OAuth2 Workshop

Martin Danielsson, martin.danielsson@haufe-lexware.com

GitHub, Twitter: DonMartin76

Agenda

- OAuth2 – What's that?
- Which Problem does OAuth2 solve?
- OAuth2 Roles
- Client Credentials Flow LAB
- Client Types
- Authorization Code Grant LAB
- Scope…? LAB
- Refresh Tokens
- Public Clients
- Implicit Grant
- Implicit Grant – Refreshing Tokens LAB
- Resource Owner Password Grant LAB
- Native/Mobile Apps
- PKCE Extension LAB
- Implementation Options
- Haufe Specialties
- Recap & Further Reading

Disclaimer

- Workshop built up using wicked.haufe.io
- Wicked is just **one implementation** of the OAuth2 Standard
- Most information is applicable to other implementations
  - Some details are left out in the spec
  - Implementations differ from system to system
  - But: OAuth2 is designed to be interoperable

OAuth2 – What's That?

What do **you** think OAuth2 is?

RFC6749

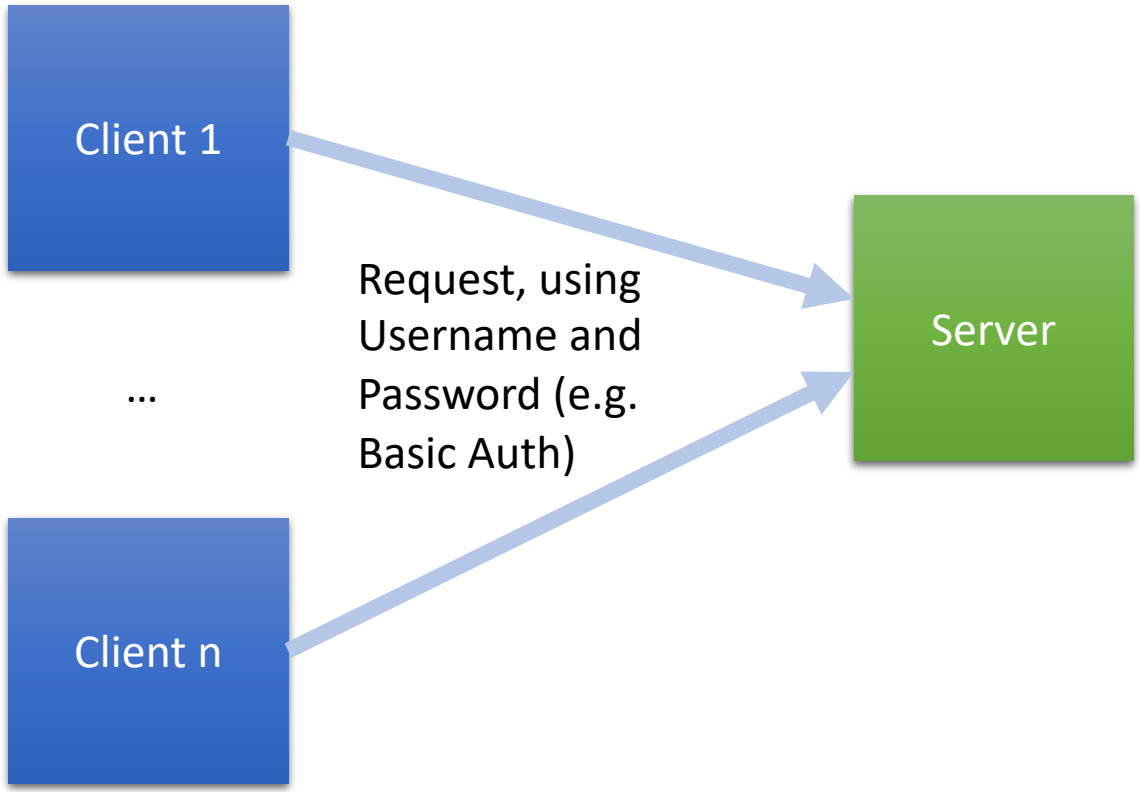
The OAuth 2.0 Authorization Framework

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.

Which Problem does OAuth2 solve?

Typical pre-OAuth2 Client/Server Auth



- Server stores user data
- Clients need username and password to access data

Problems with that approach?

- Clients need to store username and password (or credentials)
- Servers must validate credentials – might enable brute force attacks
- Server cannot distinguish clients – all clients have same type of access
  - Possibly “overly broad”
- End user cannot prevent/revoke specific client access
  - Changing password invalidates access for all clients
- One compromised client results in compromised username and password

Which Problem does OAuth2 not solve?

Authentication

But if you want to, you can mis-use it for it.

OAuth2 Roles

The Key Players in the OAuth2 Game

OAuth2 Roles

Resource Owner	Client (Application)	Resource Server (API)	Authorization Server

• Typically an End User
• Typically authorizes a client to access resources on his behalf
	• Requests access to resources
	• On behalf of
	  • itself, or
	  • of a Resource Owner
		• Serves data to clients
		• Typically an API implementation
		• Makes sure access is valid
		• May involve an API Gateway
			• Establishes identity of
			  • Client, and…
			  • End User (optionally)
			• Decides on access (authorizes)
			• Typically by asking the Resource Owner

The Standard Flows

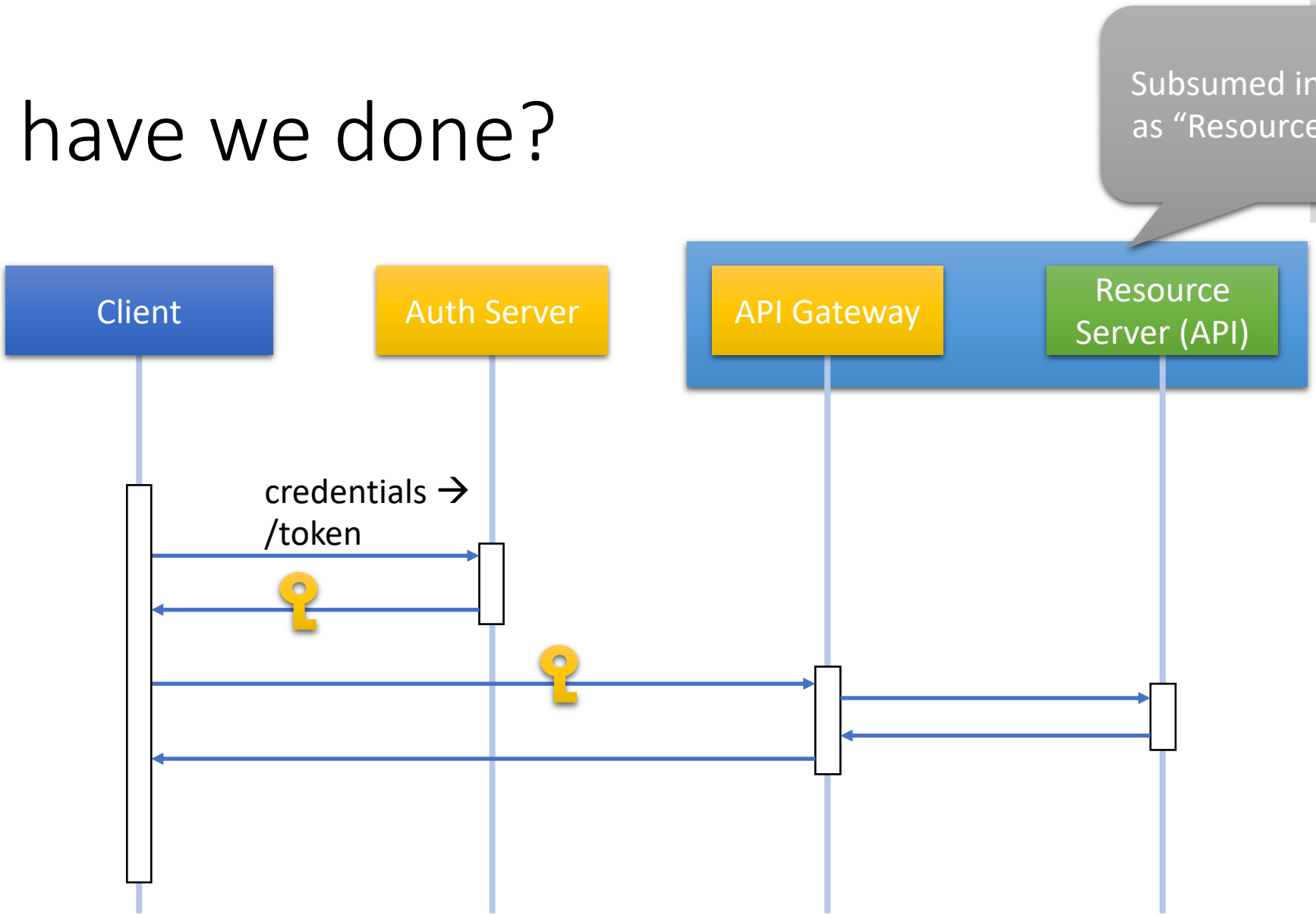
- Four Standard Flows:
  - Client Credentials
  - Authorization Code Grant
  - Implicit Grant
  - Resource Owner Password Grant
- Plus Refresh Token Grant

Client Credentials

Authenticating and Authorizing an Application (Client)

Lab 1

What have we done?



Use Cases Client Credentials Flow

- Machine-to-machine communication
- Non-personal data, or…
- Trusted systems
  - Systems are allowed to operate on any user’s data
  - User identity already established
- Data which is associated with the Client only
  - E.g., configuration for the client

Limitations Client Credentials Flow

- Only authenticates the Client
- The end user (usually the Resource Owner) is not part of flow
- Client must be confidential (more on this later)

Time for questions

Intermezzo: Client Types

OAuth2 Client Types

Confidential

"can keep secrets confidential"

Web Apps/Sites

Web Services

Public

"cannot keep secrets confidential"

Single Page Applications

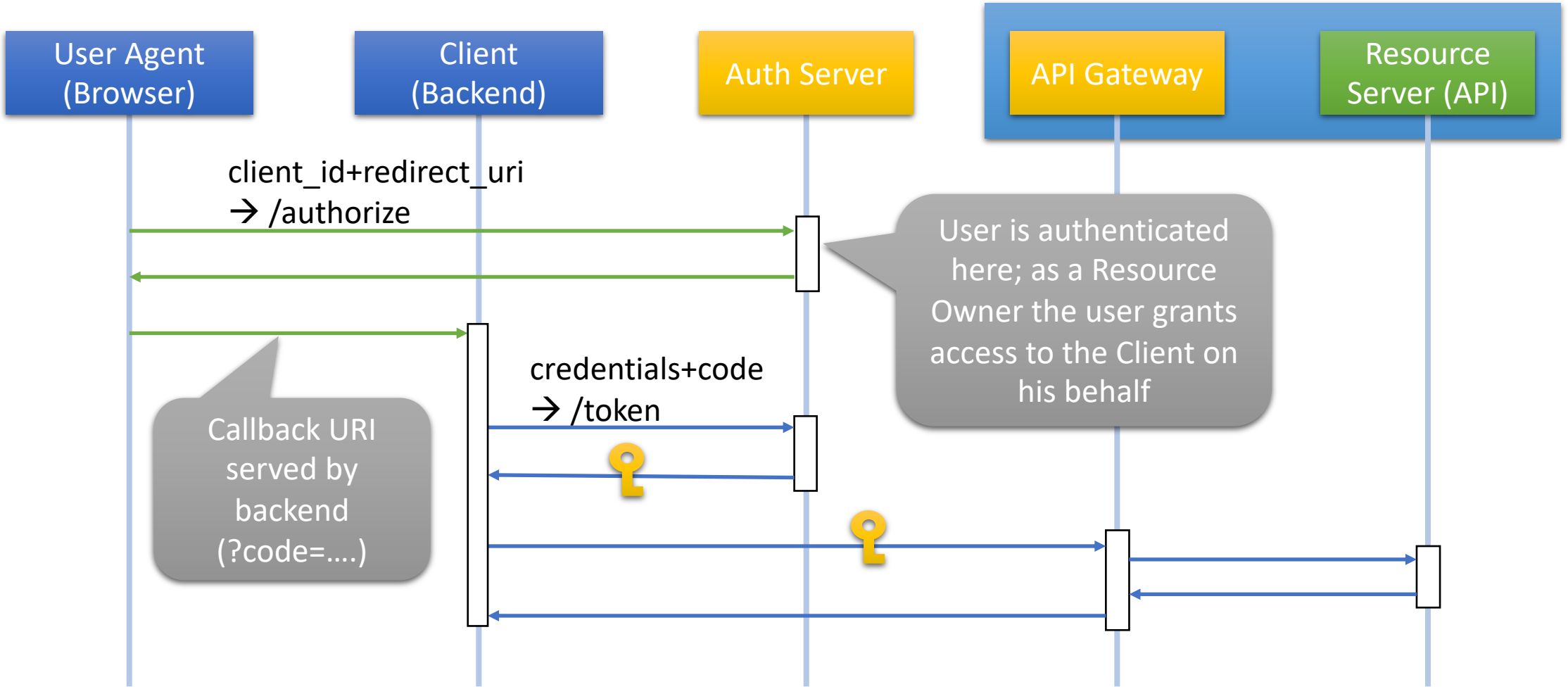
Mobile Apps
(Native Apps)

Authorization Code Grant

The ”Classical” OAuth2 Flow for Confidential Clients

Lab 2

What have we done?



Key Takeaways

- Client Application is registered
  - Credentials can be invalidated
- **The Client never sees the username and password**
- Client uses an “Access Token” with limited scope and validity
- Loss of the Access Token is bad…
  - … but not as bad as losing username and password
  - And it’s short-lived

Scope…?

How OAuth2 solves the “Overly Broad” problem

What is a Scope?

- A scope is whatever you define it to be
- Implementing handling of scopes entirely up to business logic
- Scopes are typically part of documentation

default

GET	/
GET	/api-docs
GET	/users/me
GET	/users/{user_id}
PUT	/users/{user_id}
GET	/users/{user_id}/index
GET	/notes/{note_id}
PUT	/notes/{note_id}
DELETE	/notes/{note_id}

default

GET /

GET /api-docs

GET /users/me

Retrieves the authenticated user's profile. Requires the `read_profile` scope.

Parameters

No parameters

Responses

Code	Description	Links
200	***Successful response***	*No links*

application/json

Controls Accept header.

Scope Recap

- Using Scopes is entirely optional
  - But highly recommended!
- Helps to give Clients access to just what they need
  - E.g. only read-only access
  - Mechanism against “overly broad” access

Time for questions

Lab 3

Refresh Tokens

Prolonging Access Indefinitely

Refresh Token Grant

- Authorization Code Grant also returns a Refresh Token
- Usually stored in Backend
- Used to prolong access to an API on a user’s behalf
- Only uses the token end point – headless

Time for questions

Public Clients

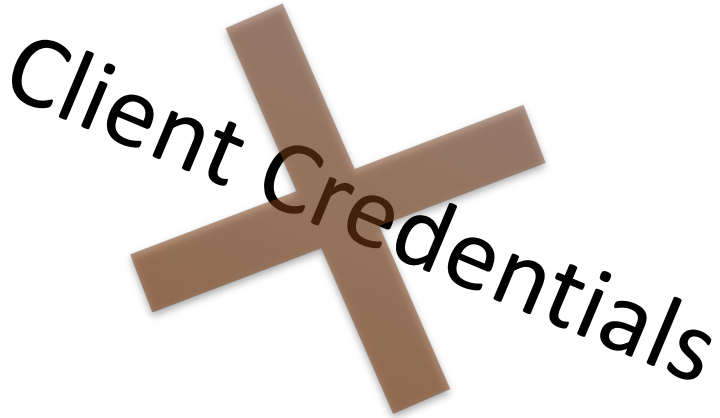
Authorizing Public Clients (Native Apps & Single Page Applications)

Public Clients: Problems

- They cannot keep things to themselves
- SPA – Source code downloaded in browser
- Android APKs: Can fairly easily be decompiled
- iOS App: With Jailbreak, no big deal

So…?

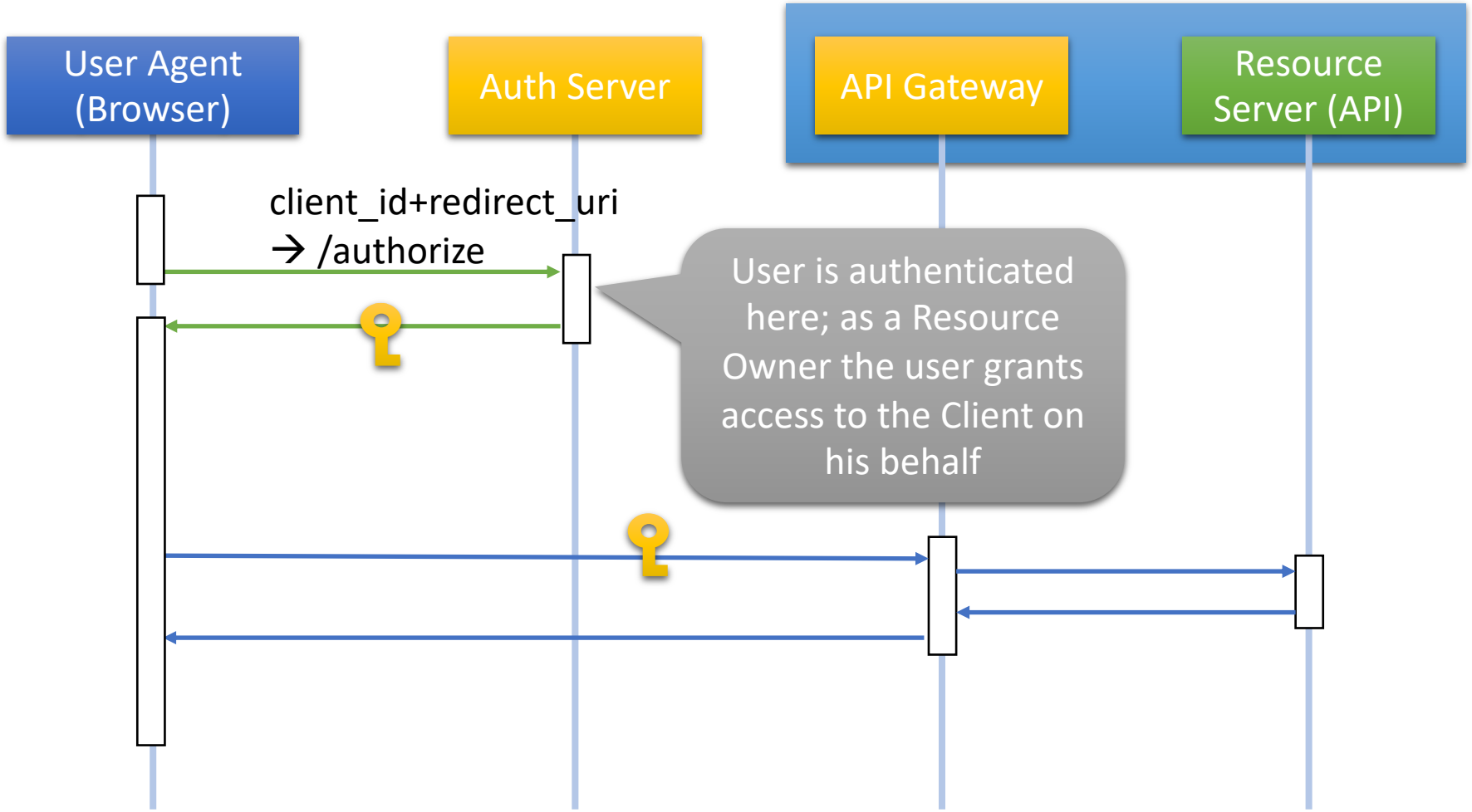
- Too easy to steal a client application's identity
- Refresh tokens can be re-used in other applications
- Client Impersonation



Implicit Grant

Simplified Grant for User Agent Based Applications (Single Page Applications)

How the Implicit Grant works



Hey, wait…?

This is a lot simpler?

Isn’t it?

Yes, **but…**

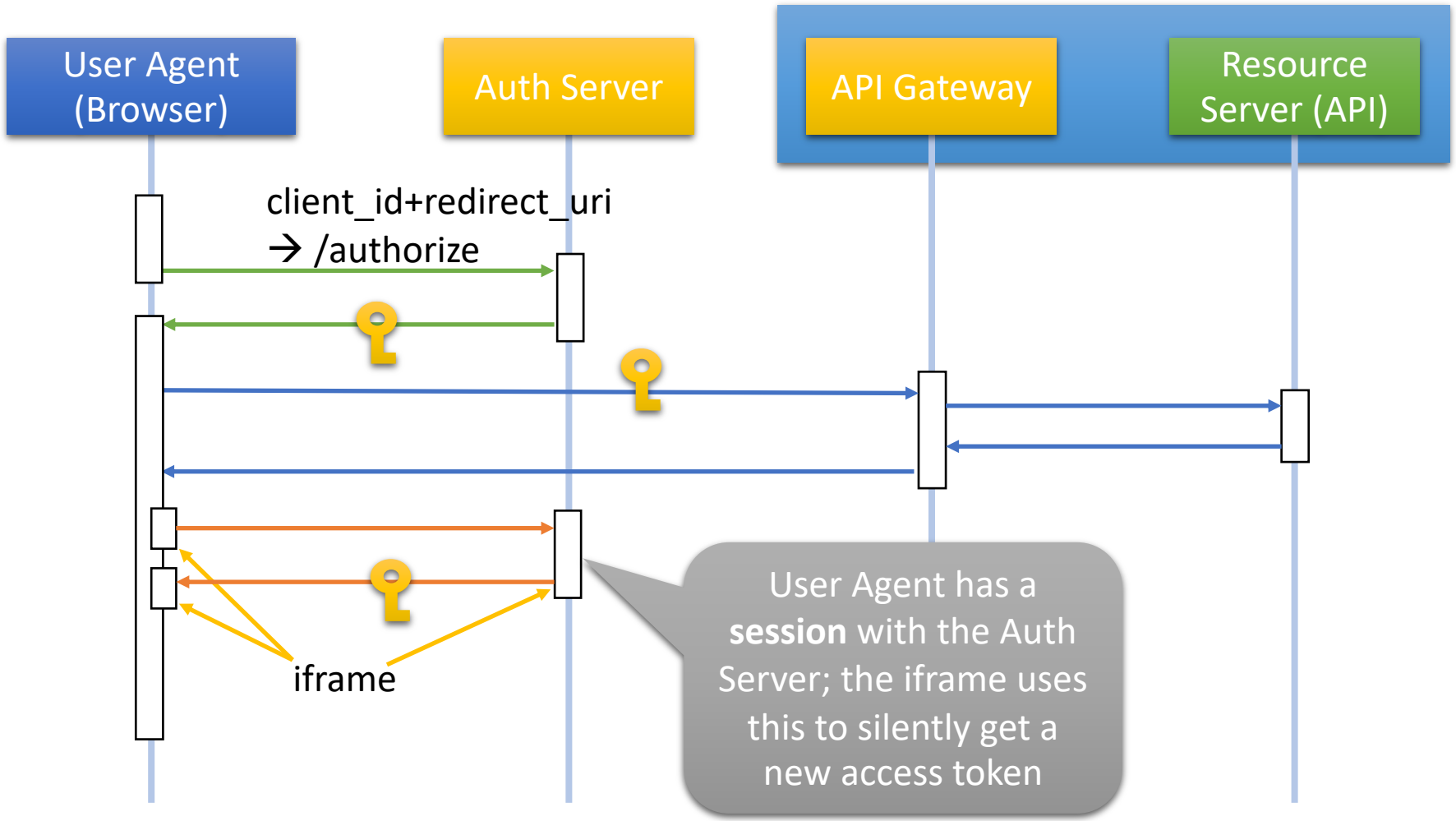
- Not as secure as Authorization Code Grant
- Access Token must be even more short lived
  - Because it can be read out from the browser, e.g.
- No Refresh Tokens! (but there’s a footnote here)

Lab 4

Refreshing Tokens

… when it’s not supposed to be possible

Silent Refresh



Silent Refresh - Gotchas

- Session duration with Auth Server must be longer than longevity of the Access Token
- Somewhat tricky to get right
- No chance to recover after Auth Server session is gone

Time for questions

Resource Owner Password Grant

As stated in the spec: Please try not to use this

Lab 5

Trusted Clients/Applications

- Trusted Clients do not need authorization from Resource Owner
- They get access with any requested scope
- Use cases:
  - The “actual” application using an API/Resources
  - Legacy migrations
  - Command line tooling
  - Backend integrations between trusted systems
- NOT Native/Mobile Apps (anymore)

Time for questions

Native/Mobile Apps

Flows to consider

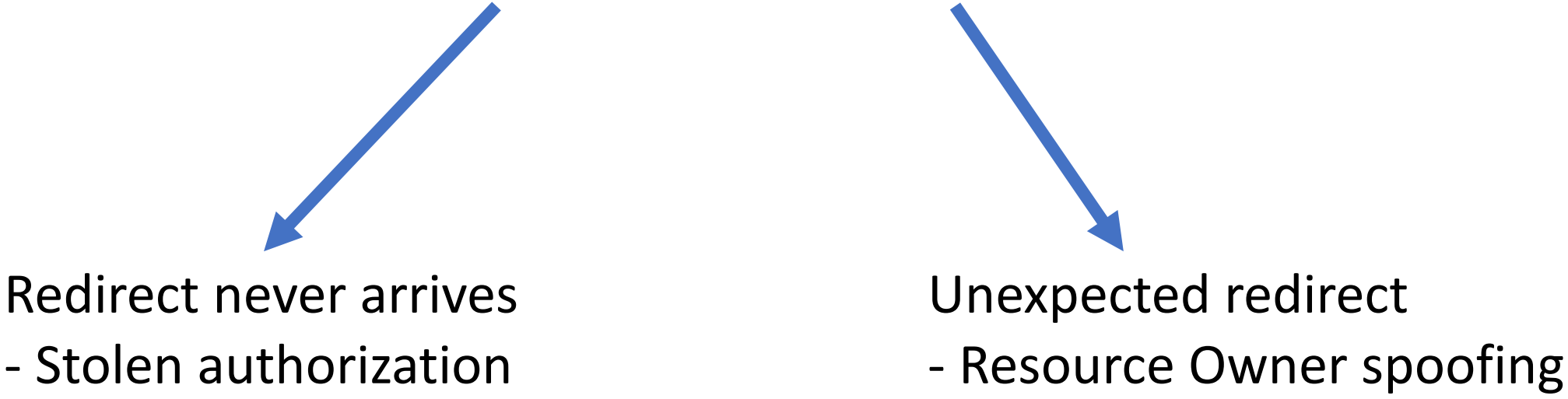
- Resource Owner Password Grant (but please don’t)
- Implicit Grant (but remember, refreshing is difficult)
- Authorization Code Grant

Didn’t you just say “Don’t use the Authorization Code Grant for Public Clients?”

Redirect in Native Clients…?

- Usually by registering custom schemes, e.g. mycoolapp://host/callback
- Schemes are not necessarily protected!
- Multiple apps can register same scheme
- Very good guidance in **RFC 8252: OAuth 2.0 for Native Apps**

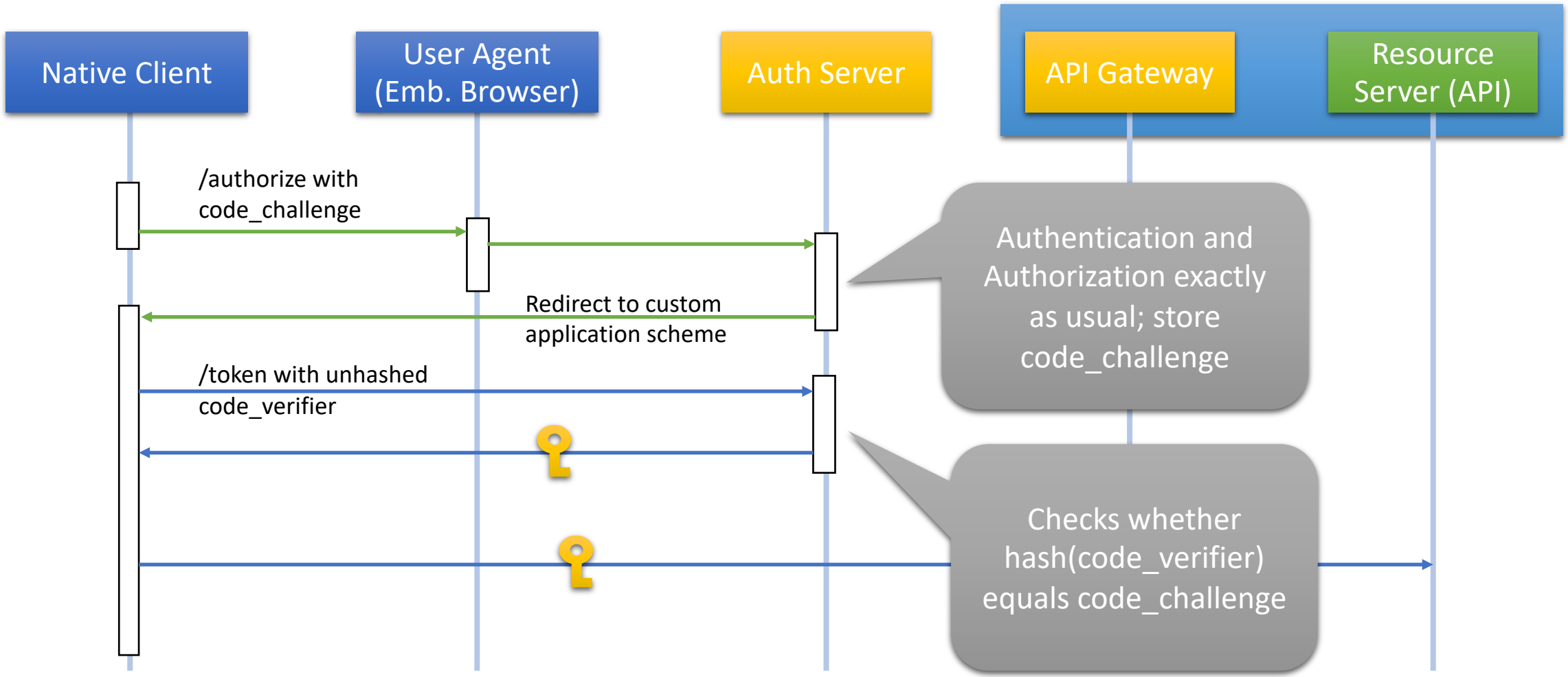
Redirect URI Confusion



PKCE Extension of Authorization Code Grant

- RFC 7636
- „Proof Key for Code Exchange by OAuth Public Clients“ is a MUST according to RFC 8252
- Mitigates both threats

Authorization Code with PKCE



Lab 6

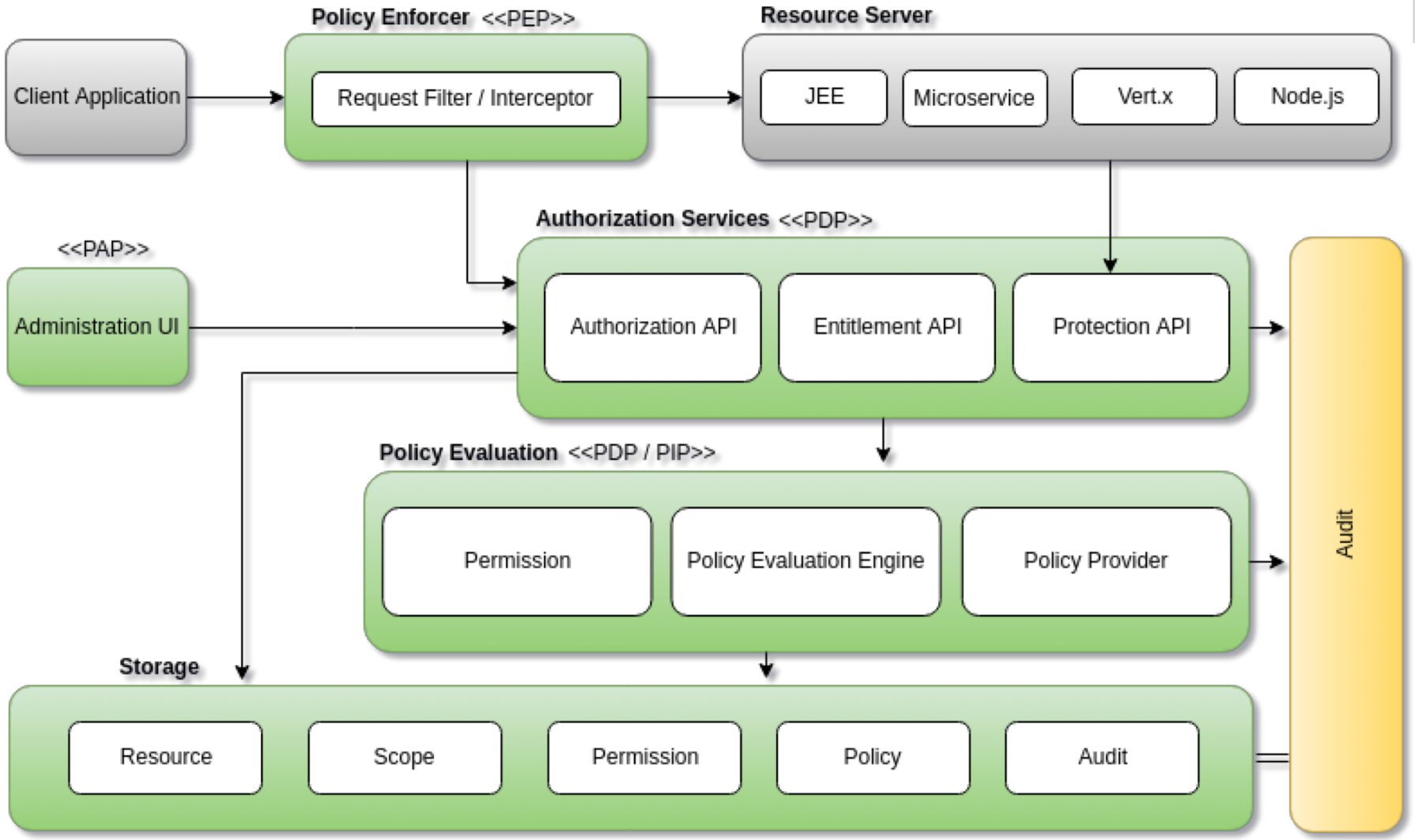
Time for questions

Implementation Considerations

Implementation Options

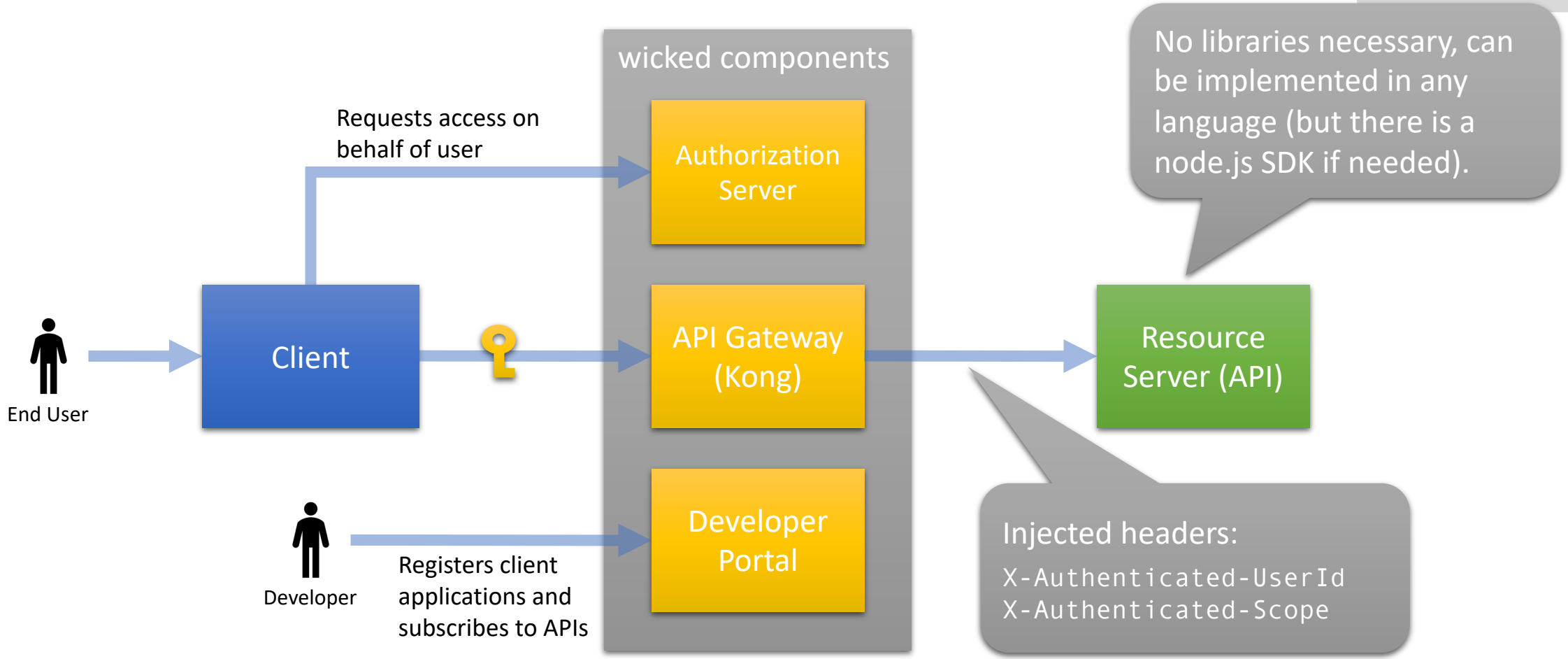
- Combined Authorization and Resource Server (naïve solution)
- Standard Software such as KeyCloak
  - Provides an Authorization Server
  - Provides Client Libraries
  - Provides libraries for the Resource Server (API) implementation
- API Management Systems
  - Some provide Authorization Servers
  - Resource Server split into API Gateway and API implementation
  - Authentication/Authorization decoupled

KeyCloak



https://www.keycloak.org/docs/3.1/authorization_services/topics/overview/architecture.html

wicked.haufe.io



Some Haufe Special Cases

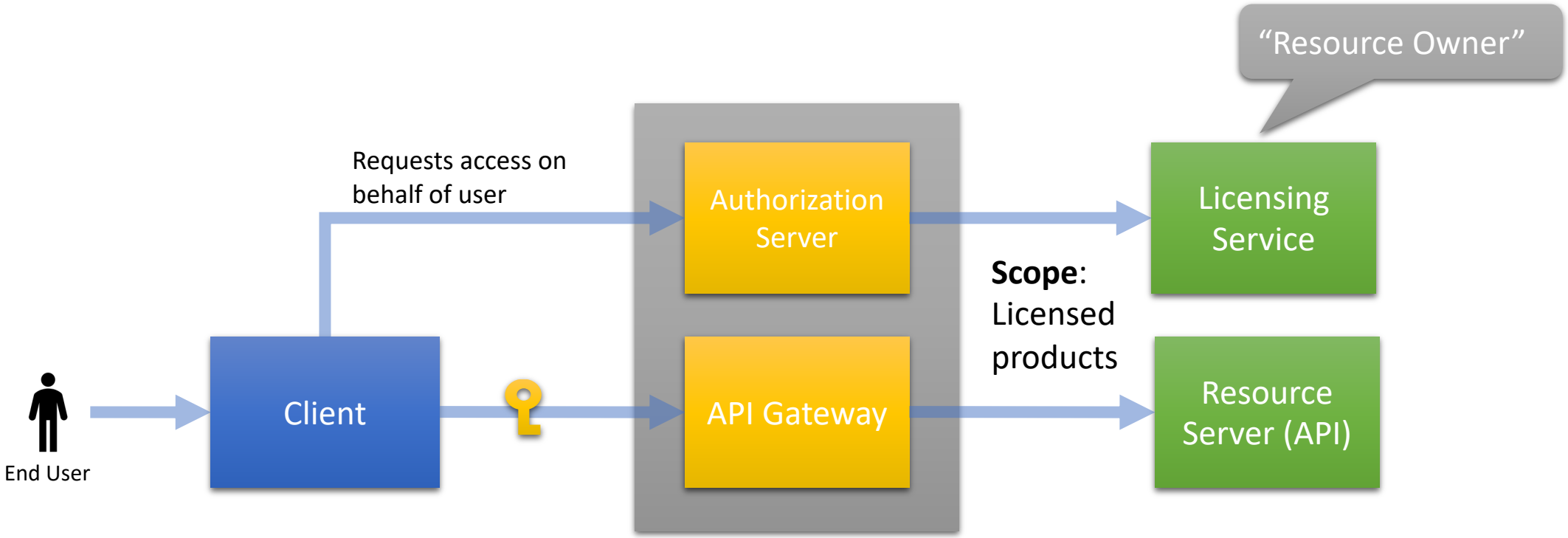
So you know who the Resource Owner is, huh?

Typical Haufe Content APIs…

- We sell licenses to content APIs
- Often (almost) no end user data
- Access to API, who decides?
  - According to OAuth2, the **Resource Owner**!
  - Wait, what?

- Here: Haufe is Resource Owner, not the End User!

OAuth2 Flow (Haufe Content)



Remarks on Haufe Content Case

- This does **not** break the OAuth2 Standard!
- Splits role of Resource Owner between
  - **End User**, as Owner of the License
  - **Licensing Service**, which verifies the license and decides on the scope

> (B) The authorization server authenticates the resource owner (via the user-agent) and establishes whether the resource owner grants or denies the client's access request.

- It doesn't say **how**…

Time for questions

Recap

What have we (hopefully) learned today?

The Four Horsemen

Client Credentials	Authorization Code Grant	Implicit Grant	Resource Owner Password Grant
• Machine-to-machine communication • Between trusted systems • No concept of "user", only "client" • Backend services	• Most common and secure flow • "Facebook flow" • For Web Applications • For Native Apps (with PKCE extension) • User friendly • Supports Refresh Tokens	• Simplified authorization flow • For Single Page Applications (SPAs) • Refresh relies on session with Authorization Server	• For certain integrations • Command Line Tools • Not recommended for most use cases

Further Topics

- OAuth2 Threat Models
- Open ID Connect
  - Extension of OAuth2, canonicalizing identity
- Device Code Flow (the fifth horseman)
- Custom Flows (explicitly allowed)
- Implementation options
  - wicked.haufe.io
  - RedHat KeyCloak
  - Various other API Gateways
  - ADFS/Azure Apps
- Token types (opaque, JWT,…)

Reading List

- RFC 6749: **The OAuth 2.0 Authorization Framework**
  https://tools.ietf.org/html/rfc6749
- RFC 6819: **OAuth 2.0 Threat Model and Security Considerations**
  https://tools.ietf.org/html/rfc6819
- RFC 8252: **OAuth 2.0 for Native Apps**
  https://tools.ietf.org/html/rfc8252
- RFC 7636: **Proof Key for Code Exchange by OAuth Public Clients**
  https://tools.ietf.org/html/rfc7636
- RFC 7591: **OAuth 2.0 Dynamic Client Registration Protocol**
  https://tools.ietf.org/html/rfc7591
- Link collection: https://oauth.net/2/
- **OpenID Connect**: https://openid.net/connect