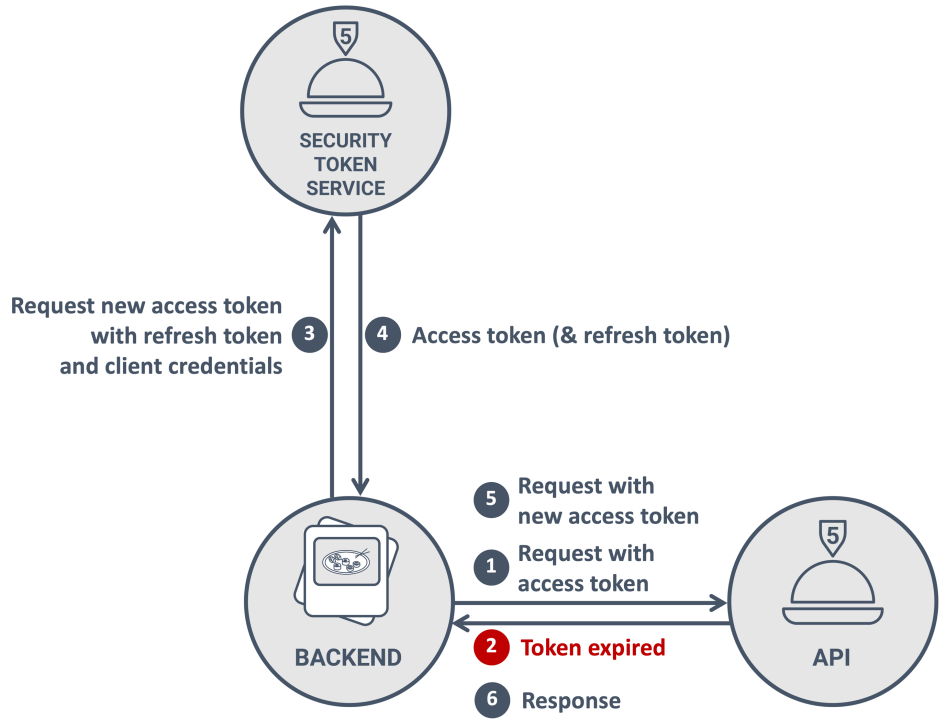
Indicates the refresh token flow

The client exchanging the code

The client needs to authenticate to the STS

The refresh token obtained during the flow

THE *REFRESH TOKEN* FLOW



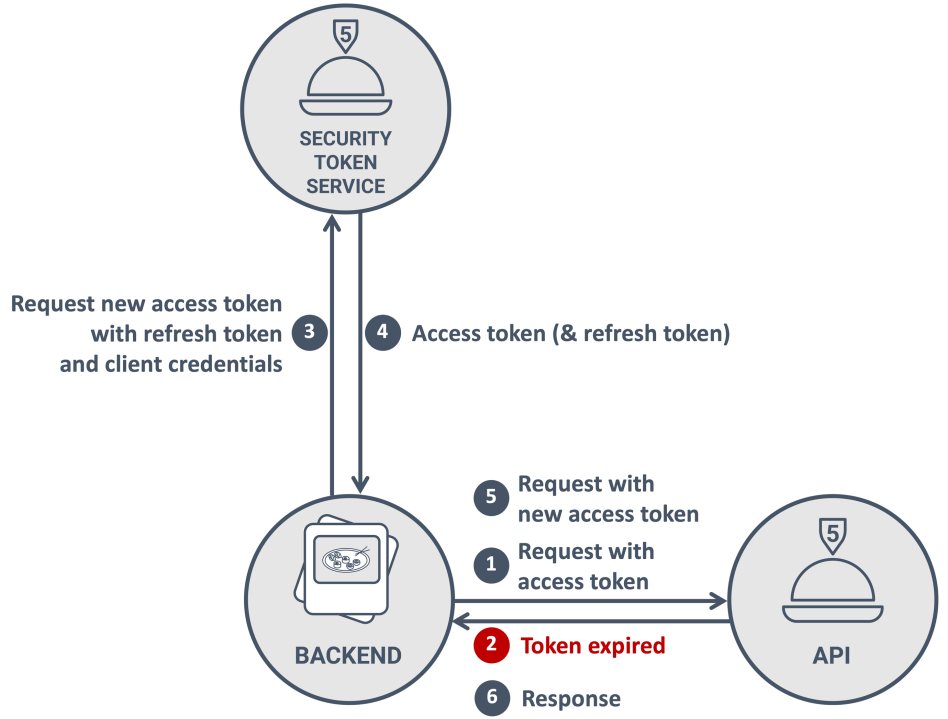
4 *The response from the Security Token Service*

```
{
  "access_token": "eyJhbGciO...du6TY9w",
  "token_type": "Bearer",
  "expires_in": 3600,
}
```

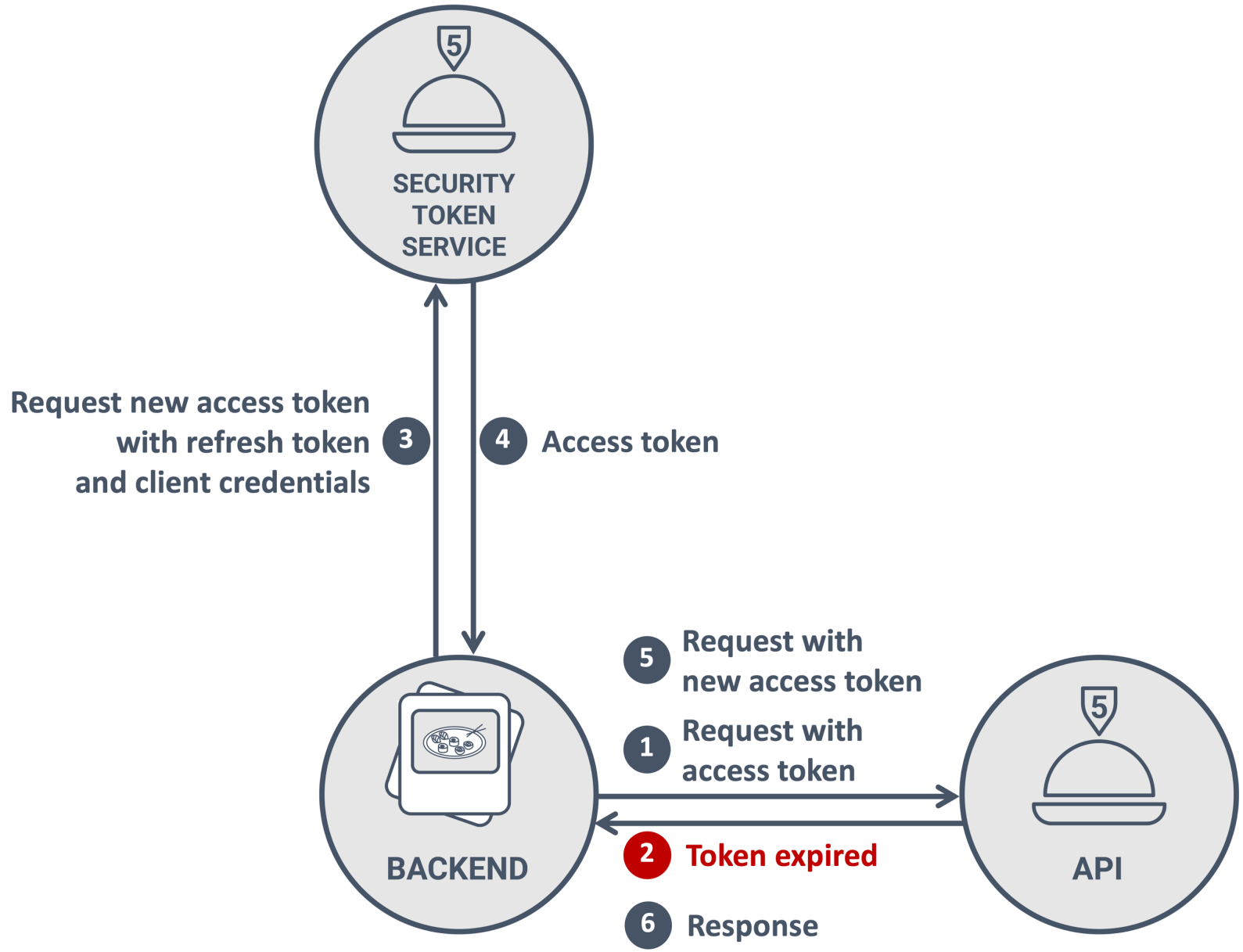
The access token to access APIs

The expiration time of the access token

THE *REFRESH TOKEN* FLOW



THE *REFRESH TOKEN* FLOW



Refresh tokens and the refresh token flow

- The Refresh Token flow allows a client to obtain a new access token
  - Refresh tokens give a client long-term access on behalf of the user
  - Refresh tokens are used by the client and consumed by the STS
- Refresh tokens are sensitive and require adequate protection
  - Confidential clients need to authenticate when using a refresh token
  - Clients need to protect refresh tokens in storage
    - Encrypt before storing and preferably keep in a separate service and data store
  - Advanced defenses include *binding the token to the sender* and *refresh token rotation*
- The STS has full control over the properties of refresh tokens
  - The STS can decide to give a client a refresh token, along with the lifetime of the refresh token
  - Clients often use the ***offline_access*** scope to indicate they would like a refresh token
    - This scope value is originally defined in the OIDC spec, but typically also applies to OAuth 2.0

Running an *Refresh Token* flow

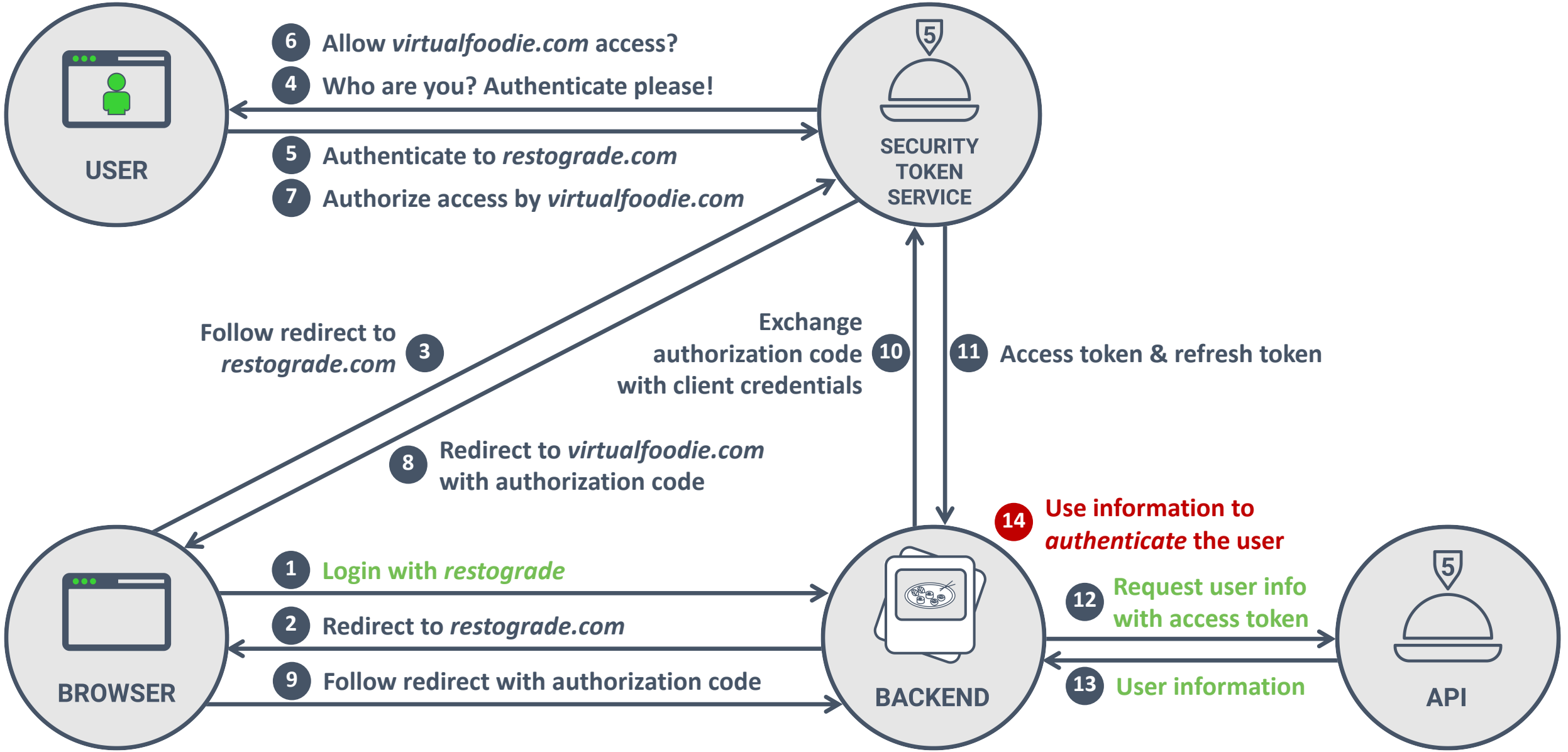
The OAuth 2.0 *Authorization Code* flow

- The client is a backend web application, running in a *secure* environment
  - The user authorizes the client to access APIs on their behalf
  - The *Authorization Code* flow gives the client an access token and refresh token
    - The access token is used by the client to access APIs on behalf of the user
    - The refresh token allows the client to obtain new access tokens, without user involvement

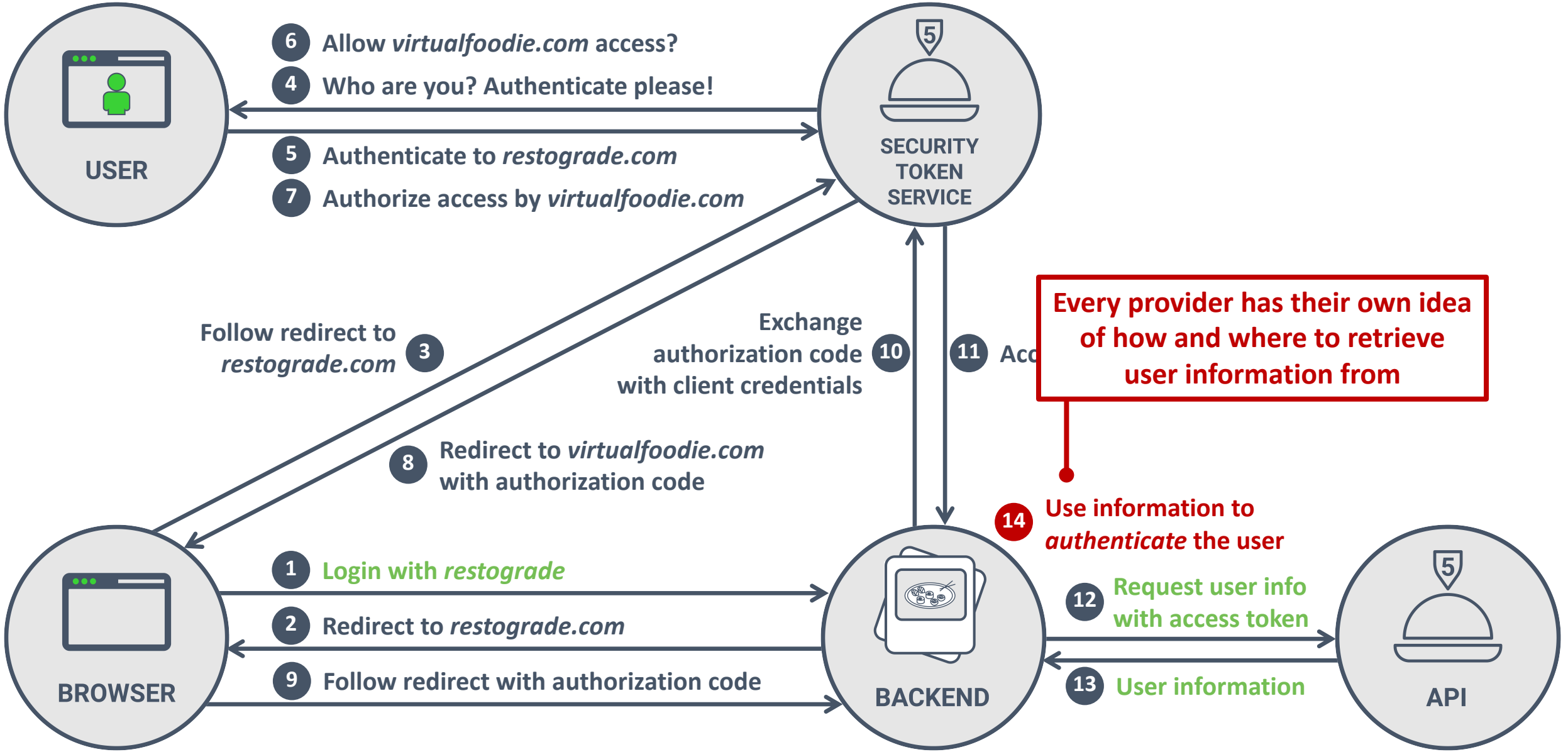
- The *Authorization Code* flow has several security measures built-in
  - The client needs to be registered with the proper redirect URIs
  - The client needs to authenticate on backchannel requests to the STS
    - Concretely, client authentication is needed to exhange an authorization code or refresh token
  - Authorization codes can only be used once in a very limited time window


RESTOGRADE
Login with Restograde

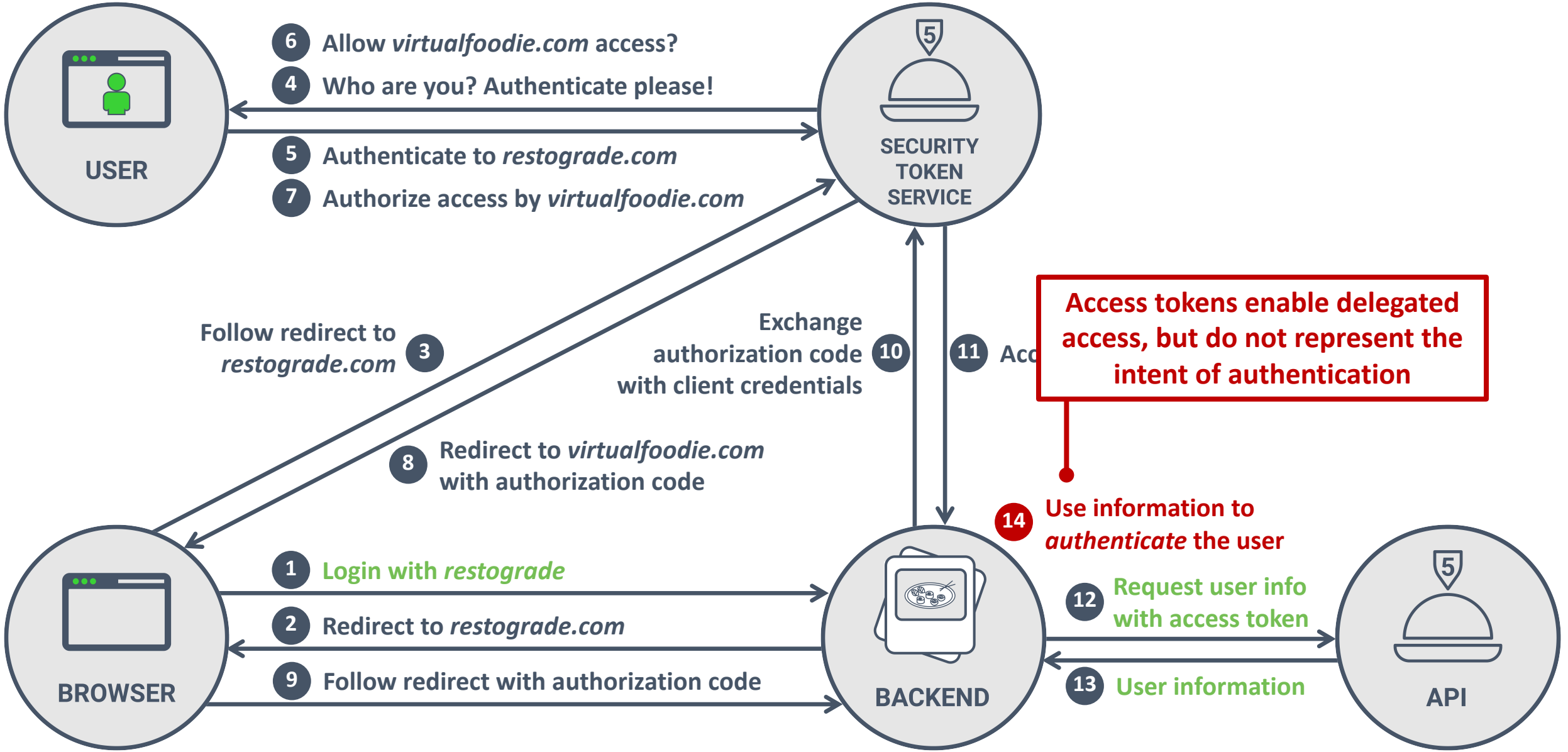
PSEUDO-AUTHENTICATION WITH OAUTH 2.0



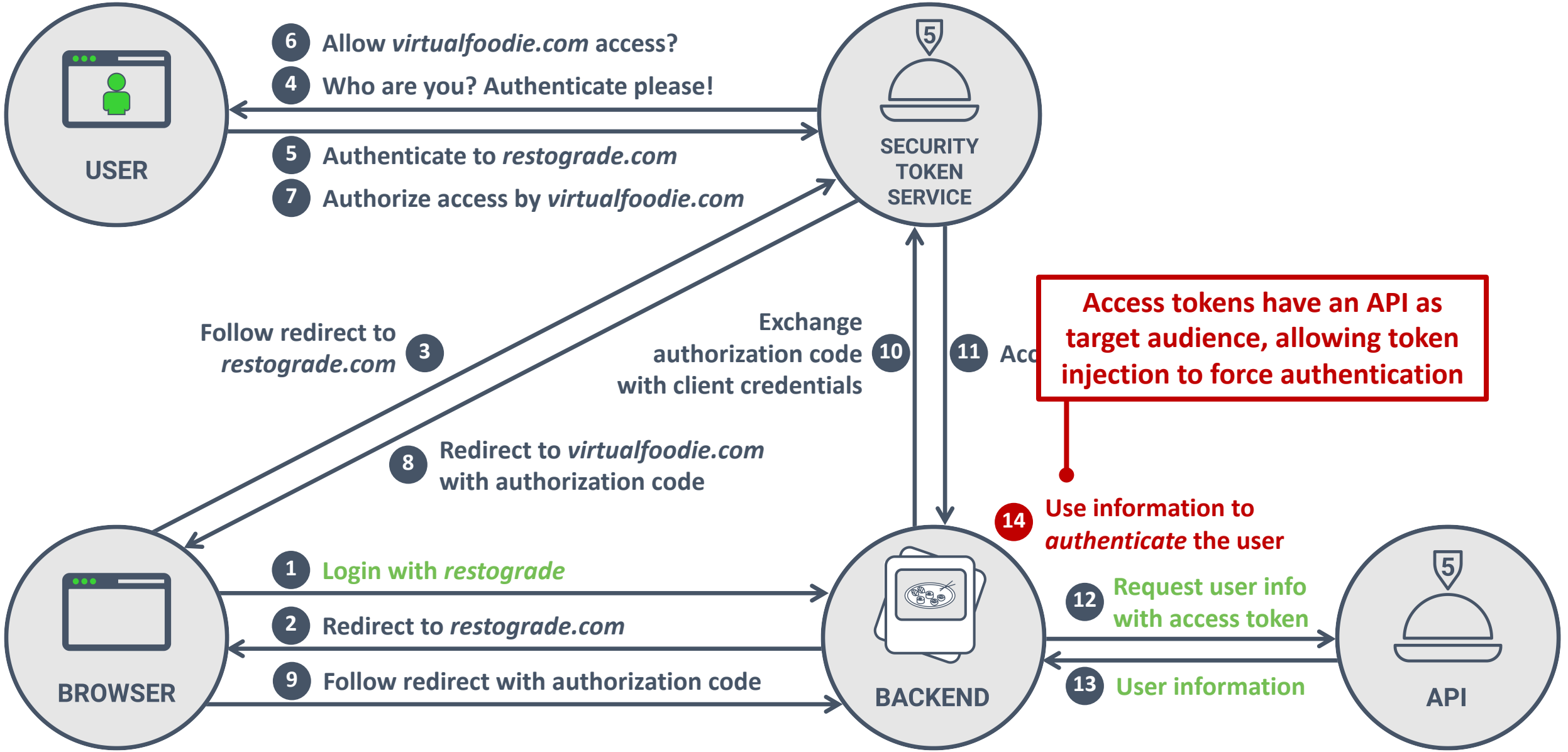
Pseudo-authentication with OAuth 2.0



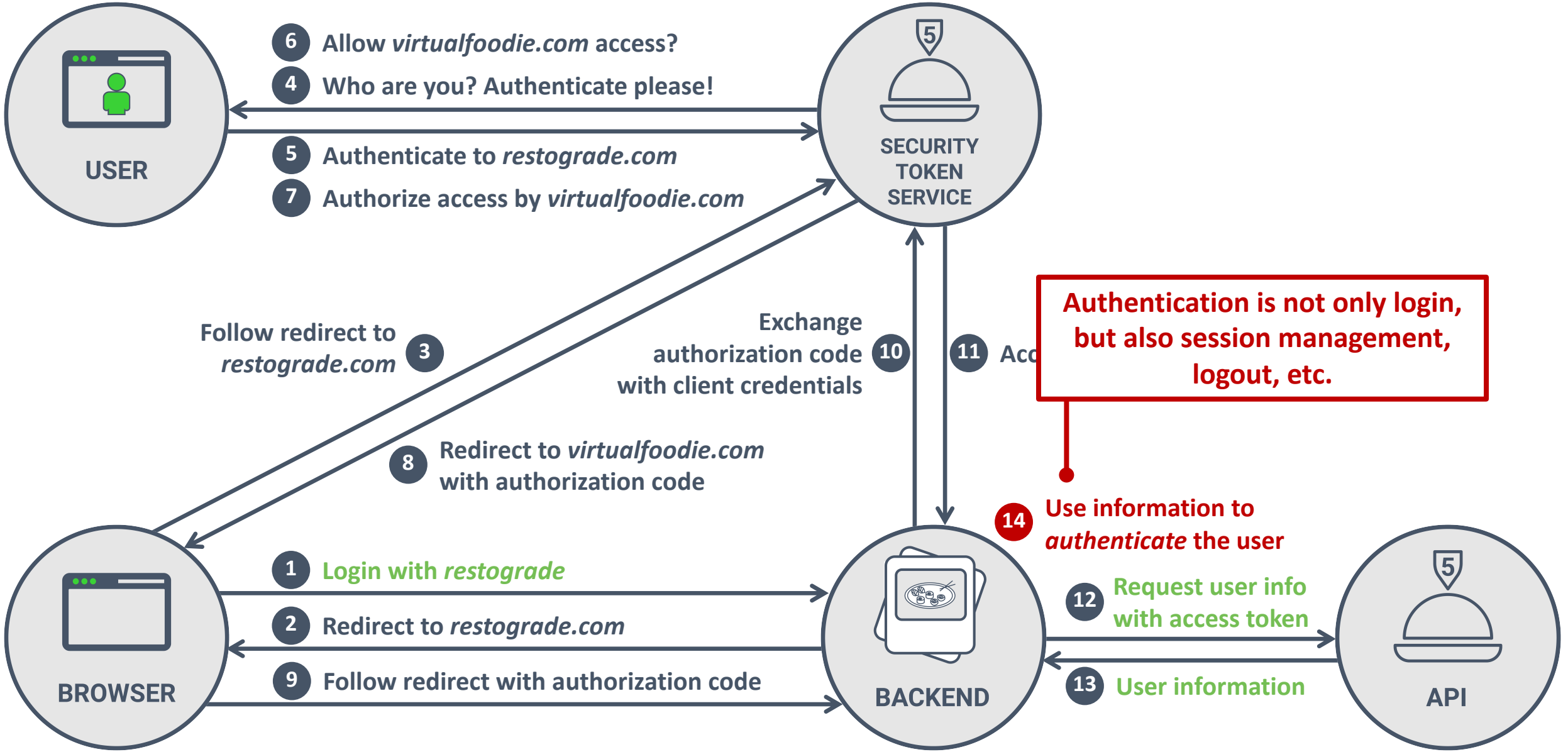
Pseudo-authentication with OAuth 2.0



Pseudo-authentication with OAuth 2.0



Pseudo-authentication with OAuth 2.0



Do not build authentication with OAuth 2.0, but use OpenID Connect instead

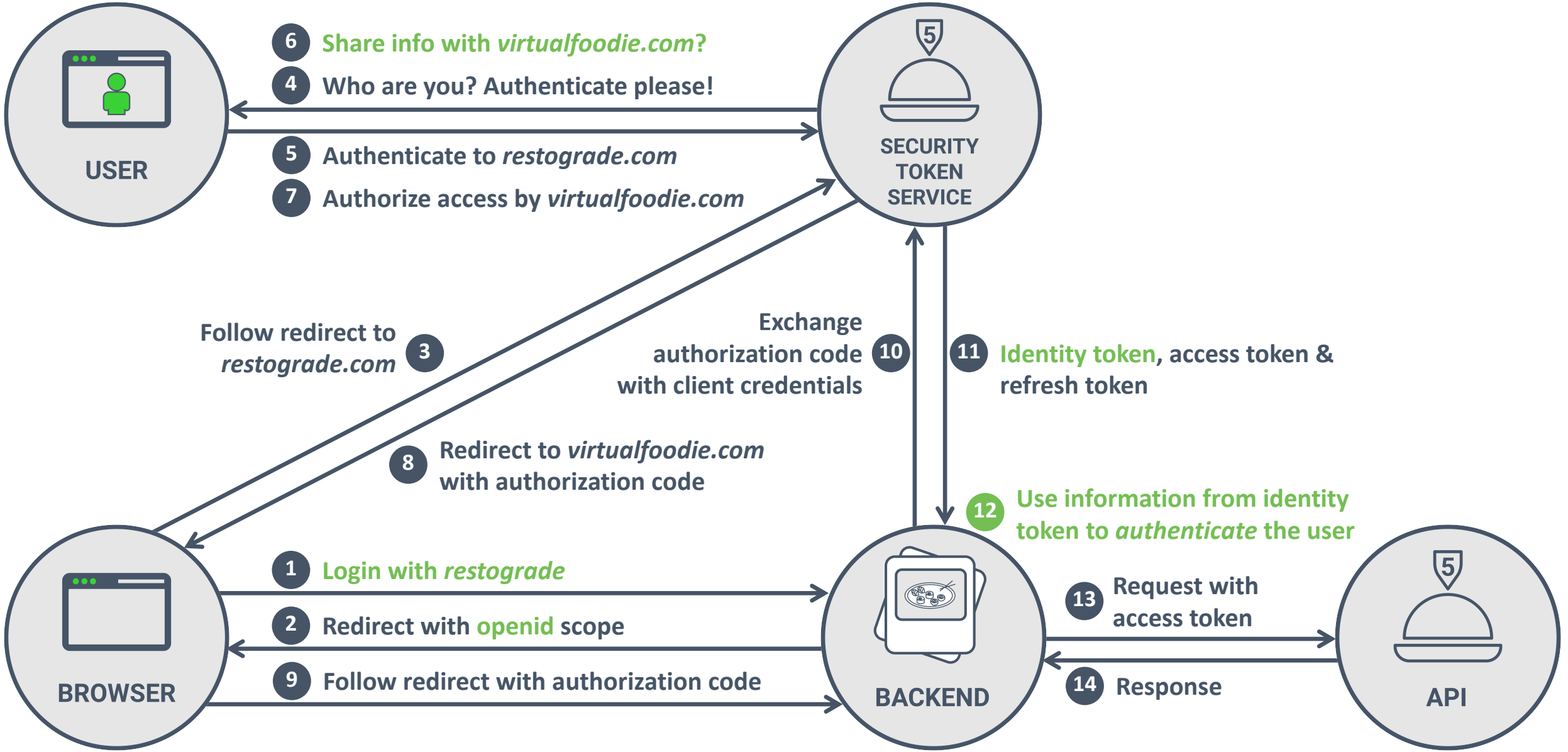
OpenID Connect (OIDC)

- OIDC builds an authentication protocol on top of OAuth 2.0
  - OAuth 2.0 provides the framework with a toolbox full of building blocks
  - OIDC selectively picks the required building blocks to implement authentication

- OIDC adds a large number of features, but most commonly used are…
  - ***Identity tokens*** to support authentication on top of an OAuth 2.0 flow
  - The ***Userinfo endpoint*** to obtain information about the authenticated user
  - ***Issuer discovery***, allowing the automatic configuration of client applications
  - ***Session management*** features, such as change detection and logout

- OIDC uses OAuth 2.0 under the hood, so both are often used together
  - Authenticate the user with OIDC and access APIs with the OAuth 2.0 access token

THE *AUTHORIZATION CODE* FLOW WITH OIDC



Running an *OIDC Authorization Code* flow

IDENTITY TOKENS

- Identity tokens contain information about the user's authentication
  - The token is a signed JWT with a set of claims about the user and the authentication
  - The token is signed by the issuer, which is the Security Token Service
- There are a number of mandatory claims that are always present
  - ***sub***: a unique identifier for this user within the realm of the STS
  - ***iss***: the identifier of the issuer of the token (i.e., the STS)
  - ***aud***: the target audience of the identity token (i.e., the client application)
  - ***exp***: the expiration date of the identity token
- The application can use the information in the identity token for authentication
  - Link the user with this particular ***sub*** to a user within the application

User registration with OpenID Connect

HEADER: ALGORITHM & TOKEN TYPE


```
{
  "alg": "RS256",
  "typ": "JWT",
  "kid":
"NTVBOTU3MzBBOEUwNzhBQ0VGMkQ0QUU5QTYxQUUyOUNEQUUxNjEyMw"
}
```


PAYLOAD: DATA


```
{
  "nickname": "philippe",
  "name": "philippe@pragmaticwebsecurity.com",
  "picture":
"https://s.gravatar.com/avatar/f40d64a4b16789508062e664b
e6ae575?
s=480&r=pg&d=https%3A%2F%2Fcdn.auth0.com%2Favatars%2Fph.
png",
  "updated_at": "2020-05-11T09:11:30.325Z",
  "email": "philippe@pragmaticwebsecurity.com",
  "email_verified": false,
  "iss": "https://sts.restograde.com/",
  "sub": "auth0|5eb916c258bdb50bf20366c6",
  "aud": "FN983CEYgx4mdUg3NKNKHjwfNAL5Fb42",
  "iat": 1589188769,
  "exp": 1589224769
}
```


User DB of the Virtual Foodie backend

ID	Name	Sub
1	alice	auth0|8c34361ea1c8bff697e3a81e

User registration with OpenID Connect

HEADER: ALGORITHM & TOKEN TYPE


```
{
  "alg": "RS256",
  "typ": "JWT",
  "kid":
"NTVBOTU3MzBBOEUwNzhBQ0VGMkQ0QUU5QTYxQUUyOUNEQUUxNjEyMw"
}
```


PAYLOAD: DATA


```
{
  "nickname": "philippe",
  "name": "philippe@pragmaticwebsecurity.com",
  "picture":
"https://s.gravatar.com/avatar/f40d64a4b16789508062e664b
e6ae575?
s=480&r=pg&d=https%3A%2F%2Fcdn.auth0.com%2Favatars%2Fph.
png",
  "updated_at": "2020-05-11T09:11:30.325Z",
  "email": "philippe@pragmaticwebsecurity.com",
  "email_verified": false,
  "iss": "https://sts.restograde.com/",
  "sub": "auth0|5eb916c258bdb50bf20366c6",
  "aud": "FN983CEYgx4mdUg3NKNKHjwfNAL5Fb42",
  "iat": 1589188769,
  "exp": 1589224769
}
```


User DB of the Virtual Foodie backend

ID	Name	Sub
1	alice	auth0|8c34361ea1c8bff697e3a81e
2	philippe	auth0|5eb916c258bdb50bf20366c6

The user is created with the information from the identity token

The *sub* value uniquely identifies the user within the issuer

User authentication with OpenID Connect

HEADER: ALGORITHM & TOKEN TYPE


```
{
  "alg": "RS256",
  "typ": "JWT",
  "kid":
"NTVBOTU3MzBBOEUwNzhBQ0VGMkQ0QUU5QTYxQUUyOUNEQUUxNjEyMw"
}
```


PAYLOAD: DATA


```
{
  "nickname": "philippe",
  "name": "philippe@pragmaticwebsecurity.com",
  "picture":
"https://s.gravatar.com/avatar/f40d64a4b16789508062e664b
e6ae575?
s=480&r=pg&d=https%3A%2F%2Fcdn.auth0.com%2Favatars%2Fph.
png",
  "updated_at": "2020-05-11T09:11:30.325Z",
  "email": "philippe@pragmaticwebsecurity.com",
  "email_verified": false,
  "iss": "https://sts.restograde.com/",
  "sub": "auth0|5eb916c258bdb50bf20366c6",
  "aud": "FN983CEYgx4mdUg3NKNKHjwfNAL5Fb42",
  "iat": 1589188914,
  "exp": 1589224914
}
```


User DB of the Virtual Foodie backend

ID	Name	Sub
1	alice	auth0|8c34361ea1c8bff697e3a81e
2	philippe	auth0|5eb916c258bdb50bf20366c6

The session is populated with the information about the authenticated user

The *sub* value is used to find the authenticated user in the Virtual Foodie database

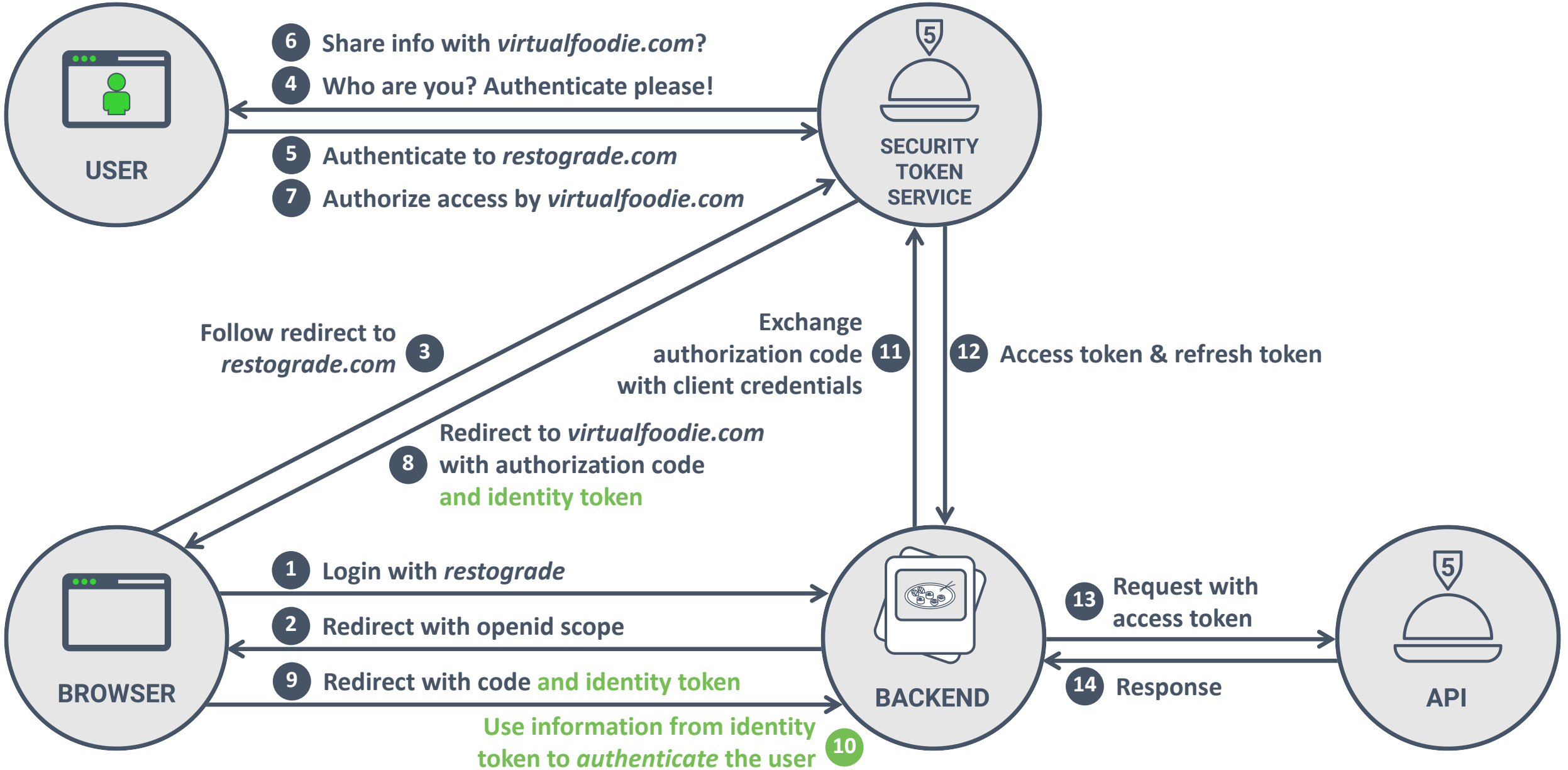
OIDC AND THE IDENTITY TOKEN

- OIDC allows the client to delegate authentication to a central service
  - OIDC allows an application to offload the increasingly difficult authentication process
  - The de facto standard to implement Single Sign-On in modern applications

- The client runs an OIDC flow to obtain an identity token
  - The client uses scopes to indicate the required information (*openid, profile, email, …)*
  - The identity token contains information about the user's authentication
  - The *iss* claim identifies the STS and the *sub* claim identifies the user

- The *sub* claim contains the user's unique identifier within the realm of the STS
  - The identifier does not change during the lifetime of the user within the STS
  - The identifier can be matched against a known user in the client application

THE OIDC *HYBRID* FLOW



THE OIDC *HYBRID* FLOW

- The OIDC hybrid flow allows a client to obtain the identity token faster
  - The identity token is delivered through the frontchannel instead of the backchannel
  - The OAuth 2.0 tokens are still delivered through the backchannel

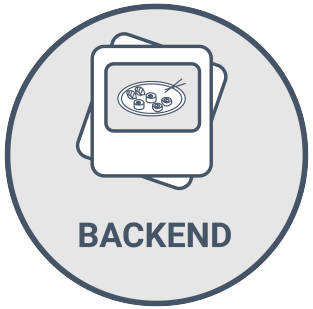
- The steps in the *Hybrid* flow are almost identical to the *Authorization Code* flow
  - The flow is initialized with the *"**code id_token**"* response type
  - The redirect back to the client contains the identity token in the URL
  - The identity token contains the hash of the authorization code to protect flow integrity

- **The *Hybrid* flow is being phased out in favor of the *Authorization Code* flow**
  - The benefits from getting the identity token sooner are limited to non-existent
  - The implementation of identity token validation in the client is crucial yet difficult

Best practices for backend applications

- Backend applications can use OAuth 2.0 and OIDC in isolation or in combination
  - E.g., authentication only (OIDC), API access only (OAuth 2.0), or both
- Respect the purpose and intented audience of the tokens
  - Identity tokens are intended for the client, to handle authentication
  - Access tokens are used by the client and sent to APIs to authorize requests
  - Refresh tokens are used by the client and sent to the STS to obtain new tokens
- Tokens are sensitive data and should be handled securely
  - Encrypt tokens before storing them
  - Preferably store tokens in a separate service
  - Do not request or store tokens that are not needed
    - E.g., one-shot API access does not require a refresh token

OVERVIEW OF BEST PRACTICES



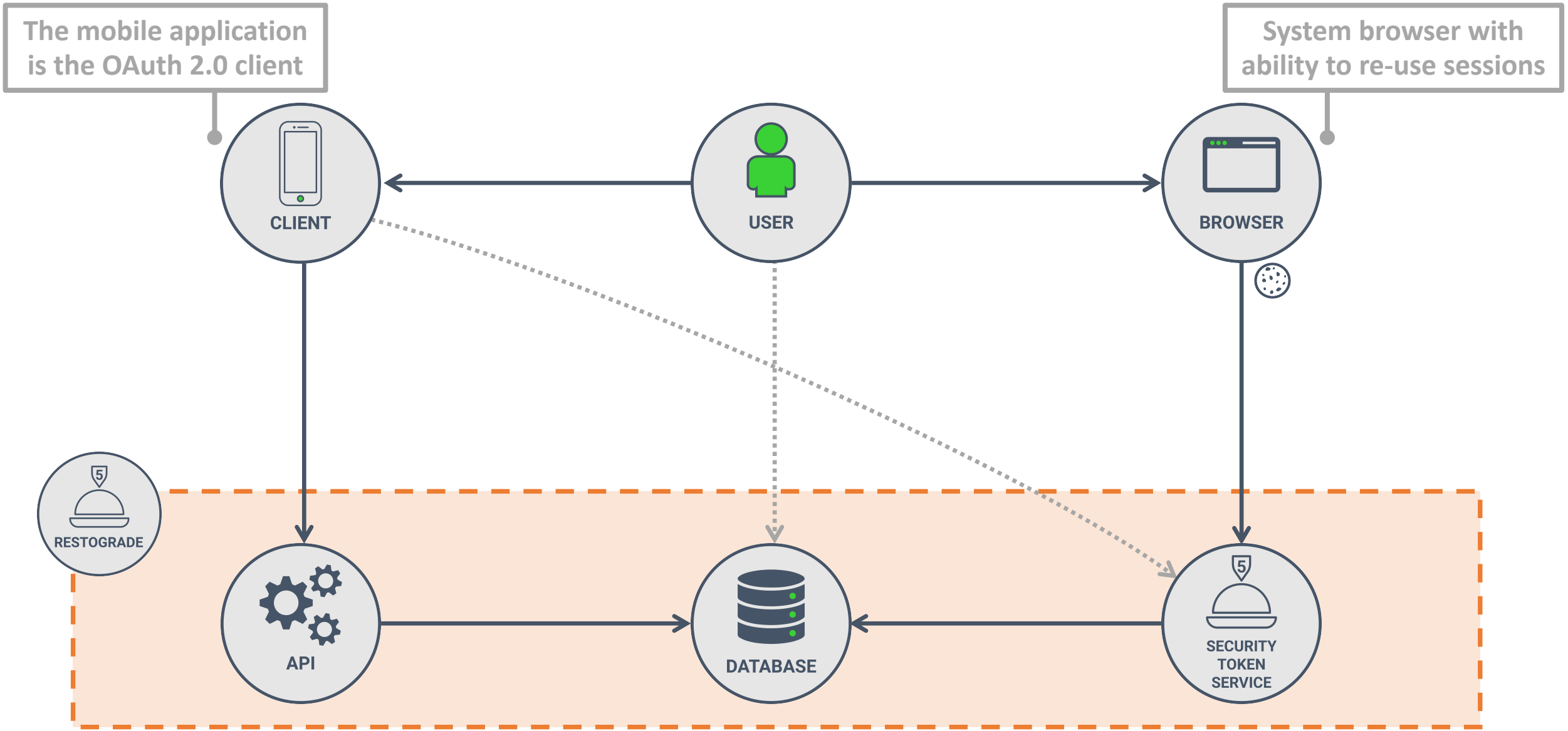
Authorization Code flow

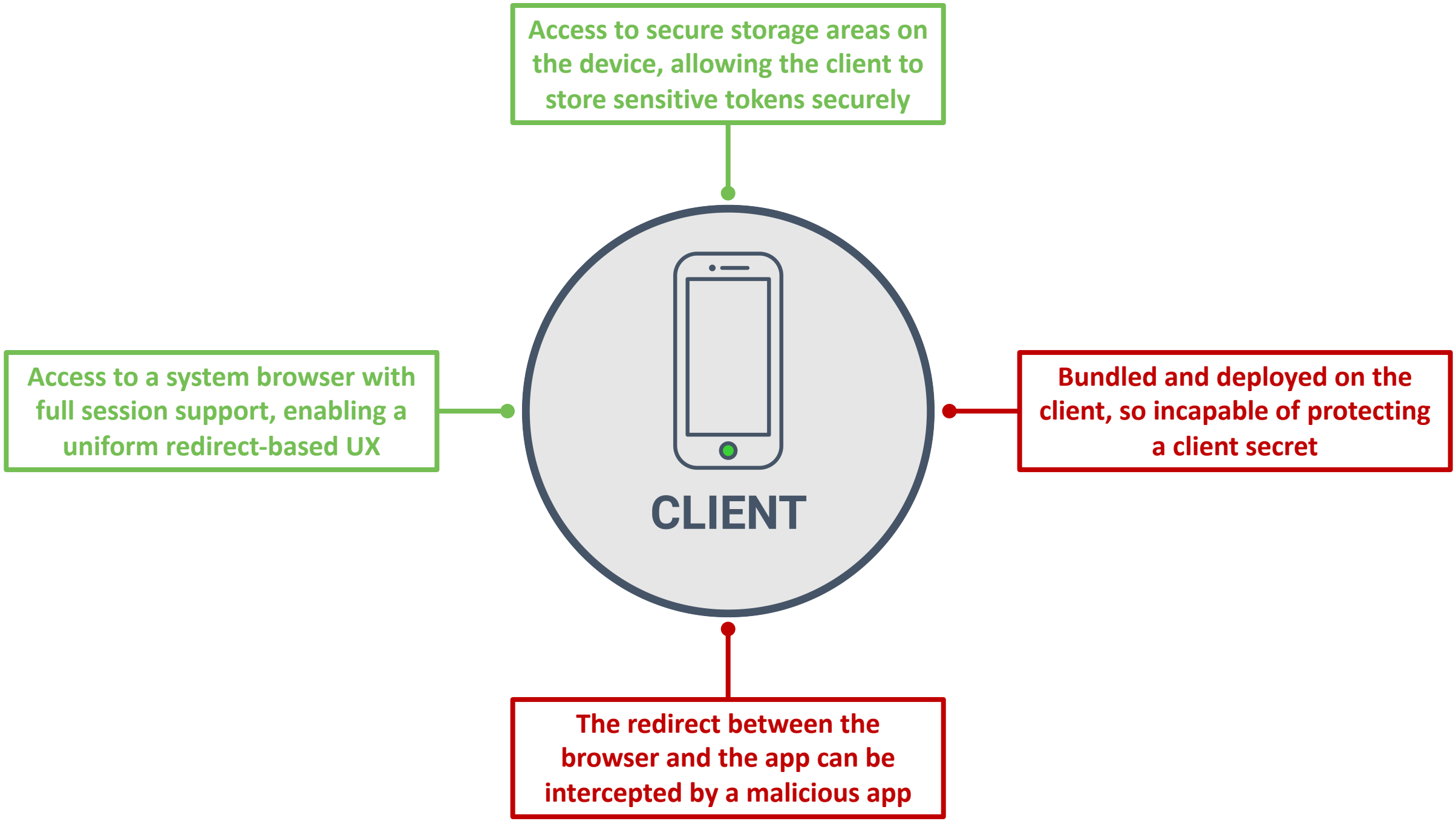
Refresh tokens with client authentication

Encrypt tokens in storage

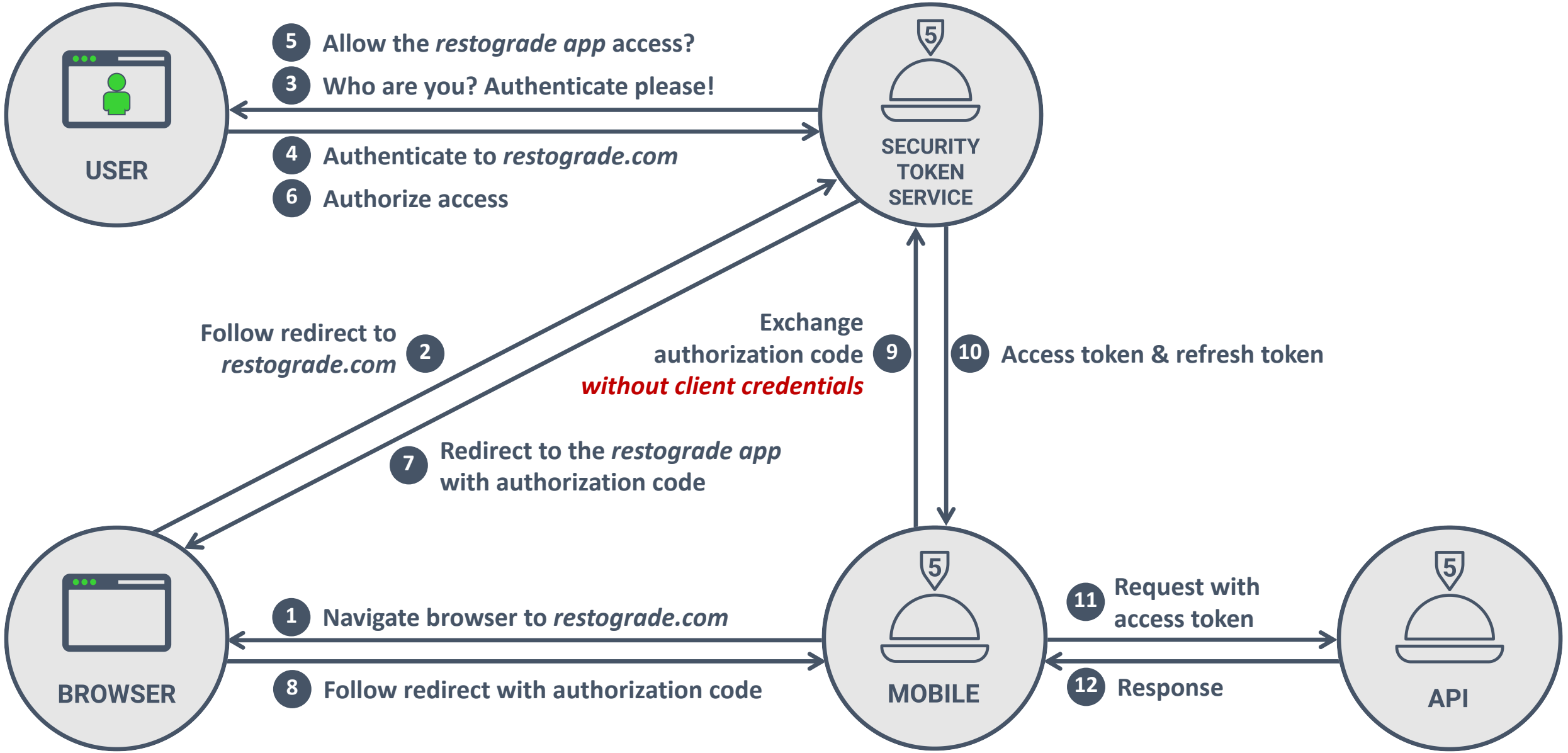
Mobile clients

A NATIVE CLIENT SCENARIO WITH A MOBILE APPLICATION

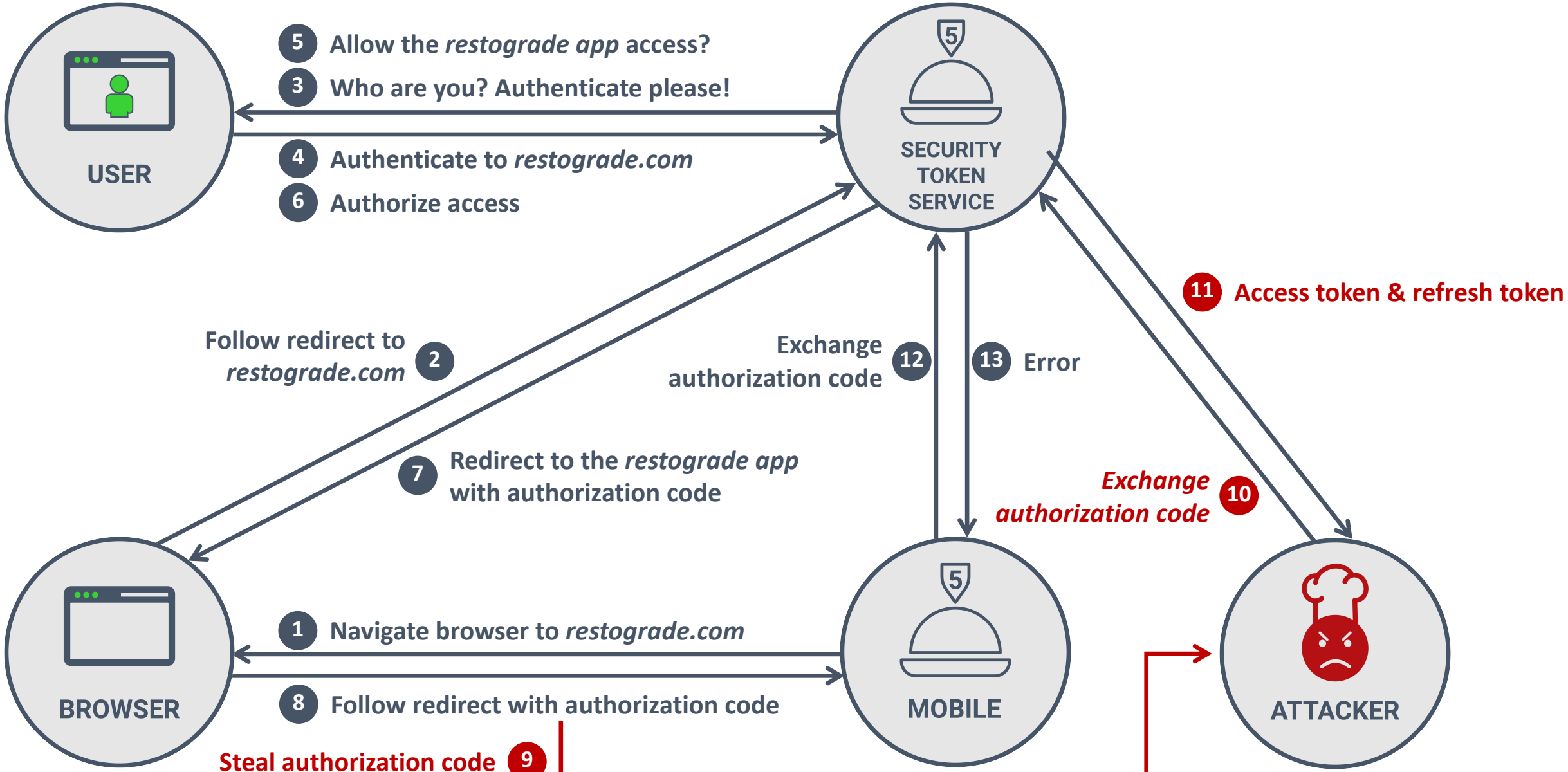



Access to secure storage areas on the device, allowing the client to store sensitive tokens securely
Access to a system browser with full session support, enabling a uniform redirect-based UX
CLIENT
Bundled and deployed on the client, so incapable of protecting a client secret
The redirect between the browser and the app can be intercepted by a malicious app

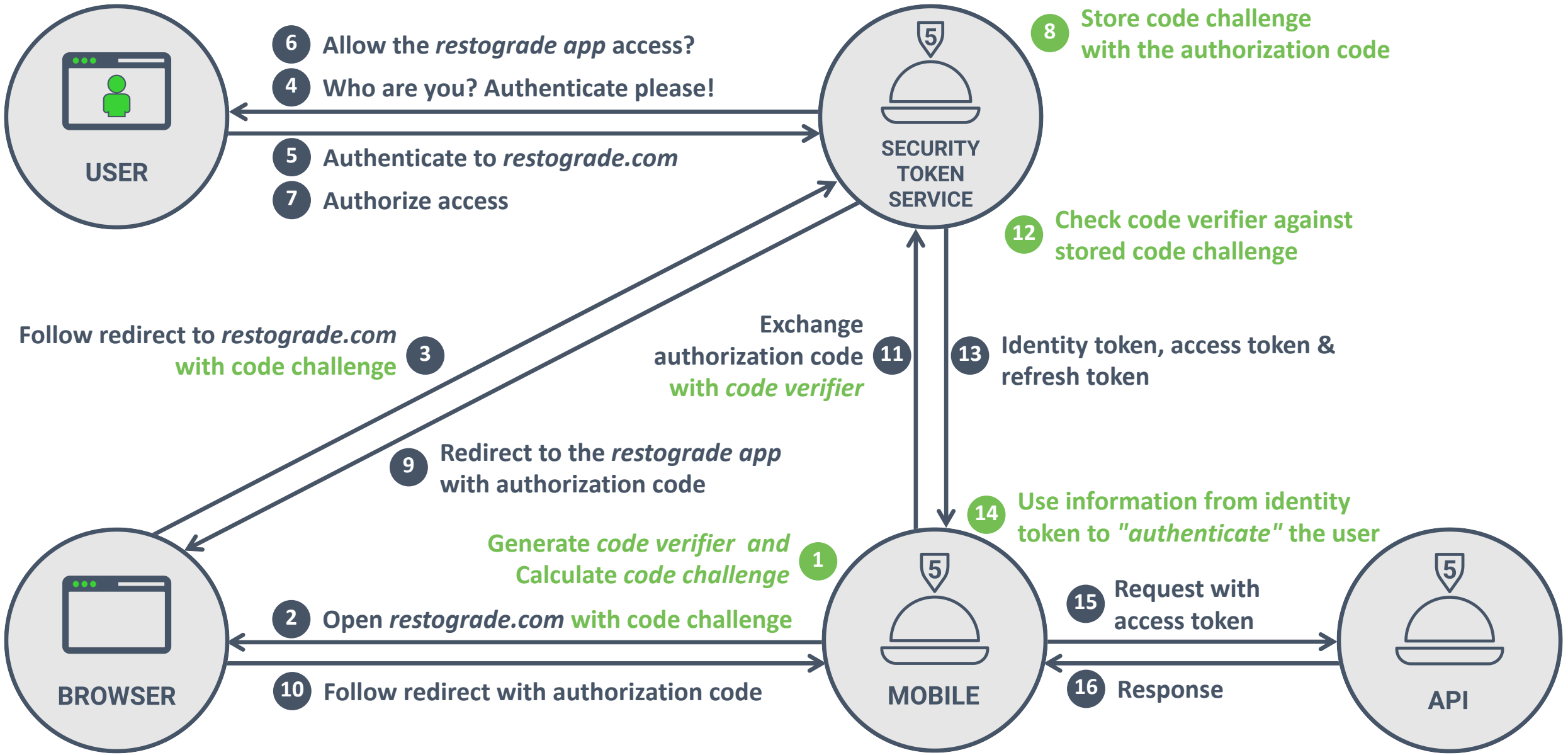
CHALLENGES WITH THE *AUTHORIZATION CODE* FLOW ON MOBILE



CHALLENGES WITH THE *AUTHORIZATION CODE* FLOW ON MOBILE



THE *AUTHORIZATION CODE* FLOW WITH PKCE



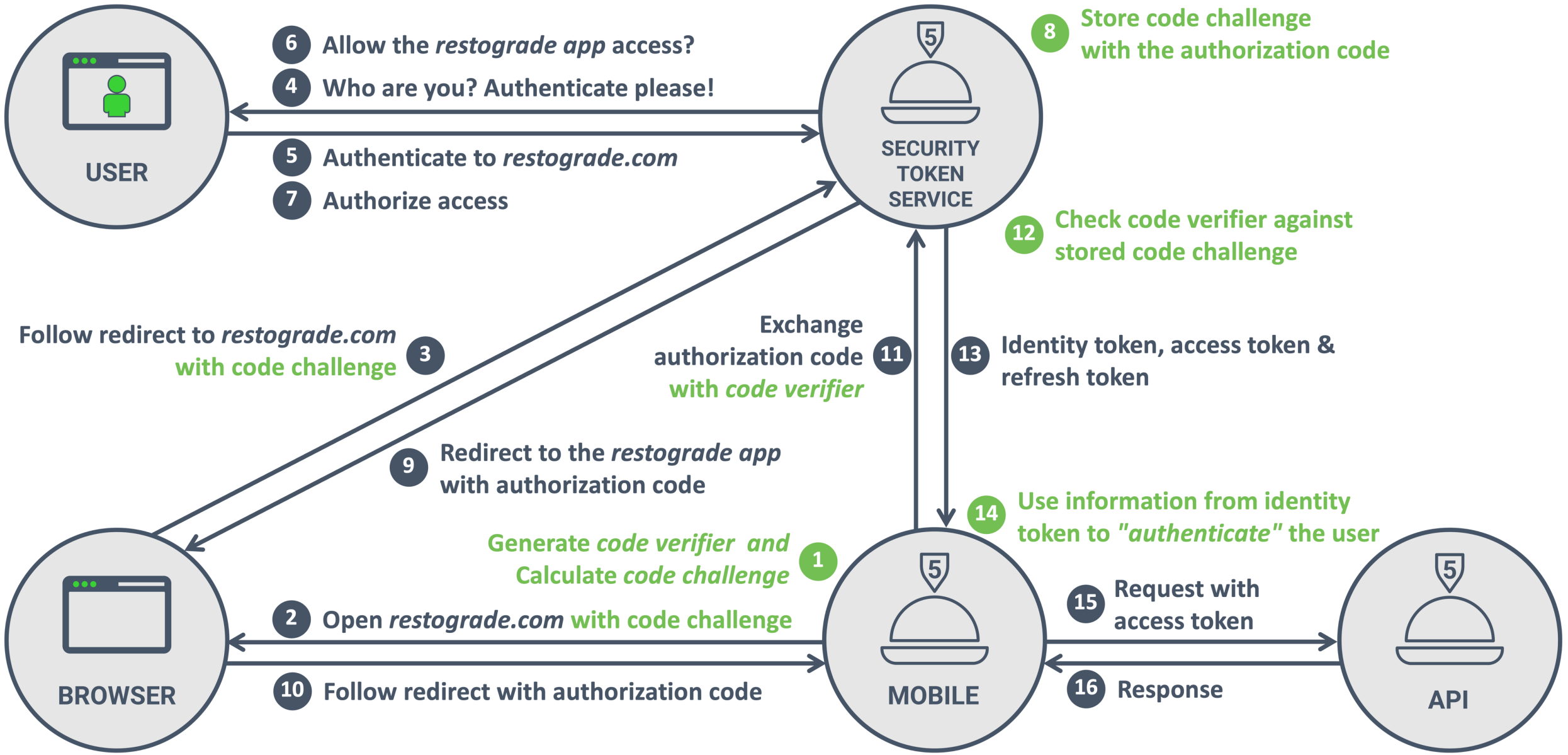
Proof Key for Code Exchange (PKCE)

- PKCE ensures that the same client intializes and finalizes the flow
  - The main use case for PKCE is to prevent authorization code theft in public clients
  - PKCE acts as a one-time password for a particular client instance

- PKCE consists of a code verifier and a code challenge
  - The code verifier is a cryptographically secure random string
    - Between 43 and 128 characters of this character set: ***[A-Z] [a-z] [0-9] - . _ ~***
  - The code challenge is a ***base64 urlencoded SHA256*** hash of the code verifier
    - The hash function uniquely connects the code challenge to the code verifier
    - The code verifier cannot be derived from the code challenge

- PKCE is **mandatory** for native clients to prevent authorization code theft

THE *AUTHORIZATION CODE* FLOW WITH PKCE



2 3 *The redirect URI*

```
1  https://sts.restograde.com/authorize
2    ?response_type=code
3    &client_id=lY5g0BKB7Mow4yDlb6rdGPsO2i1g7Osv
4    &scope=read
5    &redirect_uri=https://app.restograde.com/callback
6    &state=s0wzojm2w8c23xzprkk6
7    &code_challenge=JhEN0Amnj7B…Wh5PxWitZYK1woWh5PxWitZY
8    &code_challenge_method=S256
```

Indicates the *authorization code flow*

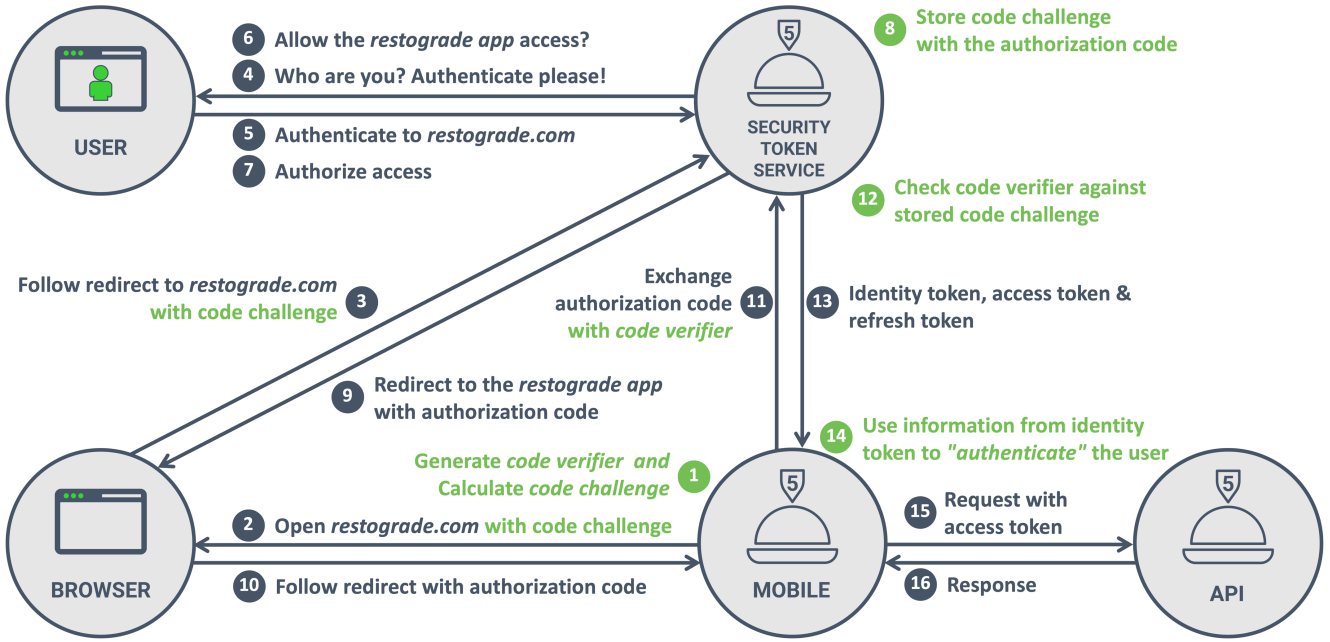
The client requesting access

Where the STS should send the token

The PKCE code challenge

The PKCE hash function

THE *AUTHORIZATION CODE* FLOW WITH PKCE

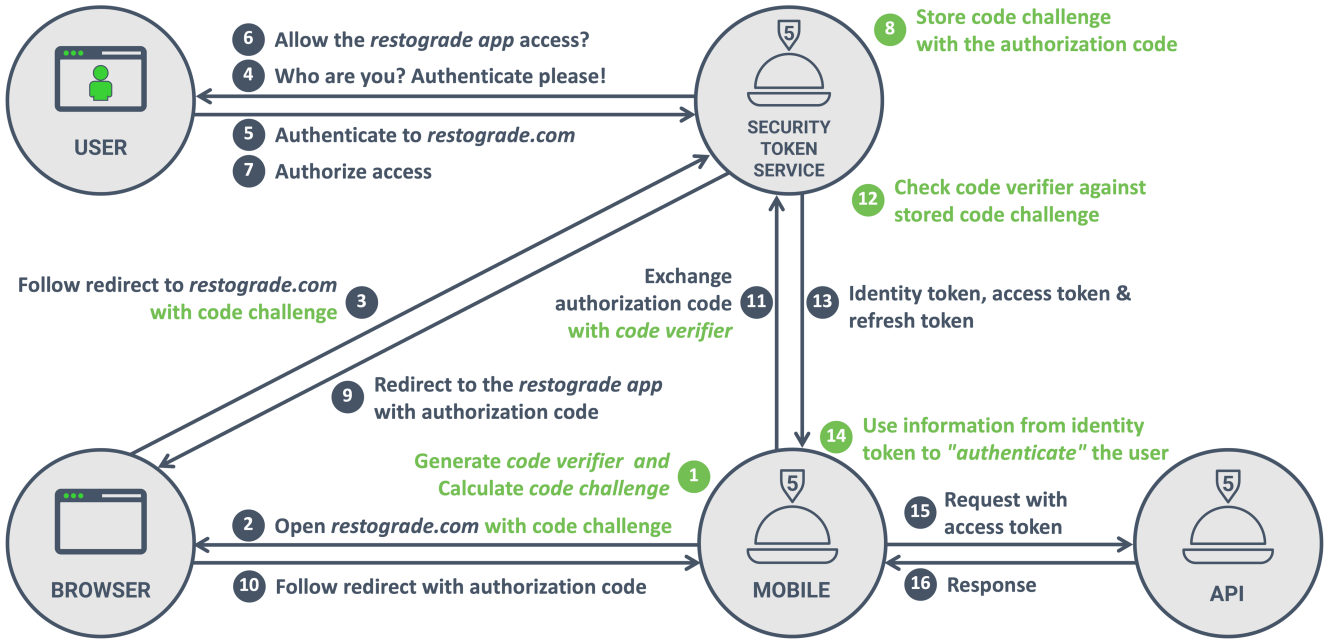


10 *The request to exchange the authorization code*

```
1  POST https://sts.restograde.com/oauth/token
2
3   grant_type=authorization_code
4  &client_id=lY5g0BKB7Mow4yDlb6rdGPsO2i1g7Osv
5  &redirect_uri=https://app.restograde.com/callback
7  &code=SplxlOBeZQQYbYS6WxSbIA
8  &code_verifier=lT5q6nbPQRtdj…~IUdkErVDFG.fF4z7CzCxo
```

- `grant_type=authorization_code` — Indicates the code exchange request
- `&client_id=lY5g0BKB7Mow4yDlb6rdGPsO2i1g7Osv` — The client exchanging the code
- `&redirect_uri=https://app.restograde.com/callback` — The redirect URI used before
- `&code=SplxlOBeZQQYbYS6WxSbIA` — The code received in step 9
- `&code_verifier=lT5q6nbPQRtdj…~IUdkErVDFG.fF4z7CzCxo` — The code verifier from step 1

THE *AUTHORIZATION CODE* FLOW WITH PKCE



Running an *Authorization Code* flow with PKCE

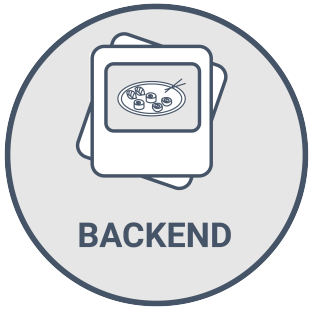
THE *AUTHORIZATION CODE* FLOW WITH PKCE

- The client is a public native application, running on a user device
  - OAuth 2.0 enables the user to authorize the client to access APIs on their behalf
  - OIDC allows the client to obtain authentication information about the user
  - The use of PKCE is crucial to tie the flow to one particular client instance
  - Clients should use ***claimed HTTPS URLs*** to restrict the redirect mechanism

- Native clients should always run the flow in a system browser
  - Either a full browser, or the integrated *SFSafariViewController* / *Chrome Custom Tabs*

- Public clients are allowed to obtain refresh tokens from the STS
  - Public clients have no credentials, so refresh tokens can be used without auhtentication
  - Refresh tokens must be stored securely on the device
    - Token encryption and the use of isolated storage are important to protect the tokens

OVERVIEW OF BEST PRACTICES



Authorization Code flow

Refresh tokens with client authentication

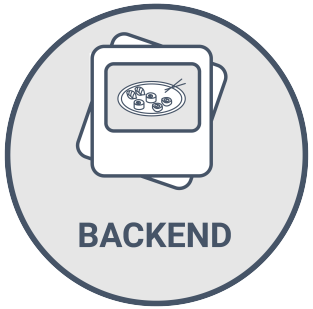
Encrypt tokens in storage

Authorization Code flow with PKCE

Refresh tokens without client authentication

Encrypt tokens in storage

OVERVIEW OF BEST PRACTICES



Authorization Code flow with PKCE

Refresh tokens with client authentication

Encrypt tokens in storage

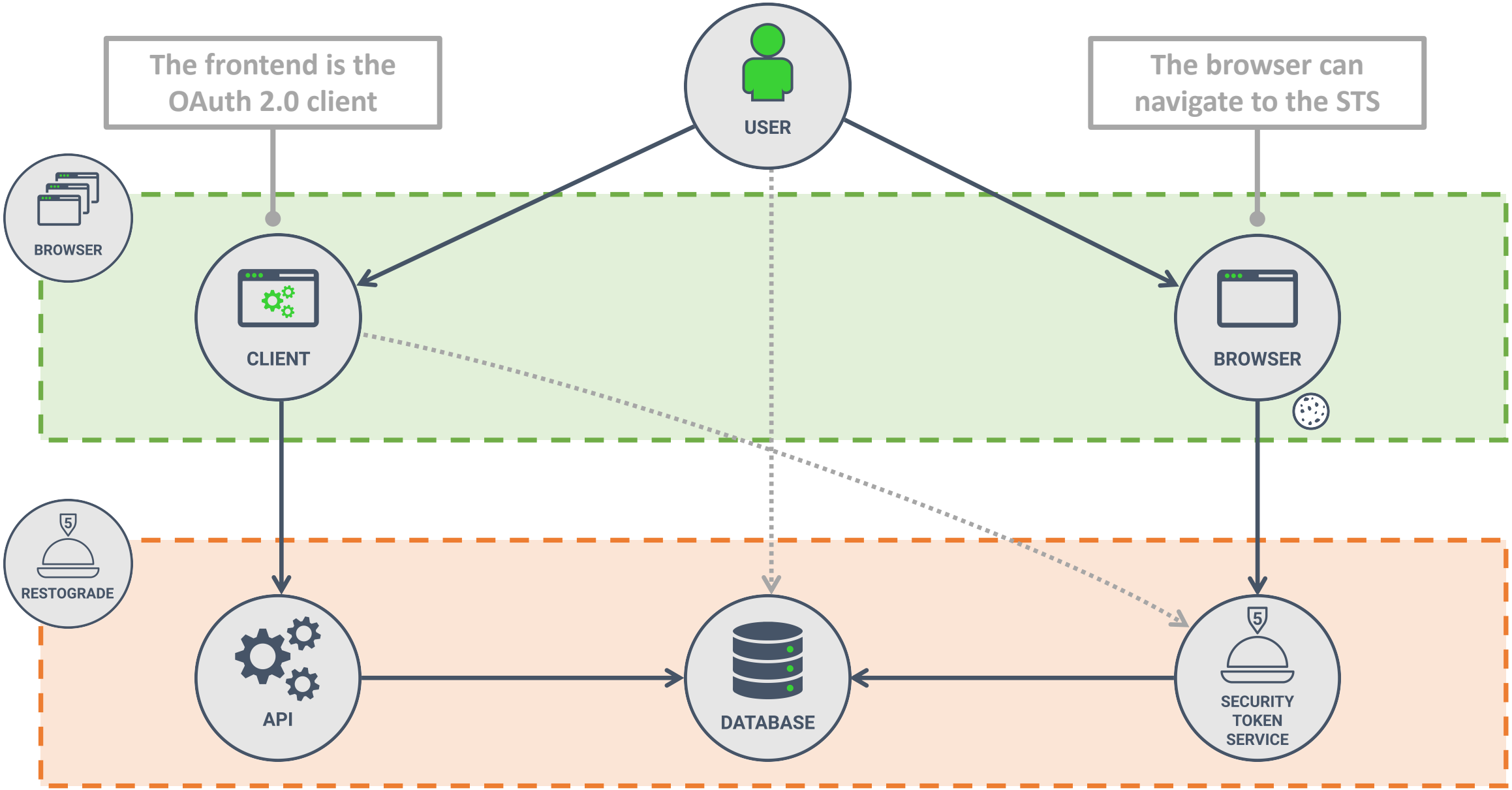
Authorization Code flow with PKCE

Refresh tokens without client authentication

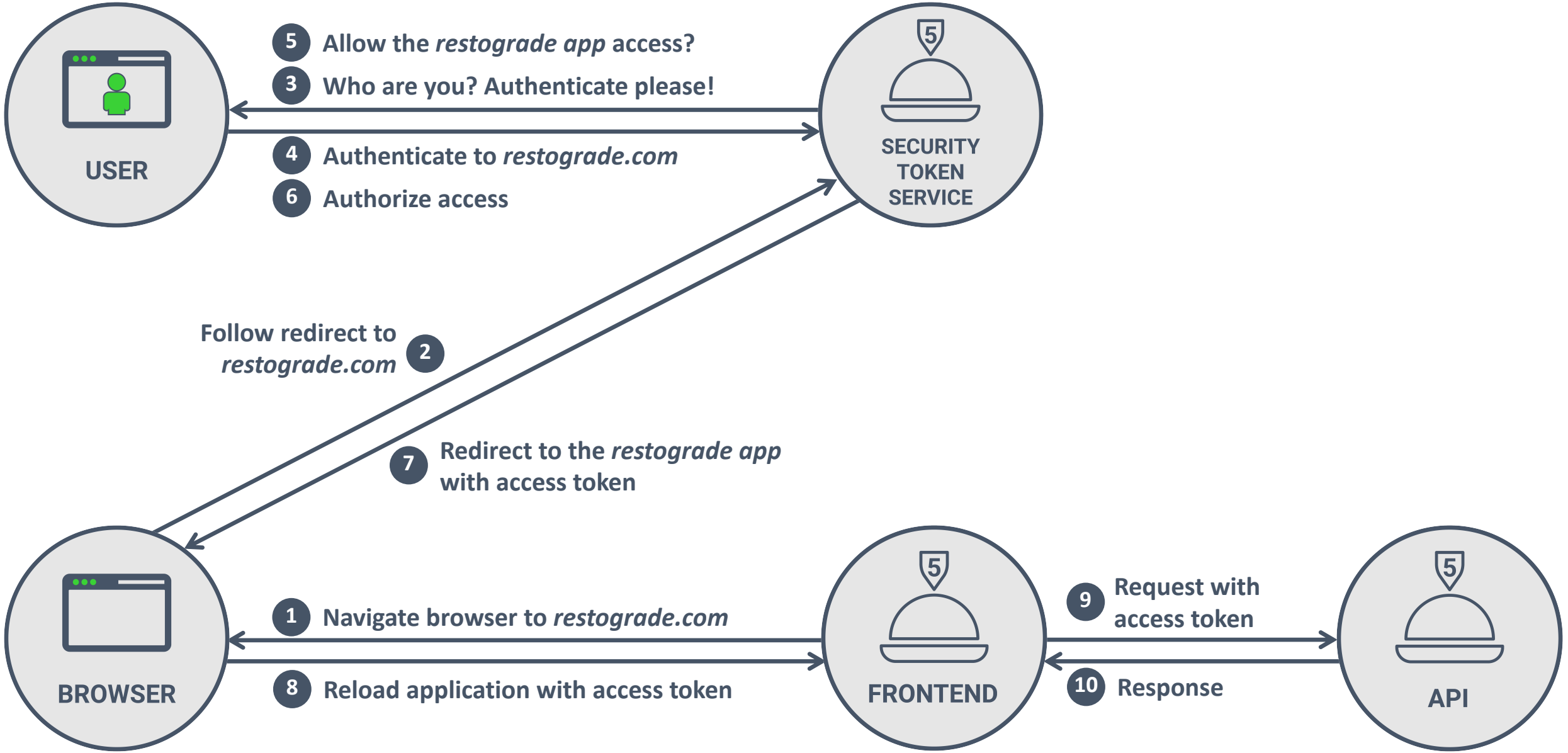
Encrypt tokens in storage

Frontend web clients

A NATIVE CLIENT SCENARIO WITH A MOBILE APPLICATION

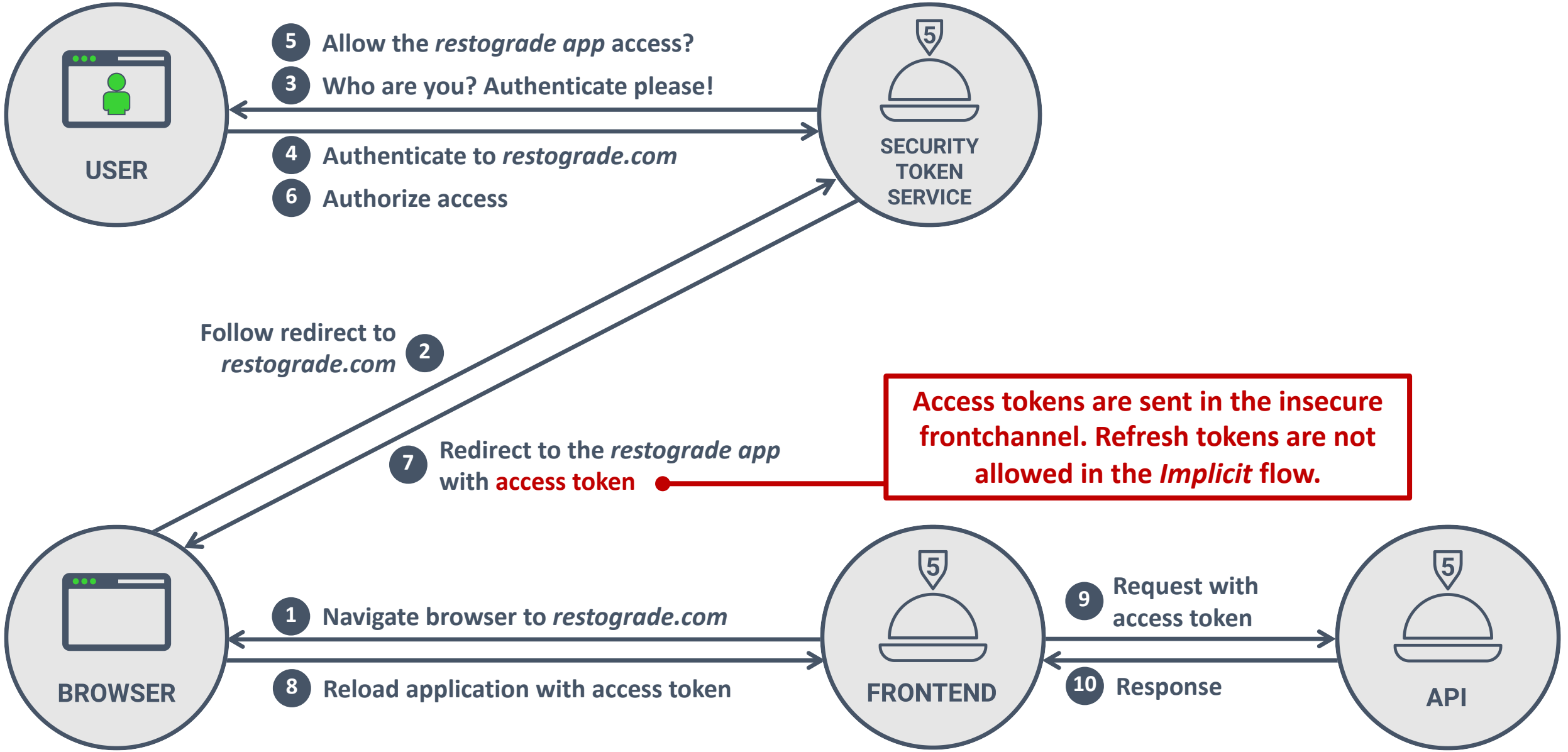


THE *IMPLICIT* FLOW

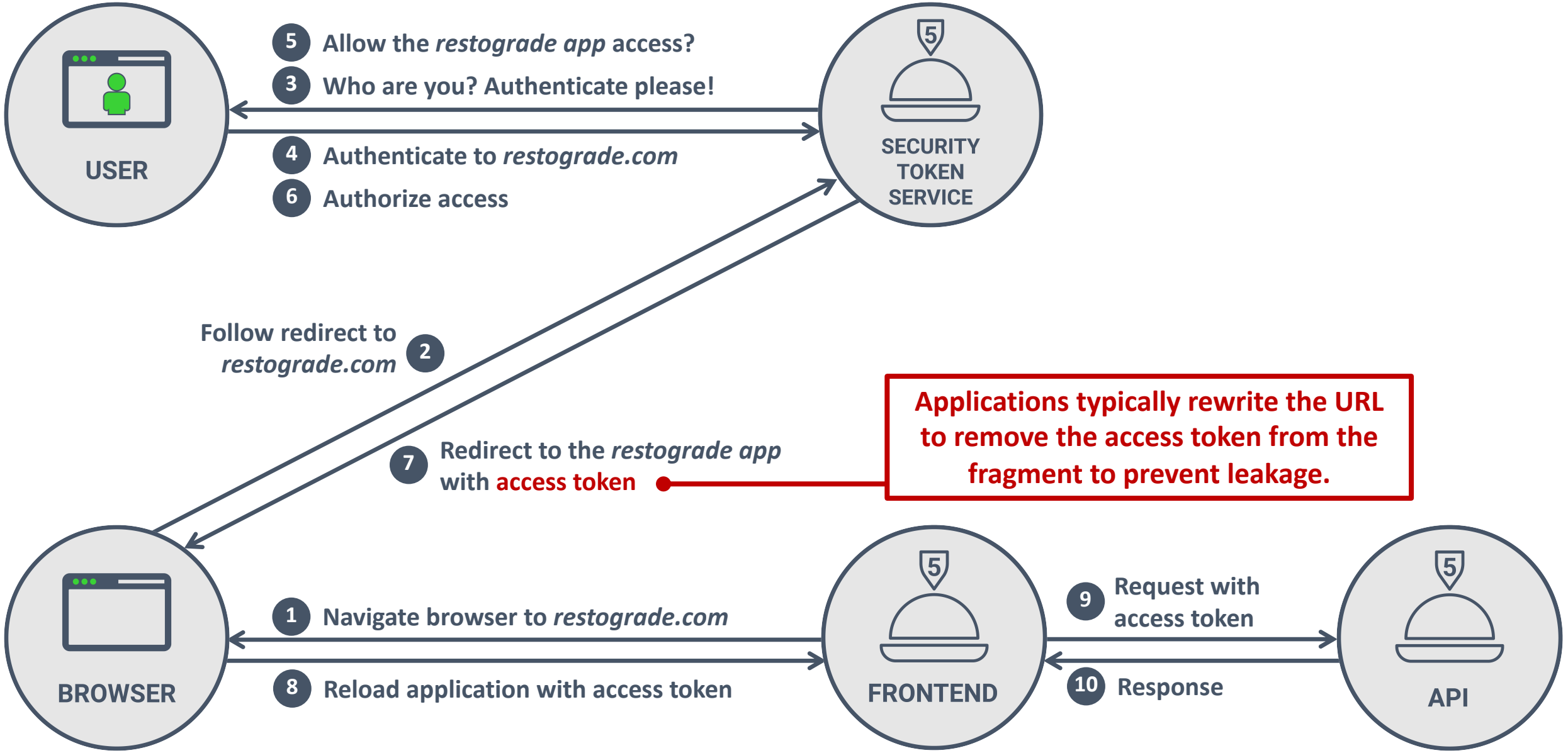


Running the *Implicit* flow

THE *IMPLICIT* FLOW

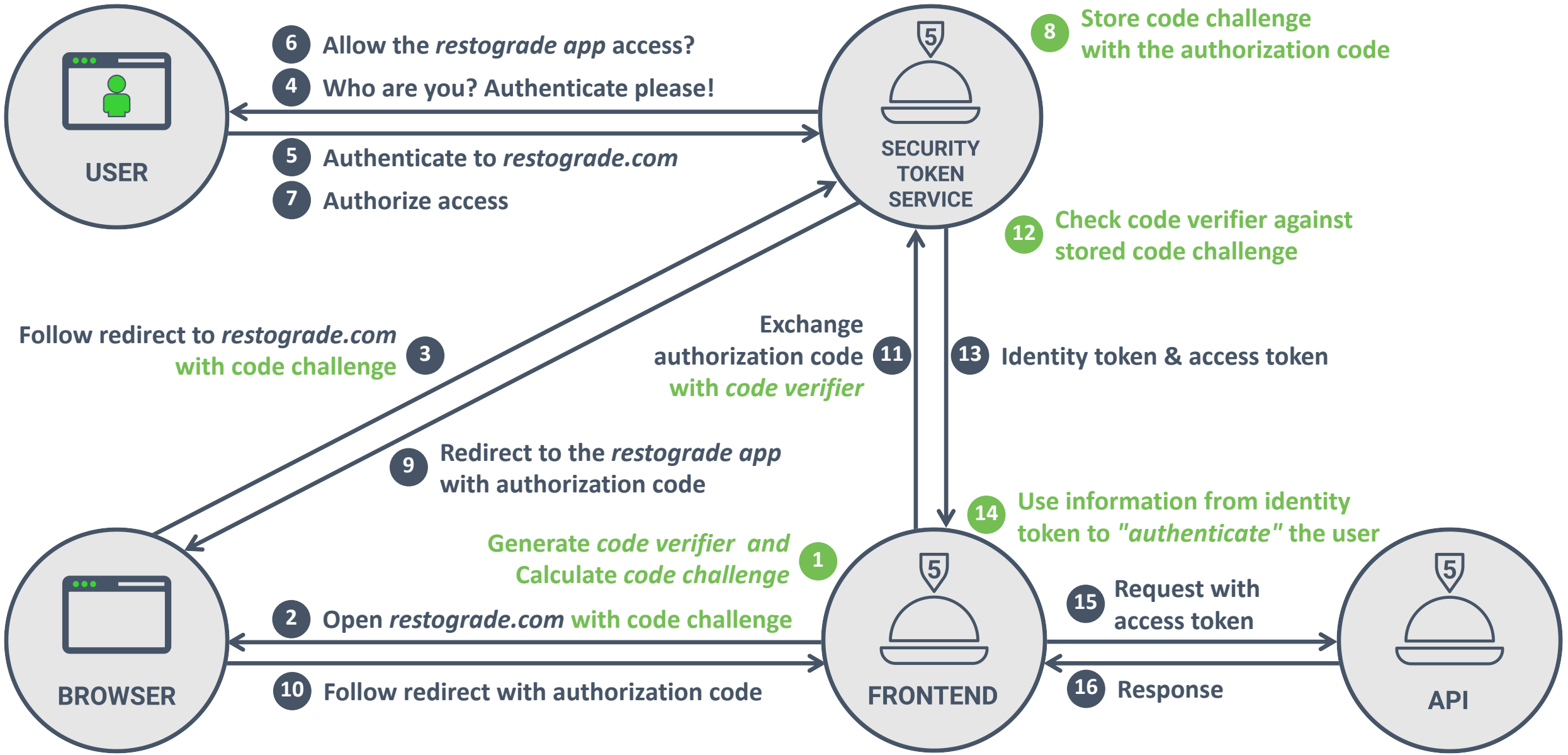


THE *IMPLICIT* FLOW



The OAuth 2.0 Security Best Practices and OAuth 2.1 specifications deprecate the *Implicit* flow

THE *AUTHORIZATION CODE* FLOW WITH PKCE



Comparing the *Implicit* flow to the *Authorization Code* flow

THE *AUTHORIZATION CODE* FLOW WITH PKCE

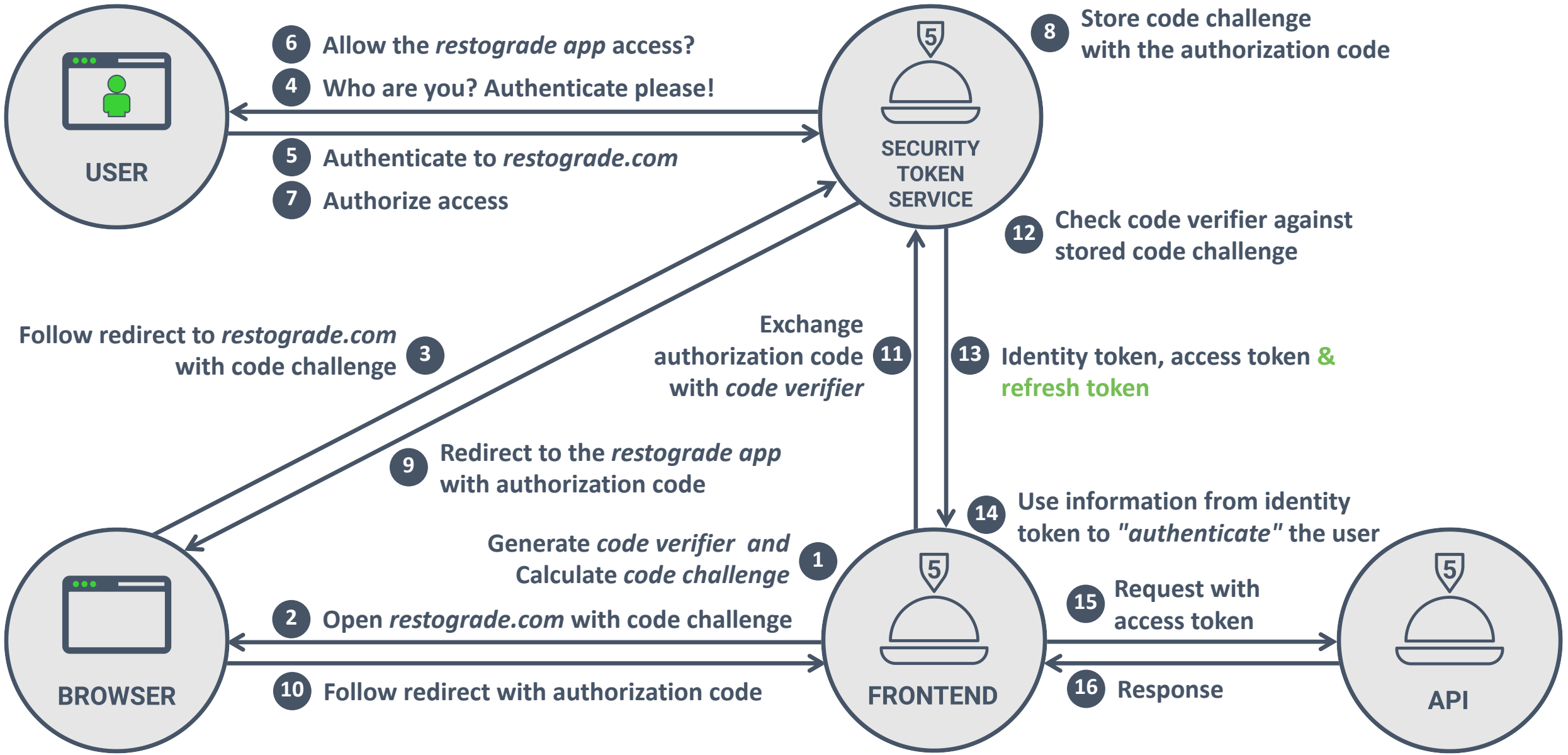
- The client is a public frontend web application, running in the user's browser
  - OAuth 2.0 enables the user to authorize the client to access APIs on their behalf
  - OIDC allows the client to obtain authentication information about the user
  - Both modern Single Page Applications as legacy JS pages can use this new flow
  - The use of PKCE is less crucial than for mobile applications, but still mandatory
- Web frontends have no access to secure storage areas
  - Even if secure storage was available, JS code needs the access token for making requests
  - The execution of malicious JavaScript code is a common attack vector
    - E.g., credit card skimming malware (Magecart) is often malicious JavaScript code
  - When malicious JavaScript code executes, it can extract the access token
- Additional security requirements are needed for frontend OAuth 2.0 clients

Additional security considerations for frontends

- Prevent the execution of malicious JavaScript code at all costs
  - The moment the attacker can execute JavaScript code, it's game over for your application
  - Common attack vectors are Cross-Site Scripting and remote JavaScript inclusions
  - Consider a defense-in-depth strategy (Content Security Policy, Subresource Integrity, …)
- Do not overestimate the security of your storage solution
  - LocalStorage is easy to access and often considered insecure
  - SessionStorage and in-memory are common alternatives with less exposure
  - However, all of them can be attacked through JavaScript
- The lifetime of access tokens should be kept as short as feasible
  - Shorter token lifetimes reduce the potential window of abuse for a stolen token

So what about refresh tokens in browser-based applications?

THE *AUTHORIZATION CODE* FLOW WITH PKCE

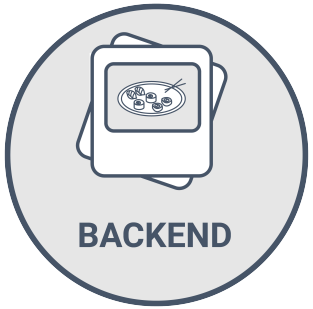


Refresh tokens in the browser

- Refresh tokens in the browser are a new relaxation in the OAuth 2.0 specs
  - Before, frontends had to rely on a *silent renew* flow to obtain new access tokens
  - Refresh tokens offer a clean solution, but also pose a significant security risk
- A single XSS vulnerability can result in a stolen refresh token
  - Refresh tokens for public clients do not require client authentication when used
  - The attacker can abuse a stolen refresh token to keep requesting new access tokens
- **Refresh tokens in browser applications require Refresh Token Rotation**
  - Each refresh token can only be used once to obtain a new access token and refresh token
  - Double use of a refresh token triggers alarms and prevents future abuse
- Sensitive frontends can use a *Backend for Frontend* to keep tokens out of the browser

Refresh Token Rotation in action

OVERVIEW OF BEST PRACTICES



Authorization Code flow with PKCE

Refresh tokens with client authentication

Encrypt tokens in storage

Authorization Code flow with PKCE

Refresh tokens without client authentication

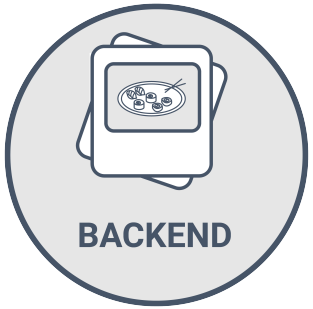
Encrypt tokens in storage

Authorization Code flow with PKCE

Refresh tokens with refresh token rotation

Prevent malicious JS code

OVERVIEW OF BEST PRACTICES



Authorization Code flow with PKCE

Refresh tokens with client authentication

Encrypt tokens in storage

Authorization Code flow with PKCE

Refresh tokens with refresh token rotation

Encrypt tokens in storage

Authorization Code flow with PKCE

Refresh tokens with refresh token rotation

Prevent malicious JS code

Additional OAuth 2.0 flows

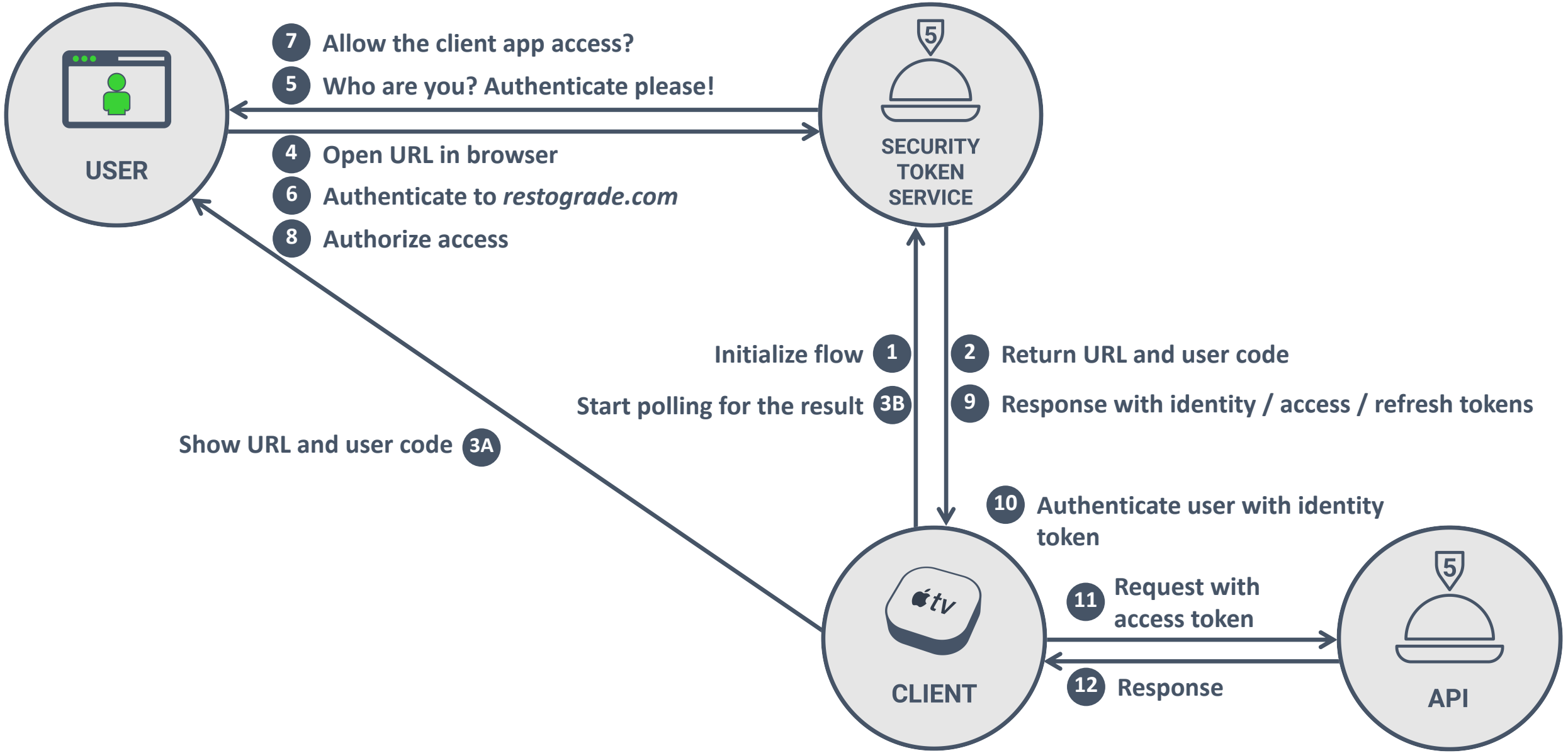
OAuth 2.0 and OIDC are in constant evolution

- The original OAuth 2.0 specification defined four flows
  - The *Authorization Code* flow: still good, but augmented with PKCE in a later specification
  - The *Implicit* flow: no longer useful, so phasing out
  - The *Resource Owner Password Credentials* flow: a bad practice, so has to be avoided
  - The *Client Credentials* flow: enables machine-to-machine communication

- Additional specifications define new mechanisms or refine the details of flows
  - One noteworthy specification is the *Device Authorization Grant*, or the ***Device* flow**

- The main flow for a user delegating access is the *Authorization Code* flow
  - Both confidential and public clients can use this flow with PKCE

THE *DEVICE* FLOW




AuthU TV
To activate this smart TV to stream directly from AuthU TV
1. On your computer or mobile device, go to: acme-demo.auth0.com/activate
2. Enter the following code: VVGG-QQKL
Or scan the QR code below

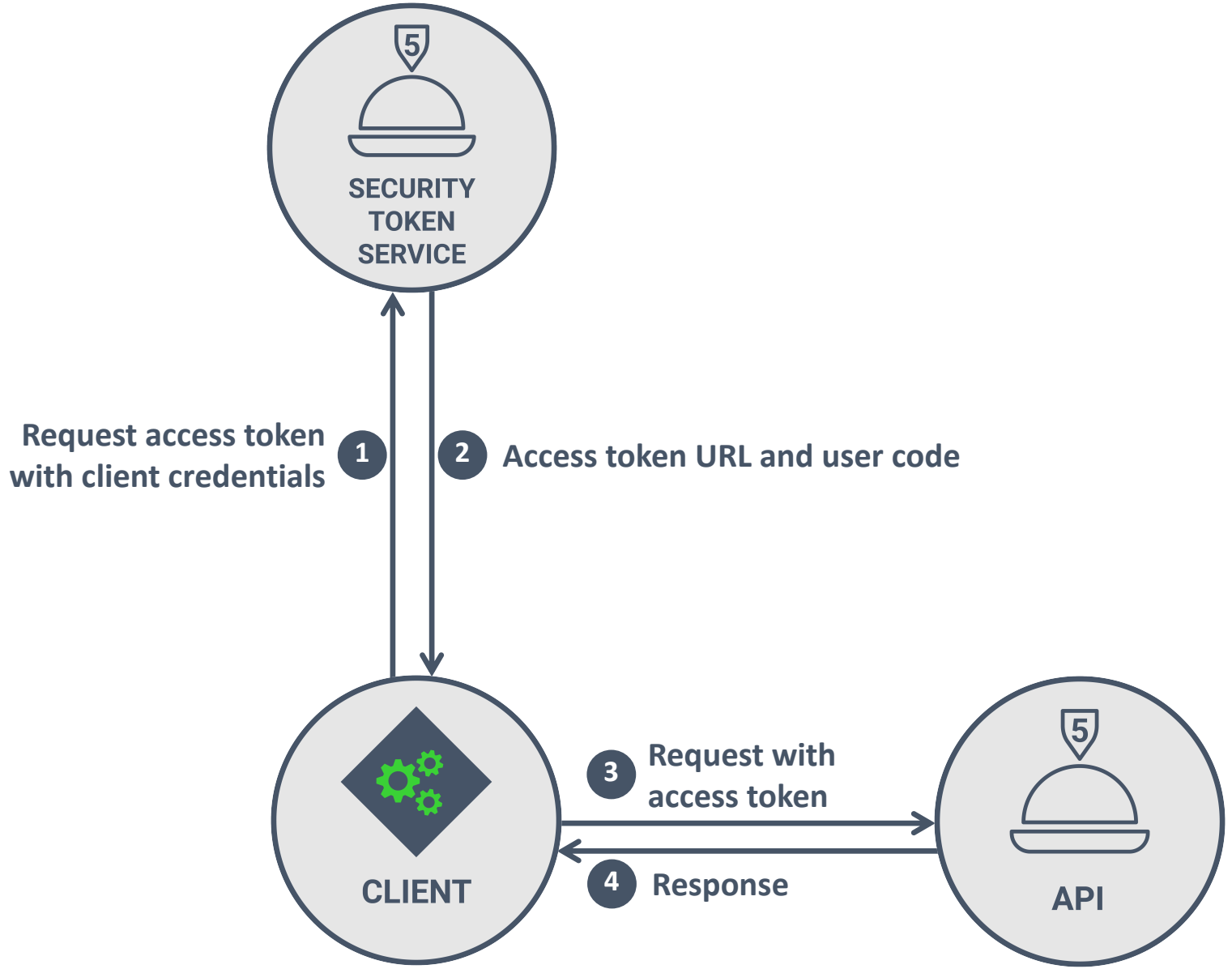
THE *DEVICE* FLOW

- The *Device* flow decouples the OAuth 2.0 flow from the user interaction
  - Intended for devices that cannot run a browser or have constrained input capabilities
    - E.g, smart TVs, kiosk systems, …
  - The device with the brower can handle user input and advanced input mechanisms
    - Enables the use of password managers and multi-factor authentication

- The client in the *Device* flow is still a normal client application
  - It can receive anm identity token, access token, and refresh token
  - The client is responsible for storing tokens securely

- The *Device* flow also increases security for the user
  - No need to enter credentials on a potentially untrusted / shared device
  - Access can always be revoked through the STS

THE *CLIENT CREDENTIALS* FLOW



THE *CLIENT CREDENTIALS* FLOW

- The *Client Credentials* flow enables machine-to-machine communication
  - There is no delegated access, because there is no user involved in this flow
  - The *Client Credentials* flow allows APIs to use a uniform authorization mechanism
- The only relevant token in the *Client Credentials* flow is the access token
  - There is no user authentication, so no need for an identity token
  - The can always re-run the flow with its credentials, so no need for a refresh token
- The *Client Credentials* flow is often combined with strong authentication
  - Clients authenticate with cryptographic credentials, such as a TLS certificate
    - The client uses mTLS to authenticate to the STS and to the API when making requests
  - The STS uses the access token to convey client permissions to the API

Wrapping up

OAuth 2.0 and OpenID Connect

- OAuth 2.0 allows a user to delegate access to a client application
  - Avoids the need for sharing credentials with the client application
  - Defines an authorization framework to allow APIs to make authorization decisions
  - OAuth 2.0 is the de facto standard for implementing distributed authorization scenarios

- OpenID Connect allows a client to delegate authentication to a central provider
  - OIDC is the de facto standard for building modern Single Sign-On systems
  - OIDC uses OAuth 2.0 flows with specific configuration settings
  - OAuth 2.0 and OIDC are typically used together, but can be used separately as well

- How the user authenticates to the central provider is not specified
  - OAuth 2.0 and OIDC define the interactions between the different components

THREE FLOWS TO REMEMBER

- The *Authorization Code* flow with PKCE
  - For scenarios where the user is delegating access to a client application
  - Recommended best practice for confidential and public clients
  - Requires the application and the system browser running on the same device
- The *Device* flow
  - Enables a user to delegate access to a client running on an input-constrained device
  - Requires explicit user interaction, so no automatic login
- The Client Credentials flow
  - For scenarios where there is no user, but only code
  - A uniform way to integrate authorization with machine-to-machine communication

IT'S ALL ABOUT TOKENS

- OAuth 2.0 and OIDC use three different token types and one code
  - ***Access token***: a token that represents a client's authority to access an API
  - ***Refresh token***: a token to retrieve new tokens from the STS without user interaction
  - ***Identity token***: a token with information about the user's authentication with the STS
  - ***Authorization code***: an intermediate artifact to exchange for tokens

- Access tokens and refresh tokens are extremely sensitive
  - Applications need to make sure they are properly protected and securely stored
  - If possible, use advanced protection mechanisms, such as token binding

- Identity tokens contain personal information, so they should also be protected

The tip of the iceberg

- OAuth 2.0 and OpenID Connect are extremely complex technologies
  - The security of these technologies depends on small details and nuances
- Rely on SDKs and libraries to integrate these technologies into your application
  - These SDKs and libraries handle most of the details out of the box
  - They save a lot of time and prevent a lot of subtle security vulnerabilities
- Do not attempt to modify or change any of the OAuth 2.0 or OIDC flows
  - Authentication with OAuth 2.0 is hard to implement securely, so use OIDC
  - Respect the target audience of each of the tokens
    - Custom solutions often ship tokens around to other parts of the application
    - Doing so can causes subtle vulnerabilities

What's next?

What's next?

- Diving deeper into OAuth 2.0 and OpenID Connect
  - Session 2 will take a close look at frontend web applications and their challenges
  - Session 3 looks at how APIs handle access tokens and how to implement authorization
  - ***Registration is still available, so don't hesitate to join us***
- Rewatch the recording
  - The recording will be available as a single video tomorrow
  - In the coming weeks, I will process the recording and publish shorter videos
- Spread the word on this course
  - Invite friends and colleagues to watch the recording and join the rest of the course
- Contact me to discuss custom advisory services for your applications

I hope you enjoyed it!

Registration for session 2 and 3 is available,
so I hope to see you again next week!

@PhilippeDeRyck

/in/PhilippeDeRyck