

authN/Z for REST services

# About me

- Software architect
- Consultancy
- Fixed price projects
- SecAppDev founder
- <https://www.johanpeeters.com>



[@YoPeeters](https://twitter.com/YoPeeters)



[yo@johanpeeters.com](mailto:yo@johanpeeters.com)



# Agenda

## Architectural principles

Externalise the Identity Provider

Use self-contained security tokens

## Enabling technology

OpenID Connect

JWT

## Wishlist

Use Proof-of-Possession tokens

Externalise the policy decision point

## Enabling technology

JWT

UMA?

# REST API consumers

Click to add text

# REST API consumers

- Web application

Click to add text

# REST API consumers

- Web application
- SPAs

Click to add text

# REST API consumers

- Web application
- SPAs
- Mobile apps

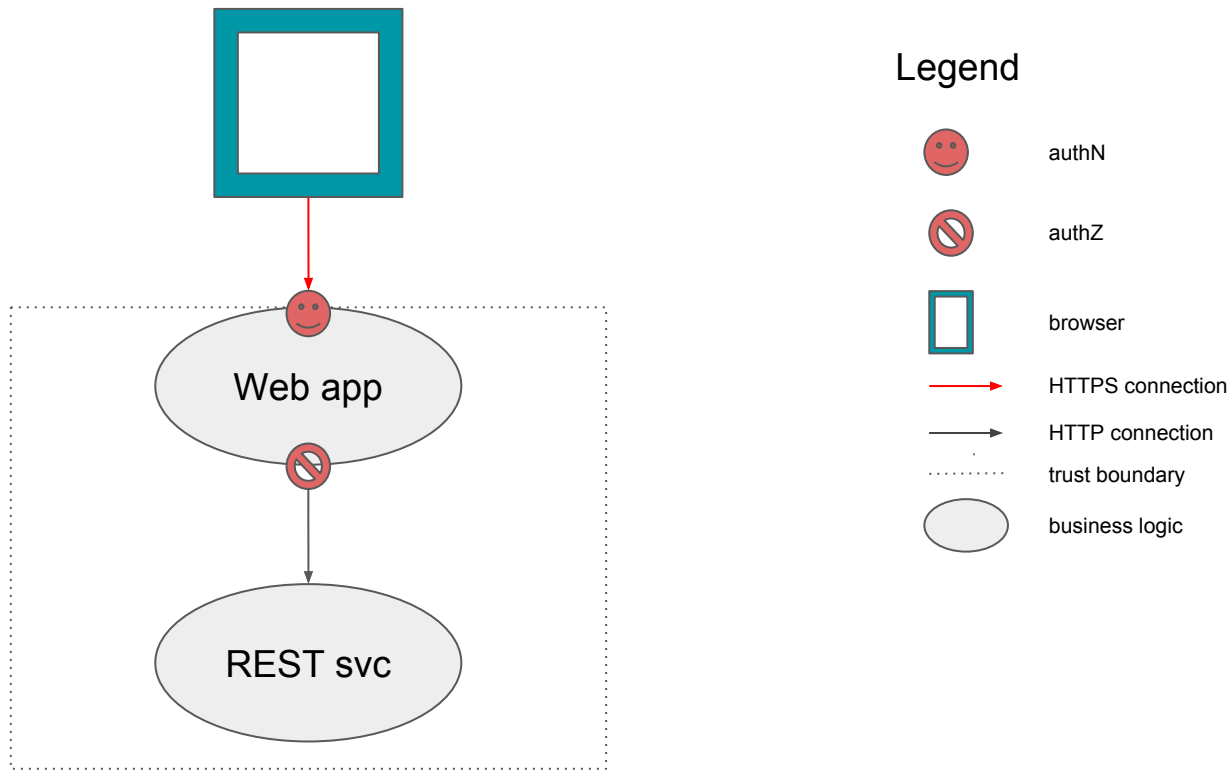
Click to add text

# REST API consumers

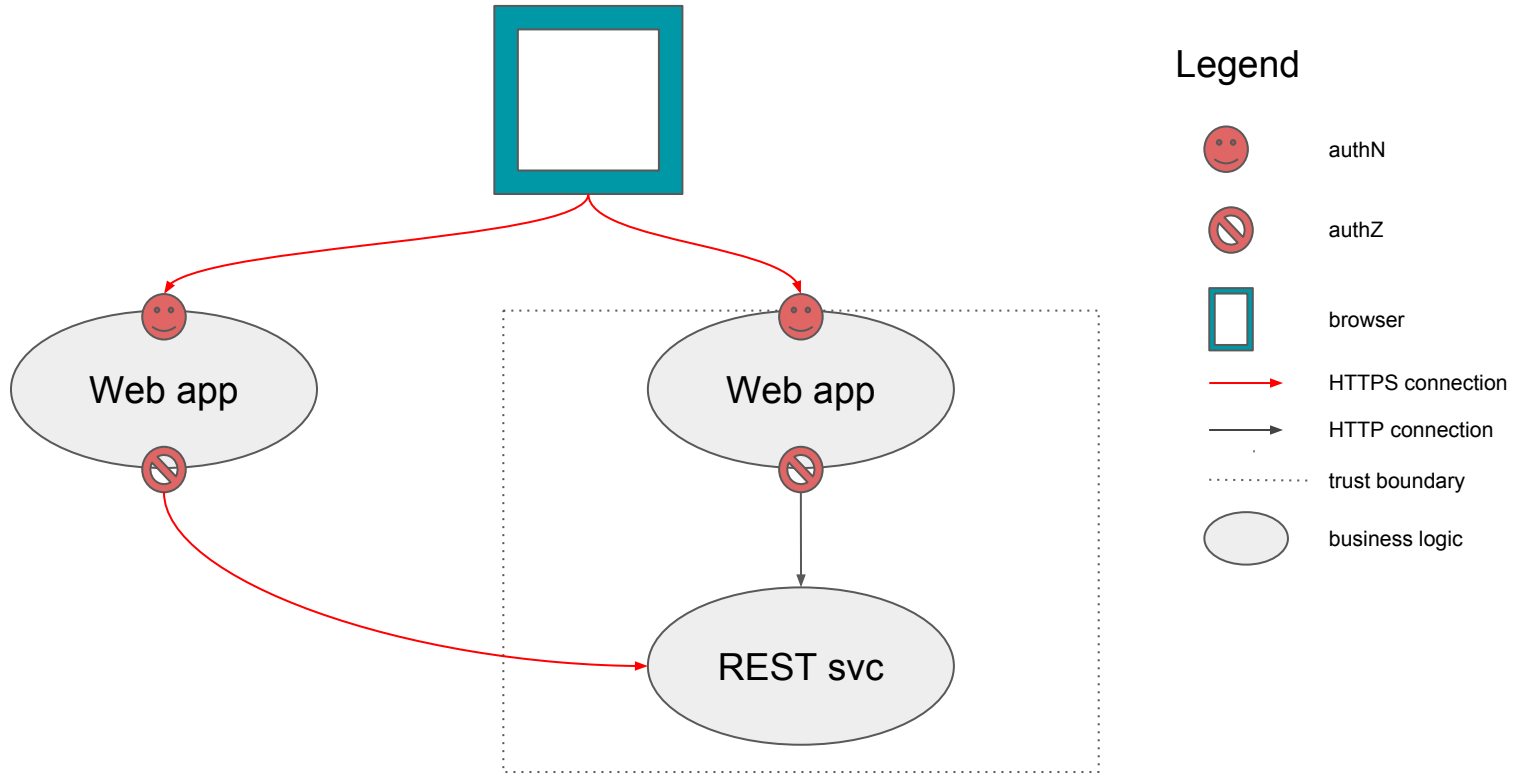
- Web application
- SPAs
- Mobile apps
- Other REST services

Click to add text

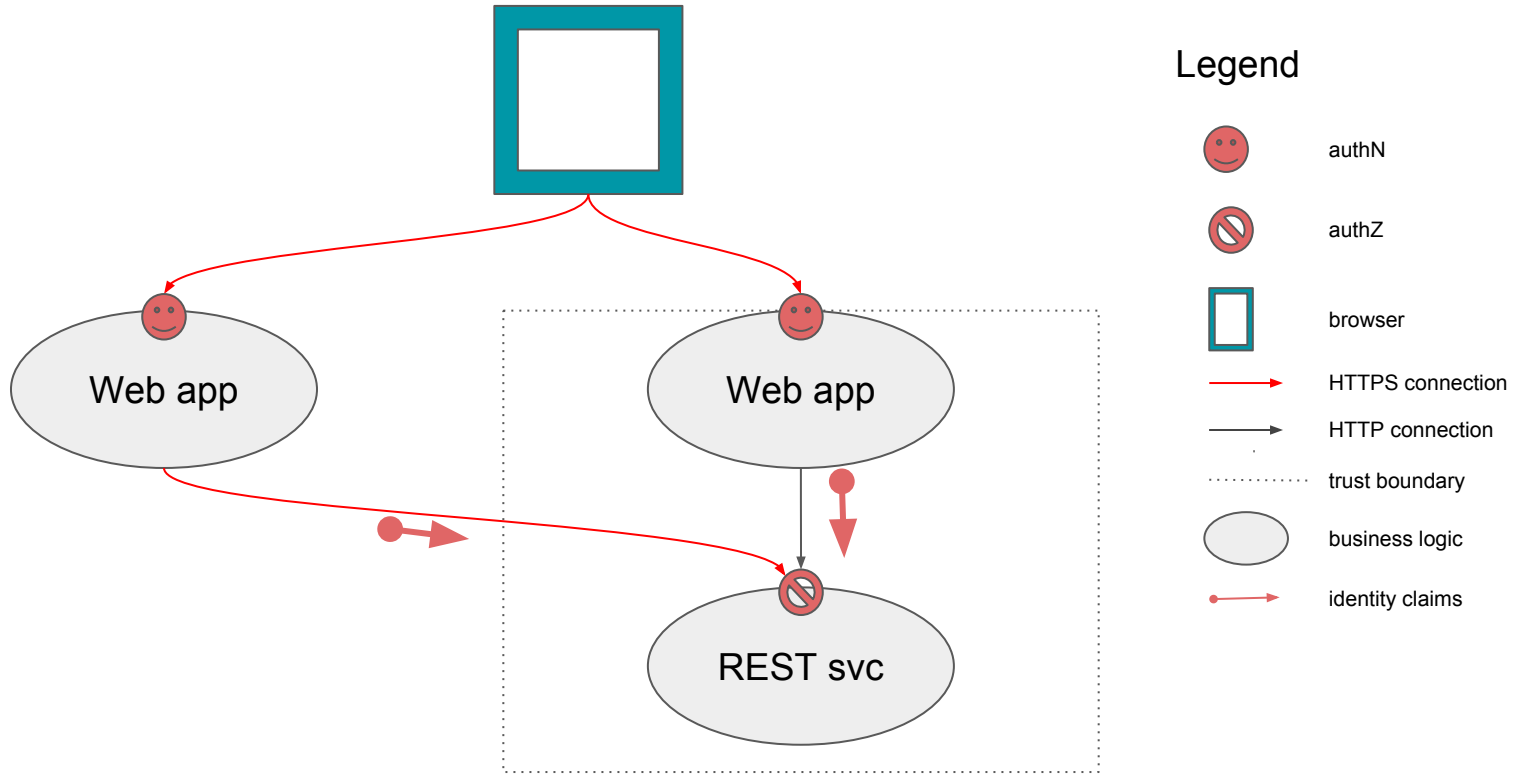




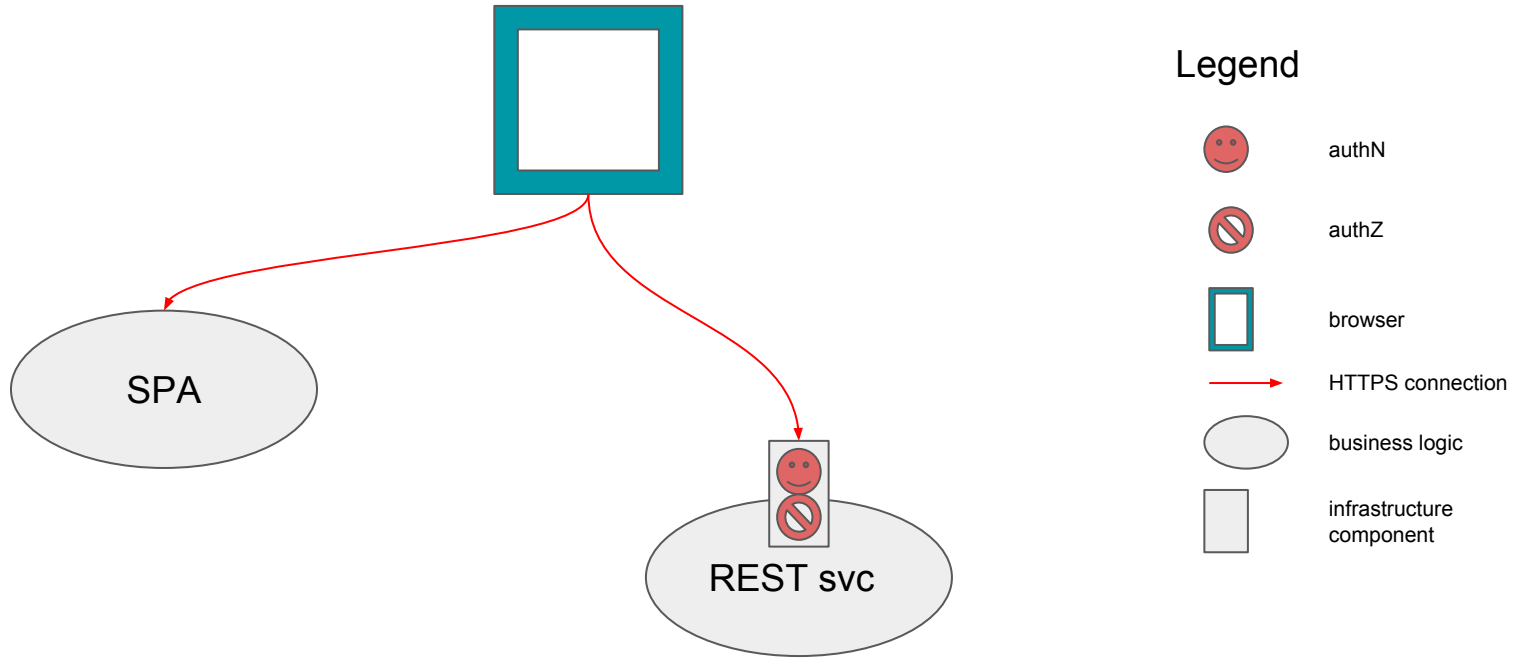
Traditional web applications



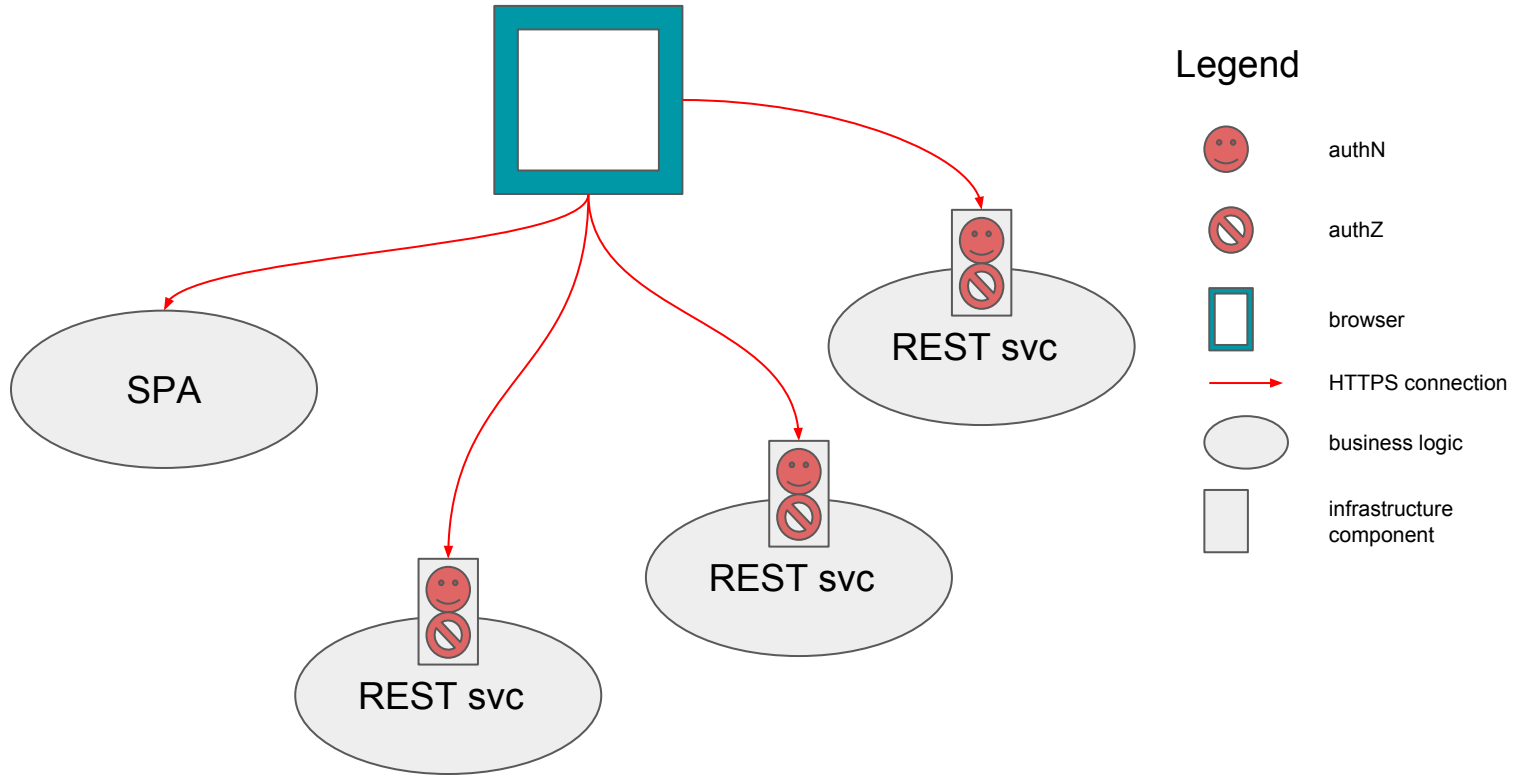
This could work...



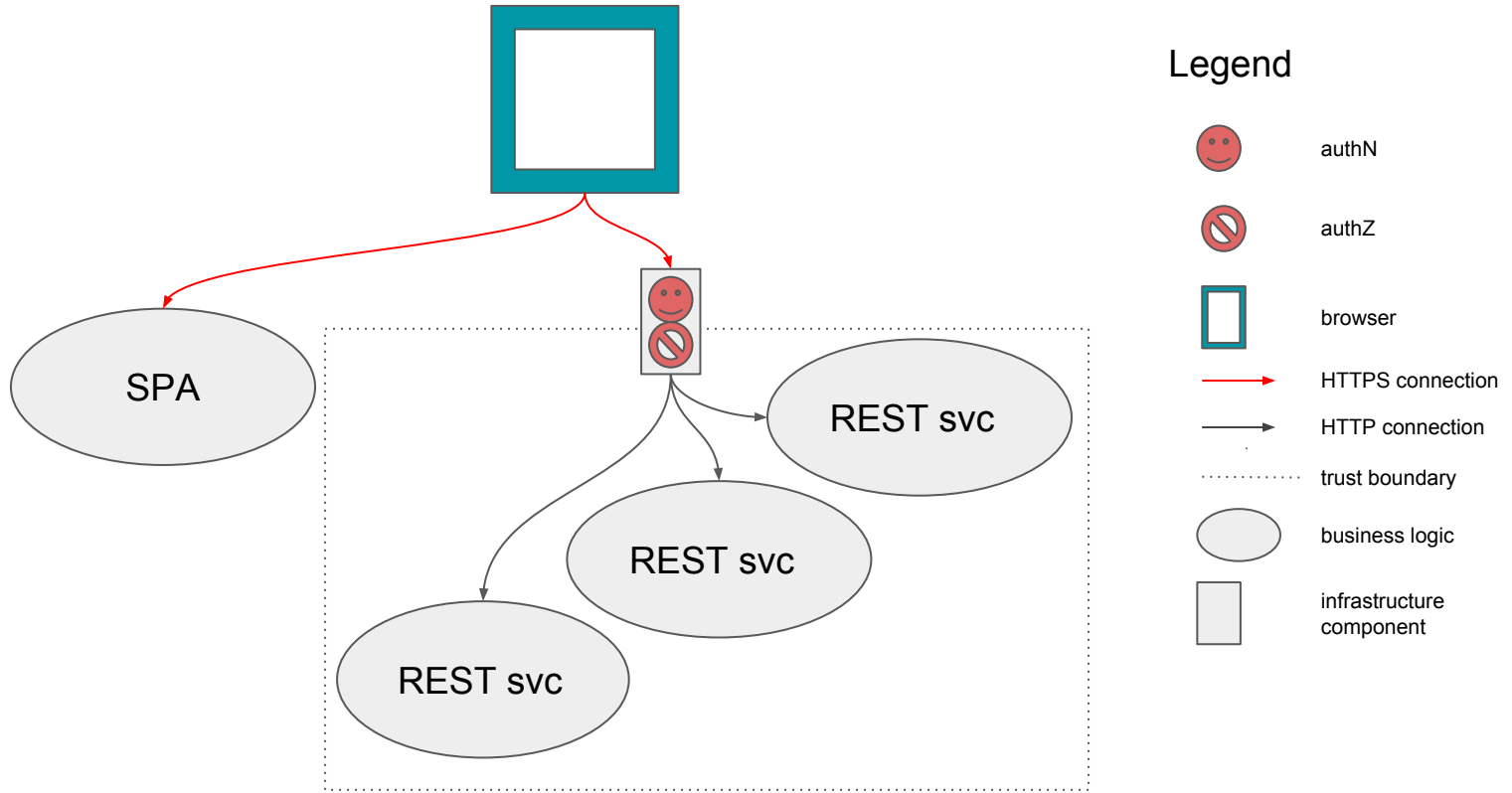
... but this is probably better



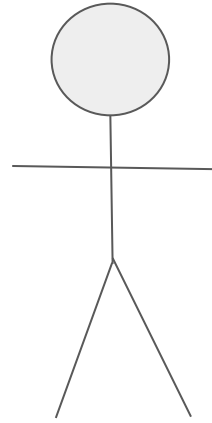
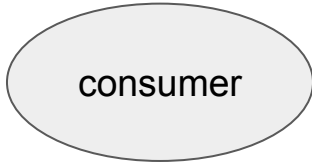
Single Page Applications (SPA) - also applies to mobile apps



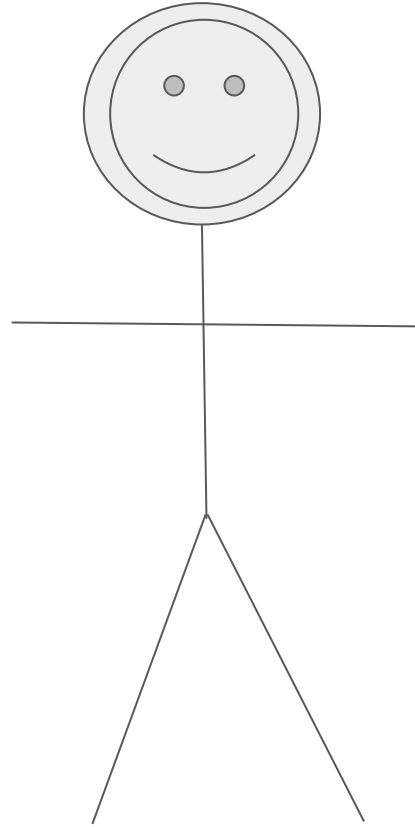
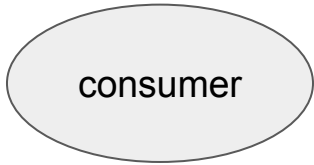
How does this pan out when calling multiple services?



How about an API Gateway?

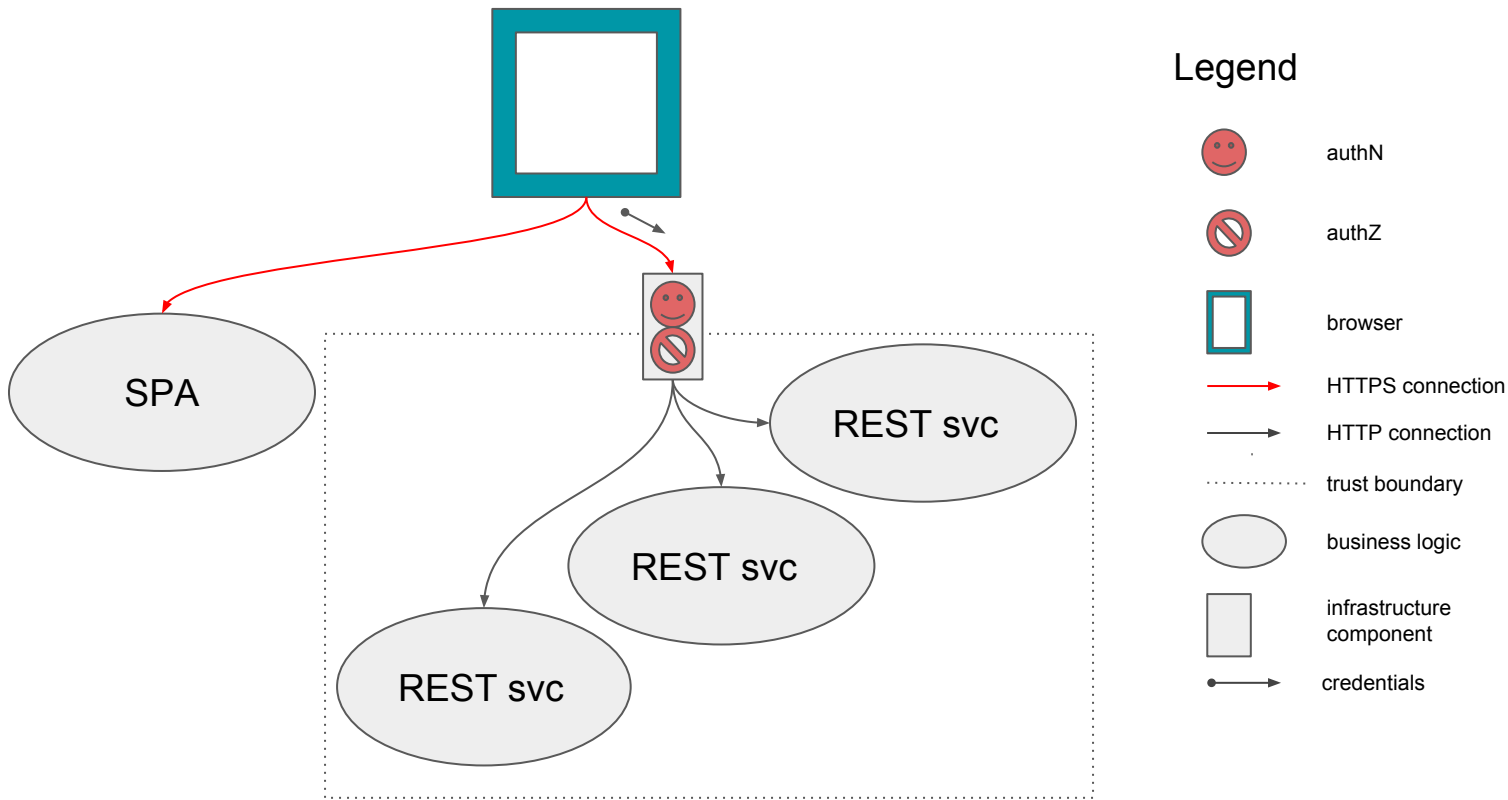


Who or what are we authenticating?












Let's start with who





### Legend

-  authN
-  authZ
-  browser
-  HTTPS connection
-  HTTP connection
-  trust boundary
-  business logic
-  infrastructure component
-  credentials

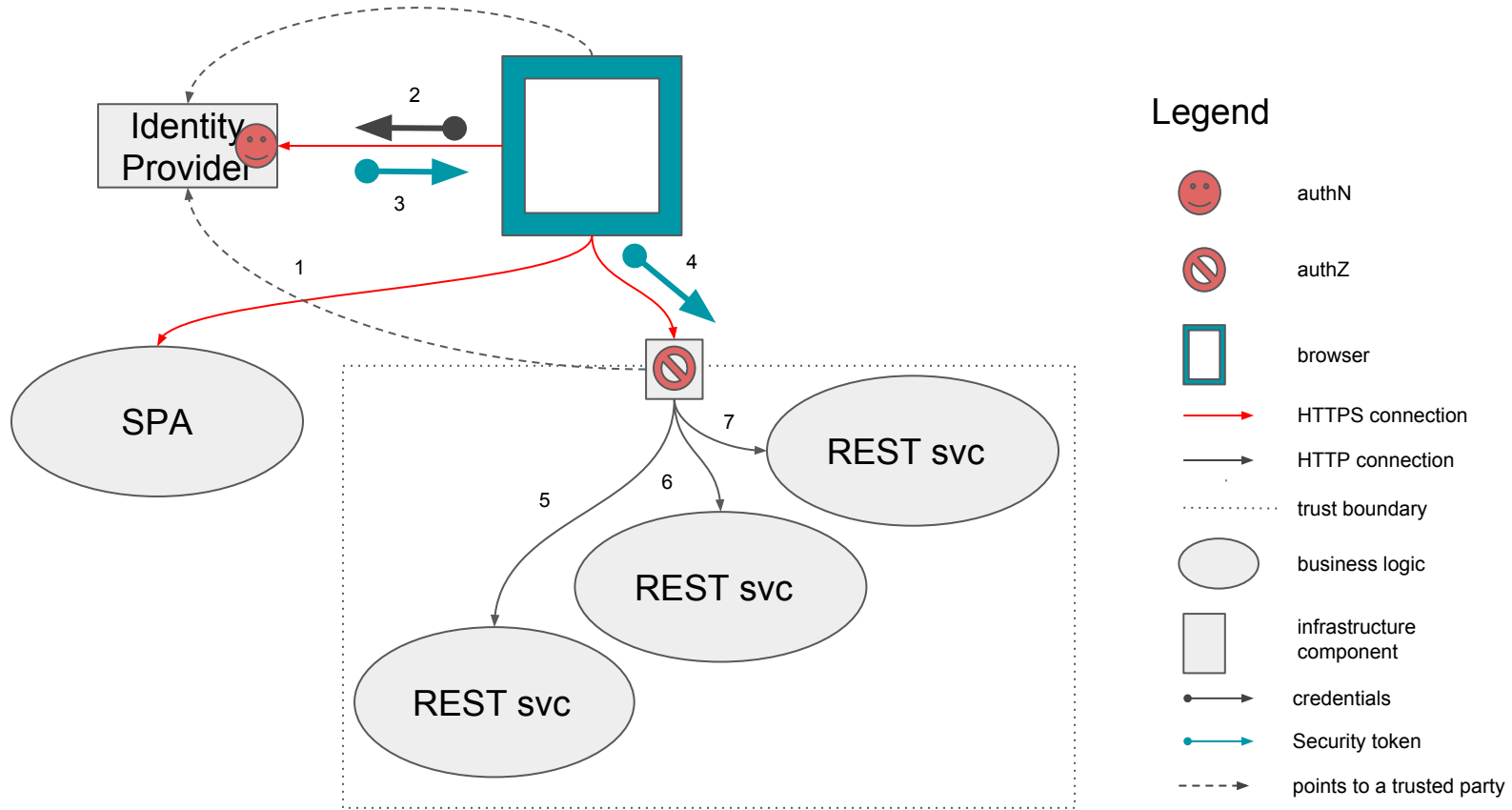
Sending credentials

# Pros and cons?












Simple

The client impersonates the user

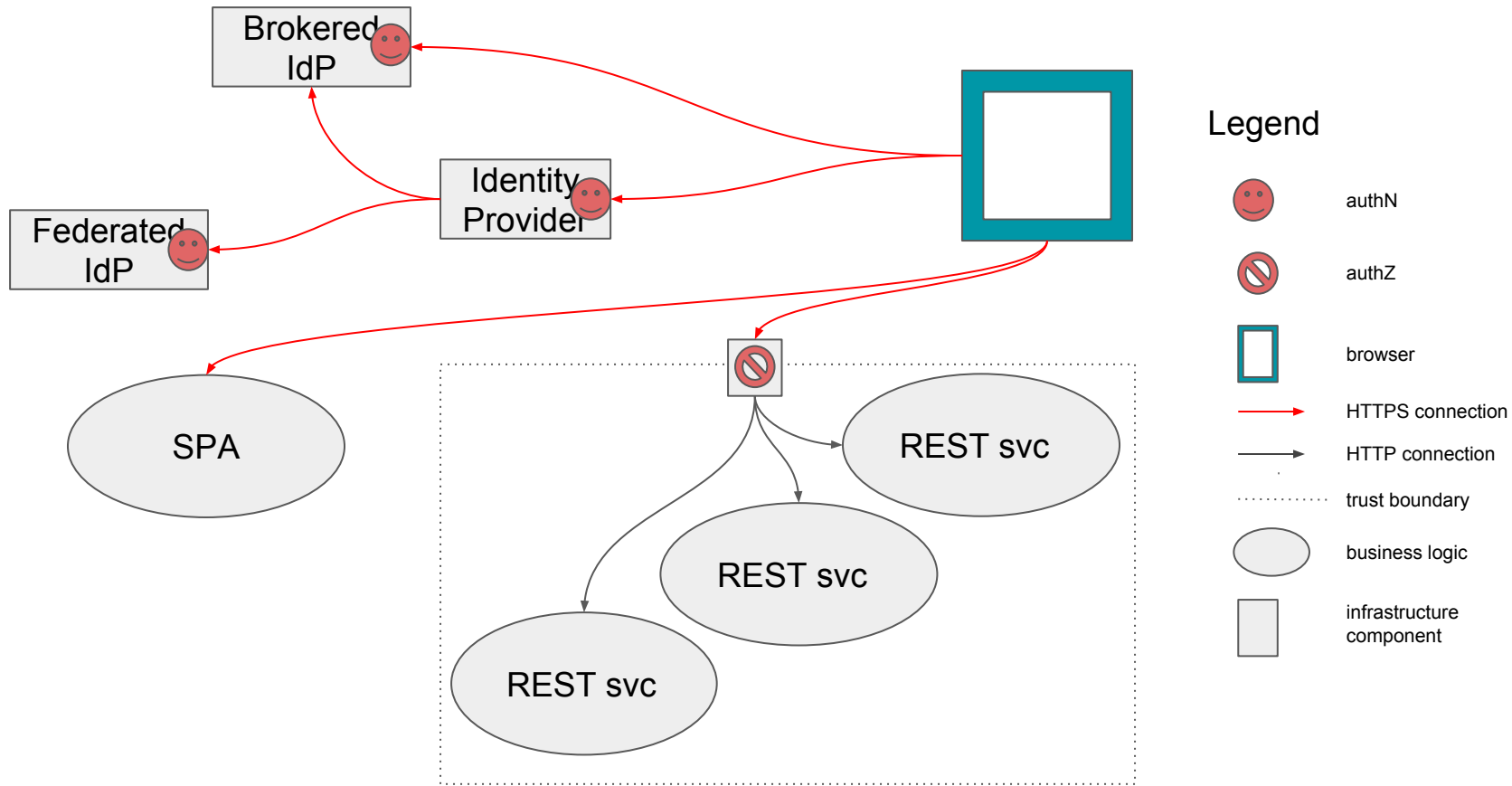
Identity management and API GW are tightly coupled



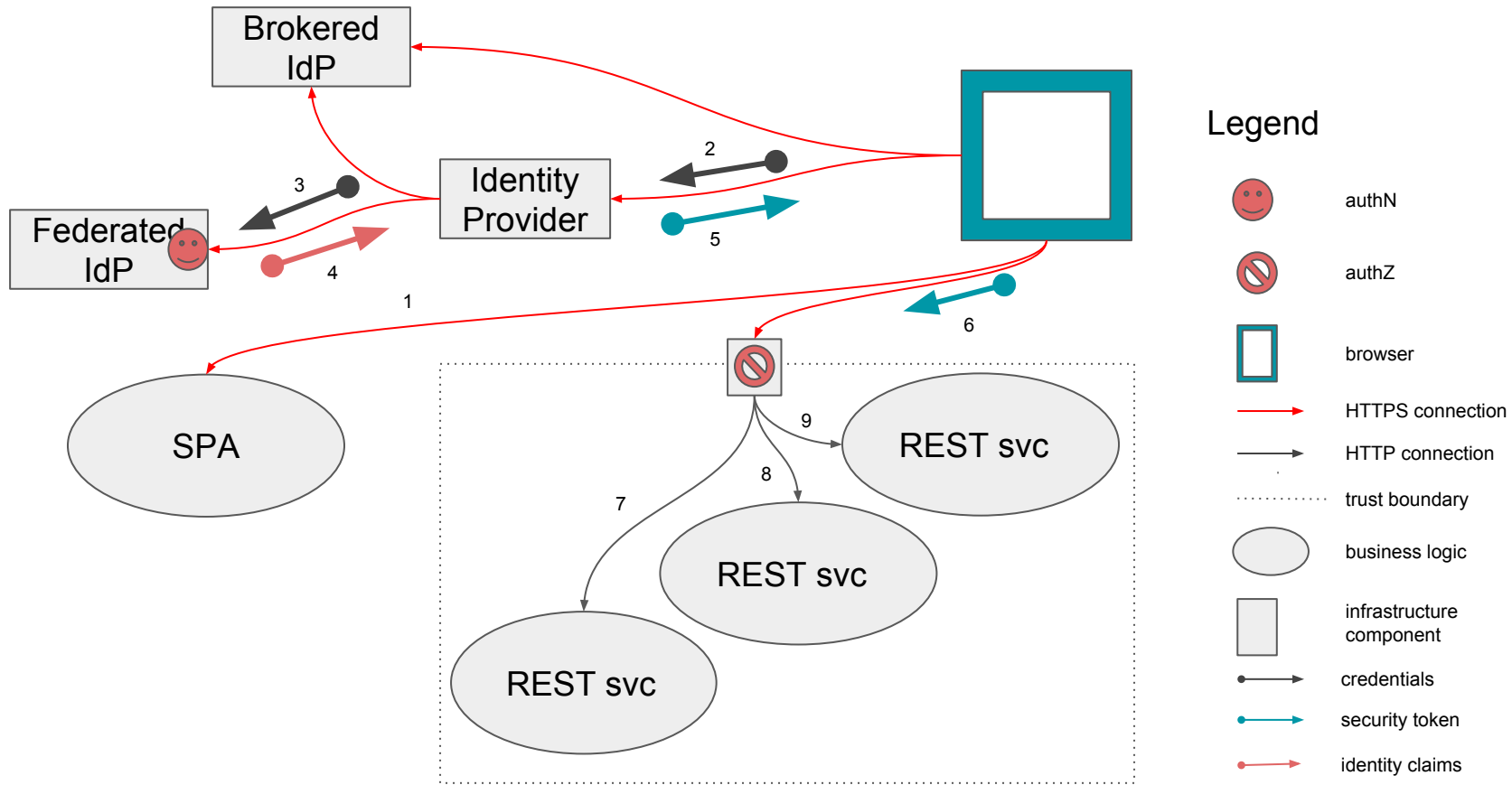
## Legend

-  authN
-  authZ
-  browser
-  HTTPS connection
-  HTTP connection
-  trust boundary
-  business logic
-  infrastructure component
-  credentials
-  Security token
-  points to a trusted party

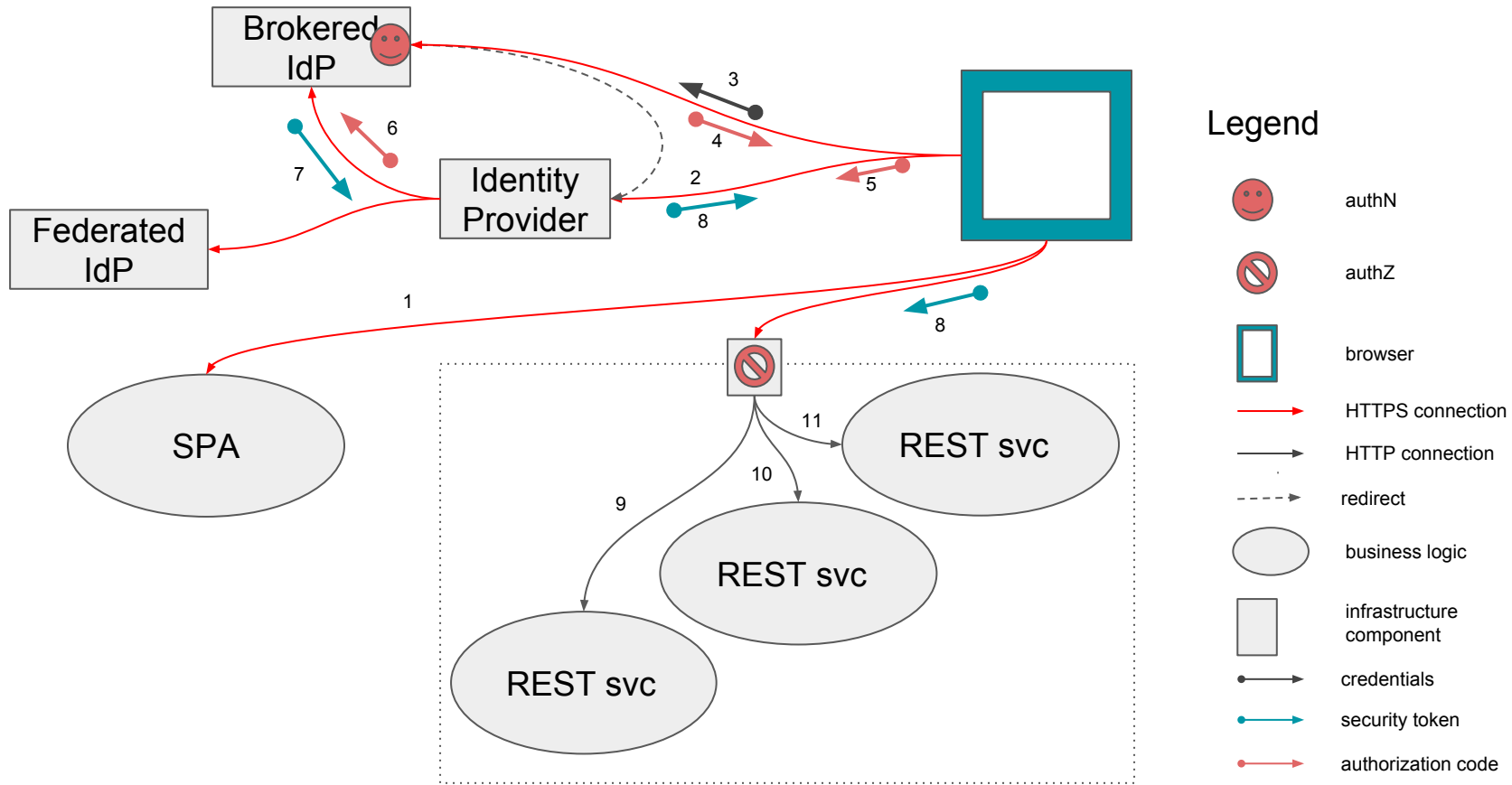
Externalizing the IdP



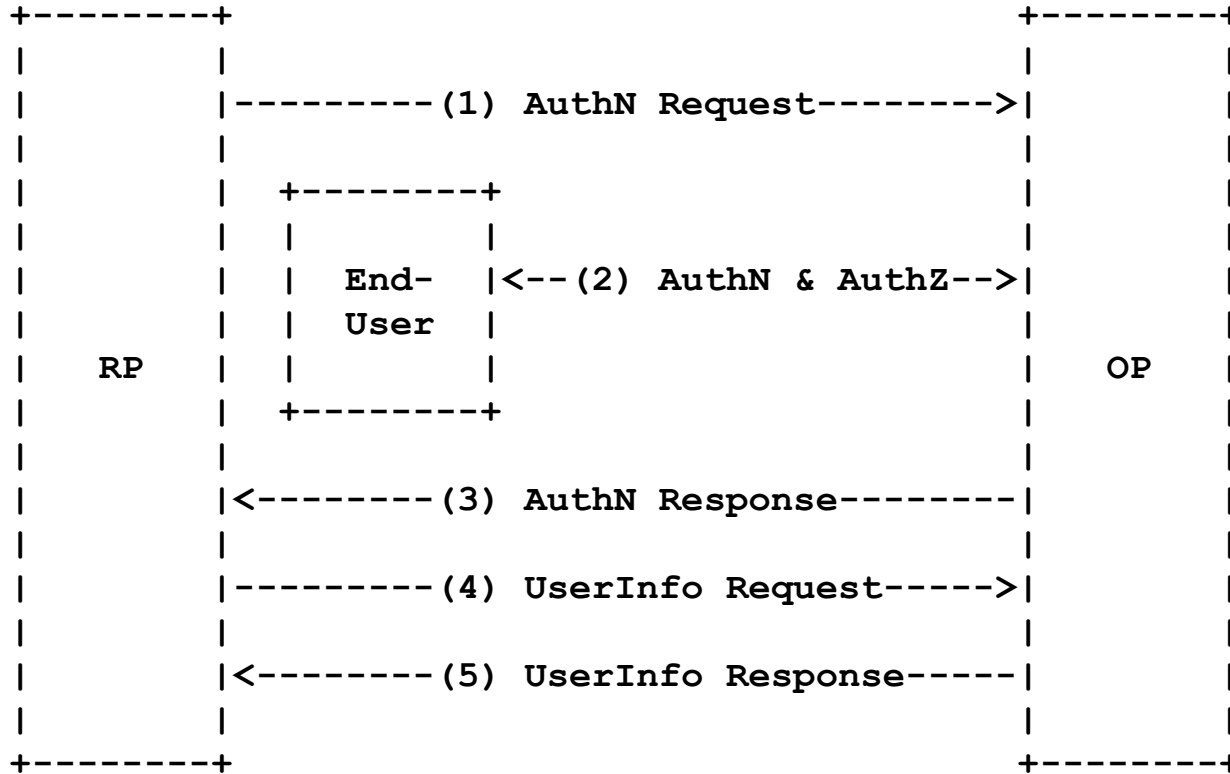
Further devolving responsibility for identity management



## Federation



IdP brokering



## [Generic OpenID Connect flow](#)

# OpenID Connect

- authN protocol implemented on top of OAuth 2.0, an authZ protocol
  - identity claims as resources
  - resource owner consents to accessing specific claims
- SSO across APIs and over time
- IdP, or OP, exposes a web app
- 3 flows:
  - Authorization code flow for web applications
  - Implicit flow for SPAs and mobile apps
  - Hybrid flow for ...



# Authorization code flow

The Authorization Code flow is suitable for Clients that can securely maintain a Client Secret between themselves and the Authorization Server.

...

The Authorization Code Flow goes through the following steps.

1. Client prepares an Authentication Request containing the desired request parameters.
2. Client sends the request to the Authorization Server.
3. Authorization Server Authenticates the End-User.
4. Authorization Server obtains End-User Consent/Authorization.
5. Authorization Server sends the End-User back to the Client with an Authorization Code.
6. Client requests a response using the Authorization Code at the Token Endpoint.
7. Client receives a response that contains an **ID Token** and **Access Token** in the response body.
8. Client validates the ID token and retrieves the End-User's Subject Identifier.

# Implicit flow

The Implicit Flow is mainly used by Clients implemented in a browser using a scripting language.

...

The Implicit Flow follows the following steps:

1. Client prepares an Authentication Request containing the desired request parameters.
2. Client sends the request to the Authorization Server.
3. Authorization Server Authenticates the End-User.
4. Authorization Server obtains End-User Consent/Authorization.
5. Authorization Server sends the End-User back to the Client with an **ID Token** and, if requested, an **Access Token**.
6. Client validates the ID token and retrieves the End-User's Subject Identifier.

# Hybrid flow

The Hybrid Flow follows the following steps:

1. Client prepares an Authentication Request containing the desired request parameters.
2. Client sends the request to the Authorization Server.
3. Authorization Server Authenticates the End-User.
4. Authorization Server obtains End-User Consent/Authorization.
5. Authorization Server sends the End-User back to the Client with an Authorization Code **and, depending on the Response Type, one or more additional parameters.**
6. Client requests a response using the Authorization Code at the Token Endpoint.
7. Client receives a response that contains an **ID Token** and **Access Token** in the response body.
8. Client validates the ID Token and retrieves the End-User's Subject Identifier.

# These tokens ...

## ID Token

- OpenID Connect
- bearer
- standardized
- JWT

## Access Token

- OAuth 2.0
- bearer
- not standardized
- 'usually opaque to the client'

# OAuth 2.0 access token

Access tokens are credentials used to access protected resources. An access token is a string representing an authorization issued to the client. **The string is usually opaque to the client.** Tokens represent specific scopes and durations of access, granted by the resource owner, and enforced by the resource server and authorization server.

The token may denote an identifier used to retrieve the authorization information or **may self-contain** the authorization information in a verifiable manner (i.e., a token string consisting of some data and a signature). Additional authentication credentials, which are beyond the scope of this specification, may be required in order for the client to use a token.

revocation



	self-contained	reference
holder-of-key	(JWT)	
bearer	JWT	

theft





### Encoded PASTE A TOKEN HERE

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoiYXV0bzR5dWV9.TjVA950rM7E2cBab30RMHrHDcEfxjoYZgeFONFh7HgQ
```

### Decoded EDIT THE PAYLOAD AND SECRET (ONLY HS256 SUPPORTED)

#### HEADER: ALGORITHM & TOKEN TYPE

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

**OOPS!**



<https://auth0.com/blog/critical-vulnerabilities-in-json-web-token-libraries/>

#### PAYLOAD: DATA

```
{  
  "sub": "1234567890",  
  "name": "John Doe",  
  "admin": true  
}
```

#### VERIFY SIGNATURE

```
HMACSHA256(  
  base64UrlEncode(header) + "." +  
  base64UrlEncode(payload),  
  secret  
)  secret base64 encoded
```

✔ Signature Verified

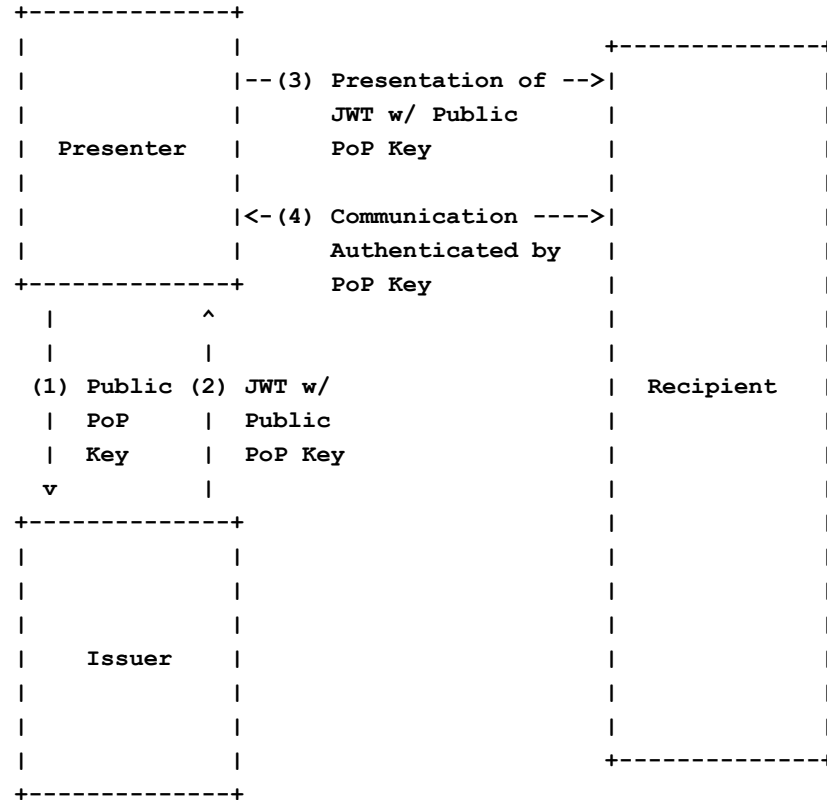
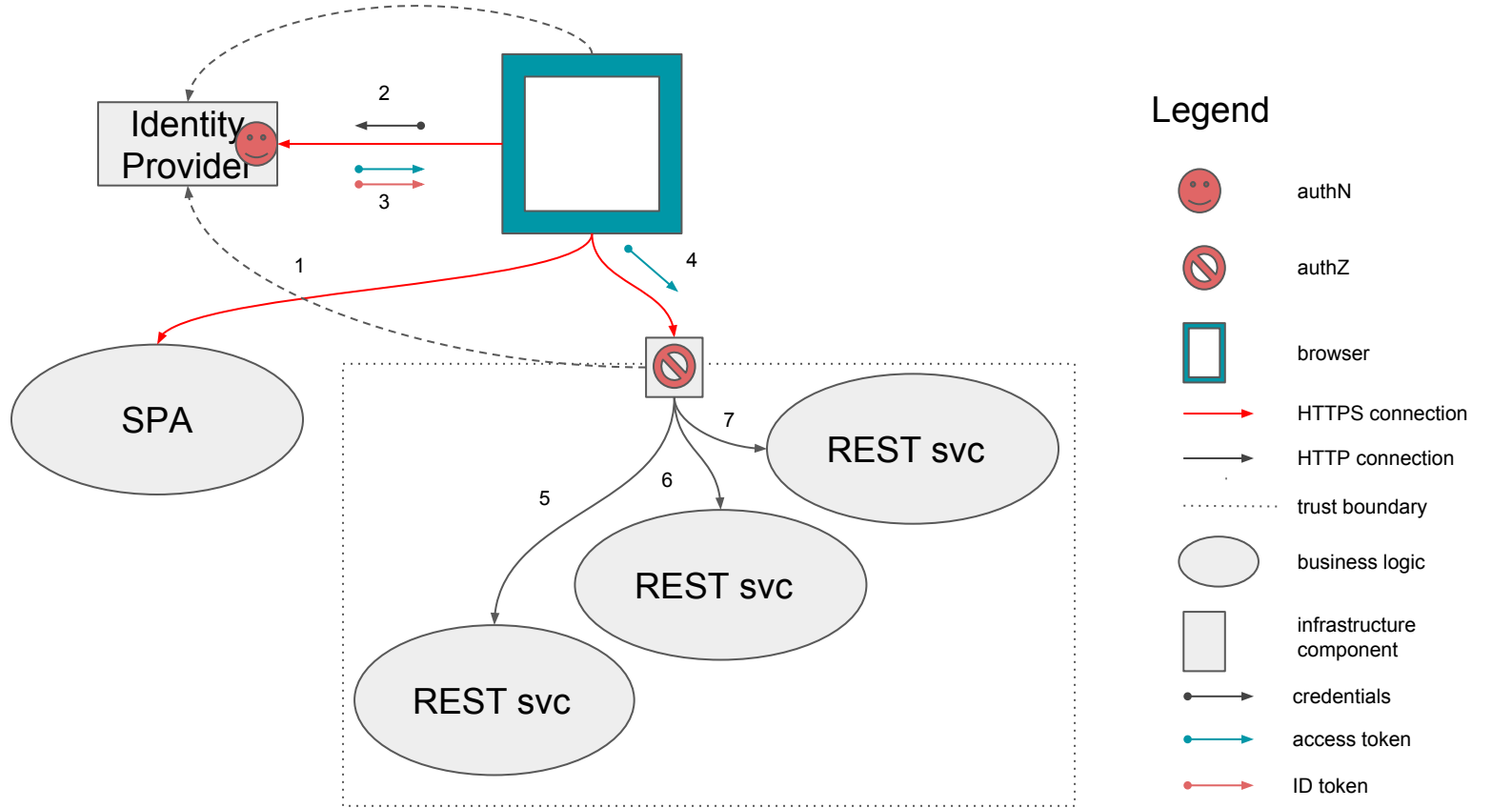


Figure 2: Proof of Possession with an Asymmetric Key

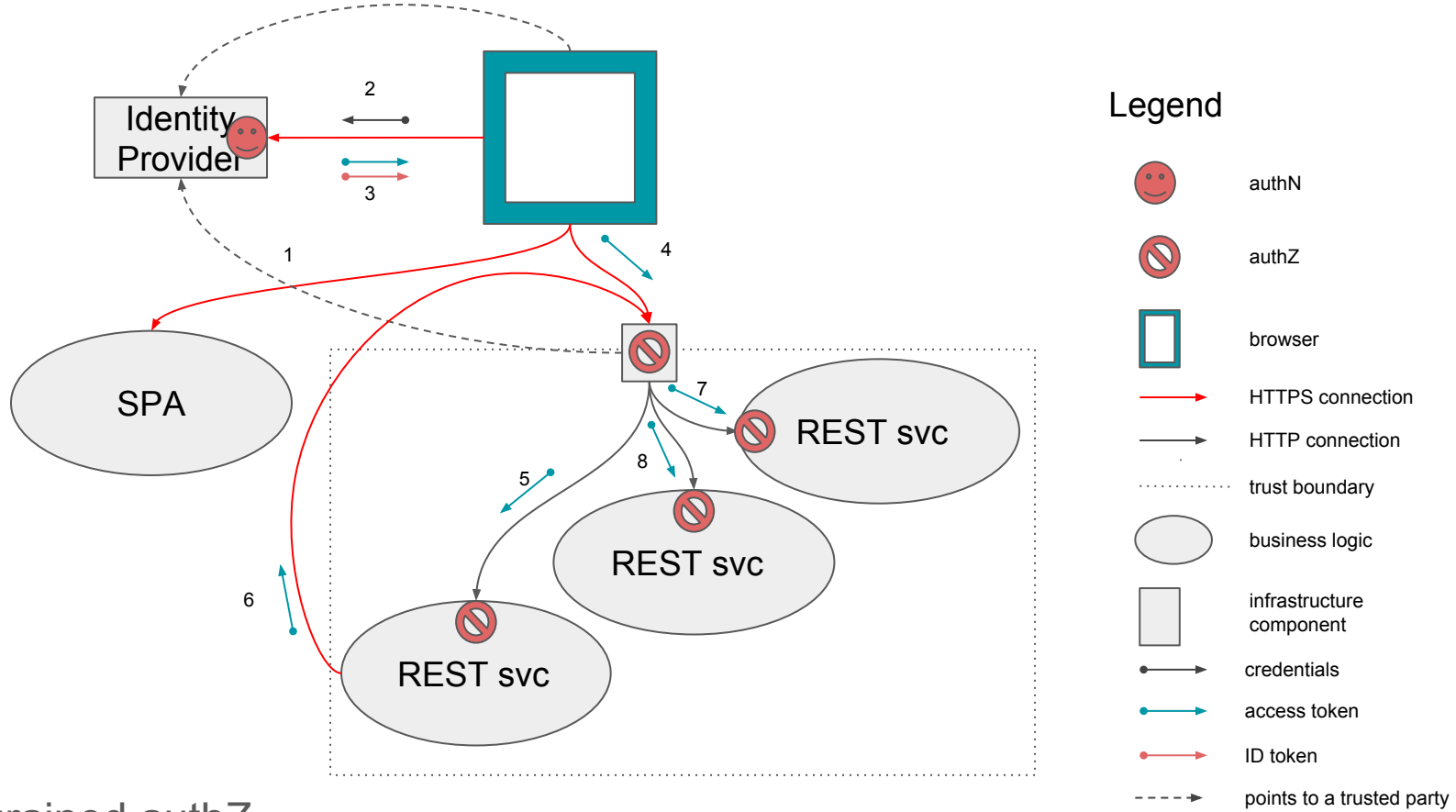


# Proof-of-Possession Key Semantics for JWTs

In the case illustrated in Figure 2, the presenter generates a public/private key pair and (1) sends the public key to the issuer, which creates a JWT that contains the public key (or an identifier for it) in the confirmation claim. The entire JWT is integrity protected using a digital signature to protect it against modifications. The JWT is then (2) sent to the presenter. When the presenter (3) presents the JWT to the recipient, it also needs to demonstrate possession of the private key. The presenter, for example, (4) uses the private key in a **Transport Layer Security (TLS) exchange** with the recipient or (4) **signs a nonce** with the private key. The recipient is able to verify that it is interacting with the genuine presenter by extracting the public key from the confirmation claim of the JWT (after verifying the digital signature of the JWT) and utilizing it with the private key in the TLS exchange or by checking the nonce signature.



Assuming access token is also a JWT...



Fine-grained authZ

Demo

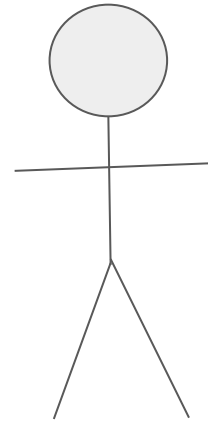
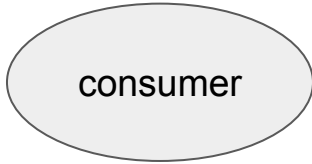
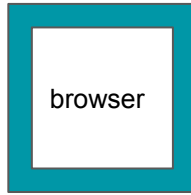
# What if an attacker...

... forges a token?

... steals a token?

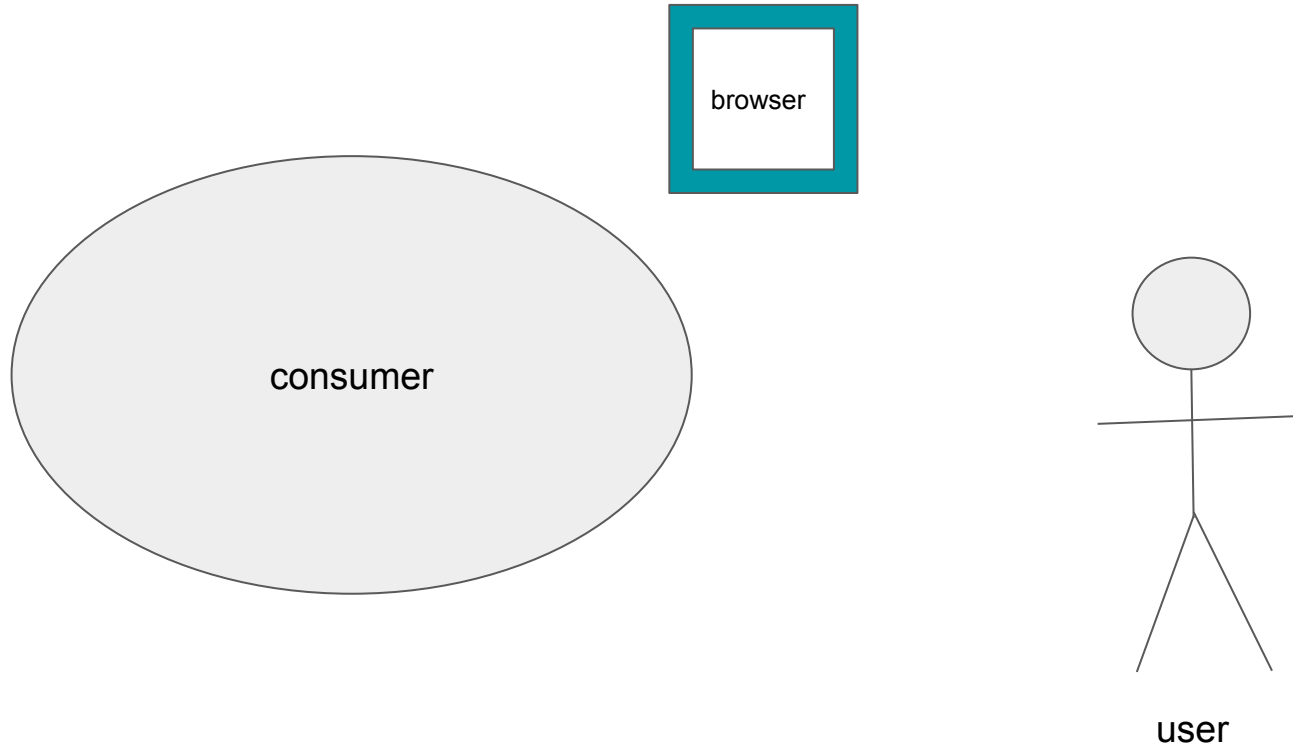
... bypasses the API Gateway?

... publishes a rogue client?



user

Who or what are we authenticating?



Registering and authenticating the client

# Authenticating the client

Register the client

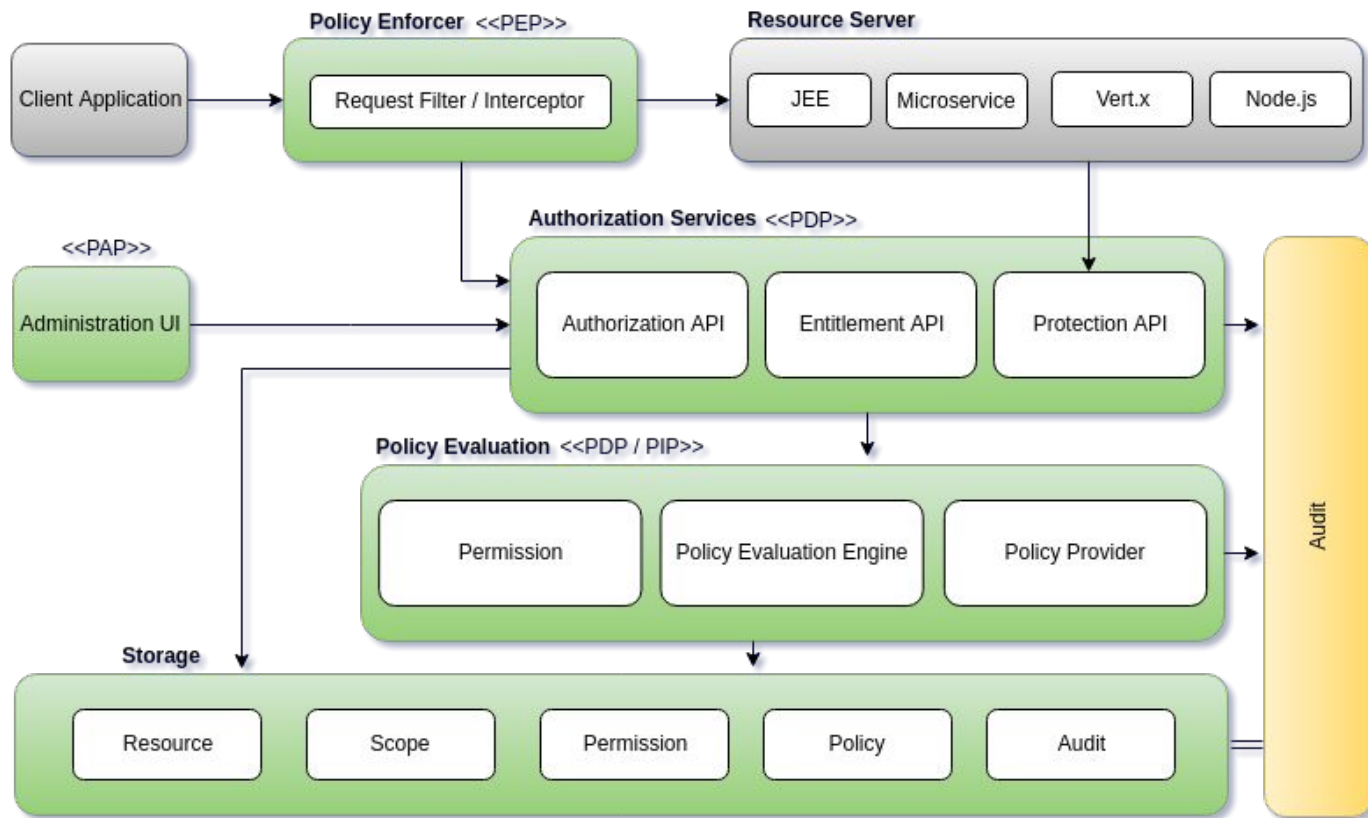
API keys are often

- Stored in the client
- Shared amongst potentially many clients
- Sent in clear text
- Hence not very secure

Signing is better than sending the secret across the wire

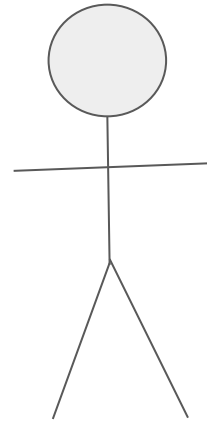
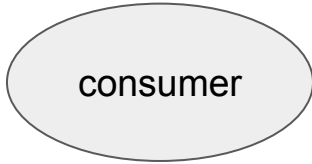
Looking forward to using Proof-of-Possession tokens





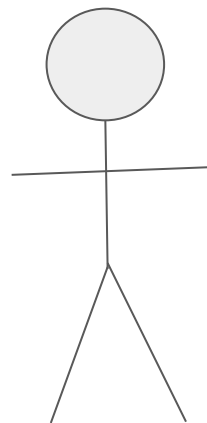
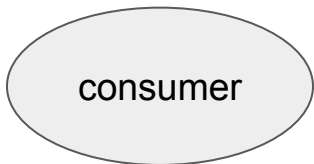
An authZ architecture





user

Who or what are we authenticating?



user

Maybe next time...

# Wrap-up discussion

## Architectural principles

## Enabling technology

Externalise the Identity Provider

OpenID Connect

Use self-contained security tokens

JWT

## Wishlist

## Enabling technology

Use Proof-of-Possession tokens

JWT

Externalise the policy decision point

UMA?