Technische Hochschule Rosenheim
Technical University of Applied Sciences

Microservices

Security

Content

Basics

Authentication

The application confirms that a party is who she is e.g. by typing in a username and a password in classical applications but there are also other options like smartcards, 2FA and many more.

Authorization

Is an already authenticated user allowed this or not.

Role-based access control (RBAC)

Role-based access control determines what a user is allowed to do based on his assigned roles e.g. AUTOMOTIVE_TEAM_LEADER

Access control list (ACL)

An ACL captures a list of principals (maybe a single person or a group) with their access level e.g.

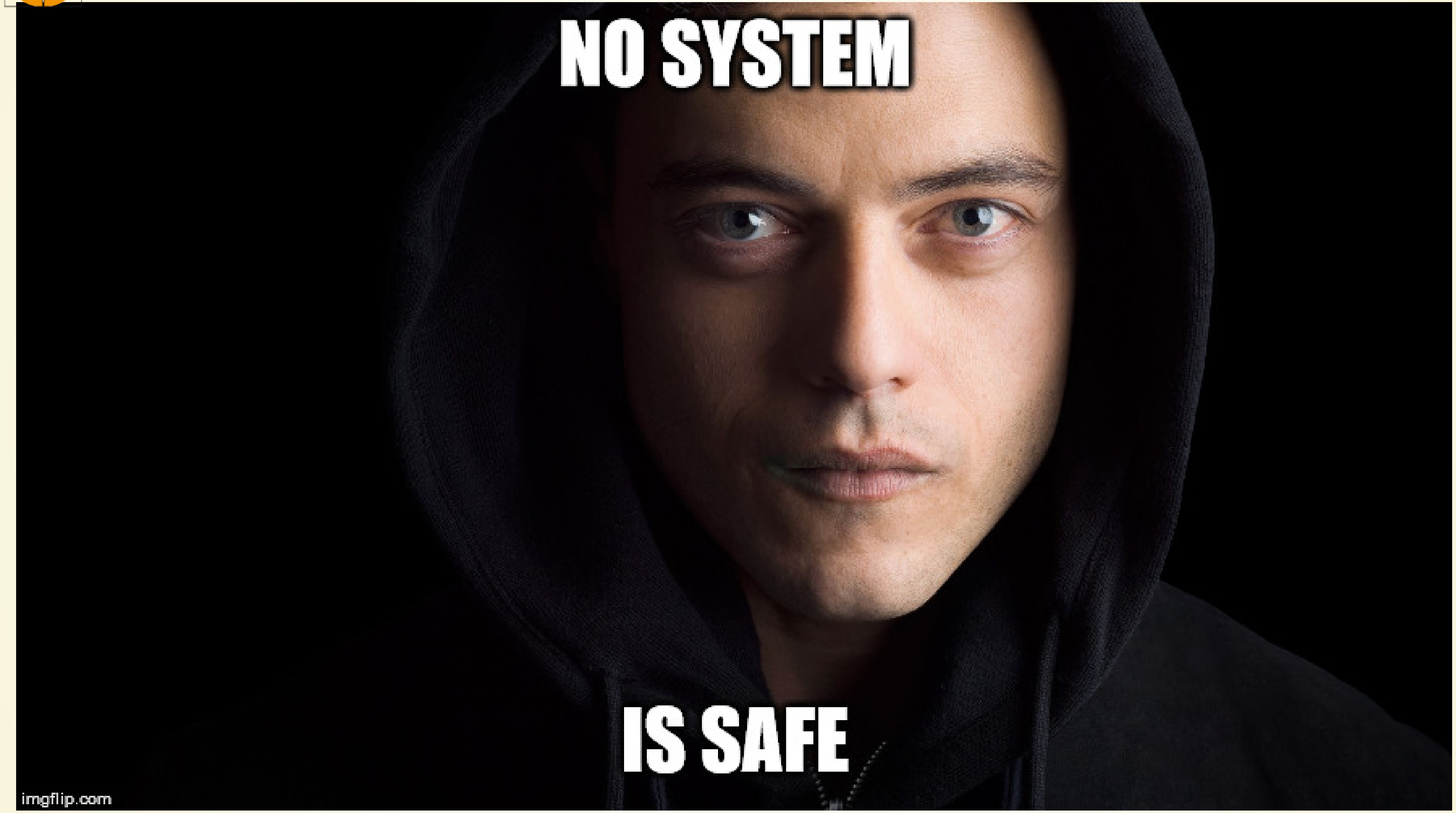
- Administrators – Full access
- Helpdesk – Read-Write access
- Any – Read access

Considerations

- Does the API gateway handle all the authentication and authorization stuff?
- If it does so, how do you enable your developers to test their services in a production-like environment?
- How fine-grained are your roles? (Organization structure!)
- How many roles do you define?
- Do you need an ACL approach (better you don’t!)
- What about other security layers? Firewalls, encryption, message authentication codes(MAC),…
- Never, never, never, never,… implement your own encryption algorithm!

Authentication & authorization basics

- When you think of authentication and authorization most people think of people consuming your service(s) but that’s just half the truth. If you’re building a microservice application your services also have authenticate and authorize to be able to call other services!
- First idea: well my services are protected by the firewall (internal perimeter) so… why should I care about authentication and authorization between them? What about man-in-the-middle attacks in the internal network?


NO SYSTEM
IS SAFE
imgflip.com

HTTP(S) Basic auth

- Client sends username and password in an HTTP header (typically the Authorization header) in the format <username><password> e.g. WhtZWQudGhlVGVycm9yaXN0OlNpbGVuY2UhMUsxbGxZMHU=
- Seems cryptographic, doesn't it? No! It's just a BASE64 encoded string…
- You'll need HTTPS for your cross services calls to ensure that the credentials are not getting stolen
- Public certificates are expensive (depending on the security level), the management of local PKI seems easier as it is (revocation of certs, …)
- How do you validate the username and the password? Local credentials store, LDAP, Single-Sign On (SSO) provider?

HTTP(S) Basic auth

- Solution 1: Use a SSO if possible (we’ll have a look at how to do this later on)
- Solution 2: Client certificates (we won’t go into this, ask if you would like to know how this works)

Service account

- Any service should have his own service account
- The password of a service account can be as complex as you can think as no one ever has to type or remember it (you got me right? never!)
- Change the passwords of the service accounts frequently (frequently does not mean every 5 years)
- A service account is always as limited as possible e.g. if the account does not have to log on to a machine it should not be able to do it!
- There are tools available to handle the creation, deletion and management of service accounts for different systems like Hashicorp Vault

Technische Hochschule Rosenheim
Technical University of Applied Sciences

Hash-based message authentication code (HMAC) over HTTP

A message authentication code is based on hashing operation. It takes the body, creates a hash of it based on a public key as kind of signature procedure and add the signature to the request. The server takes the signature validates it with his private key and drops the request if the validation failed.

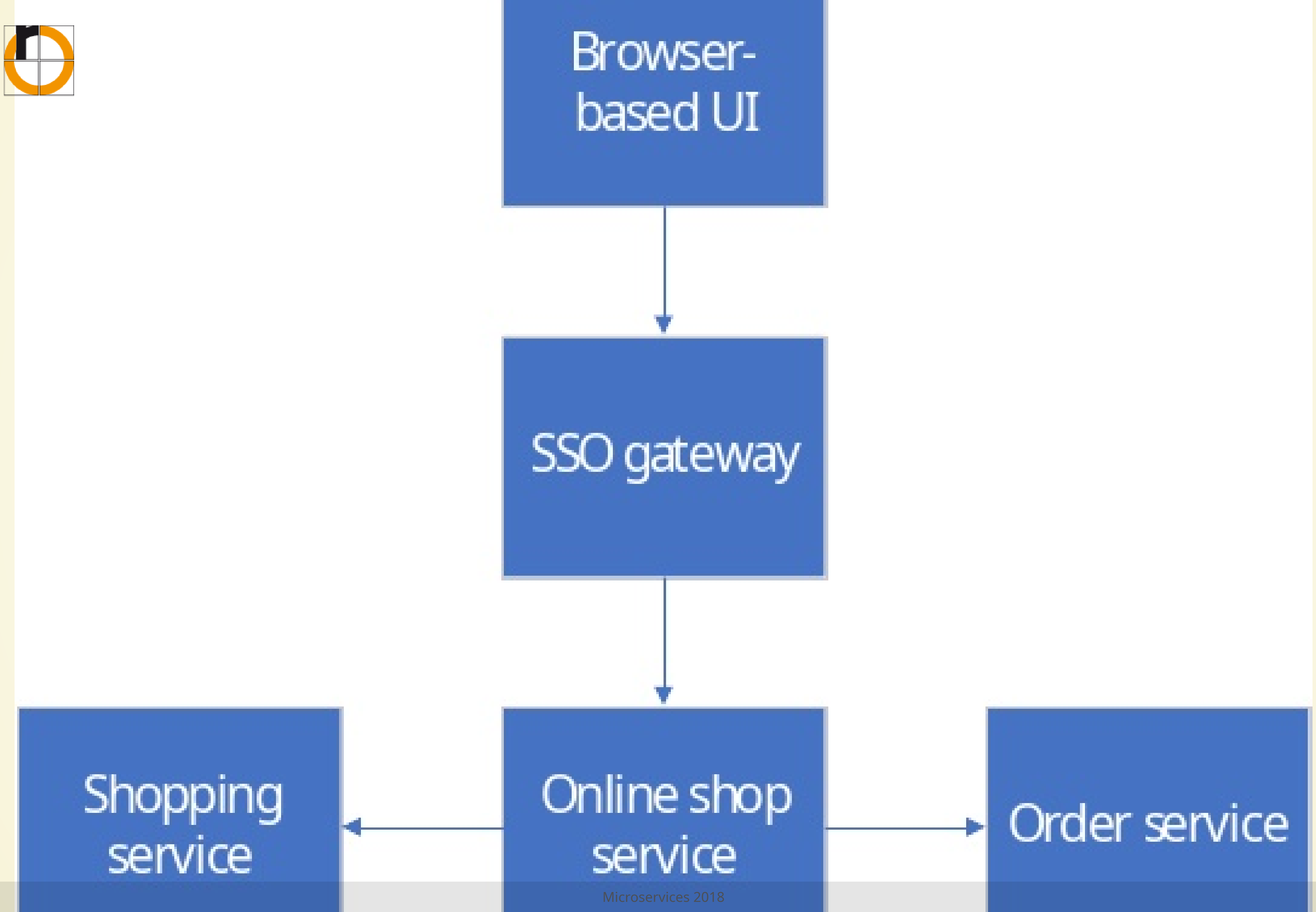
Pros and cons

- Advantages:
  - By checking the signature man-in-the-middle attacks are (mostly) impossible
  - May be faster that HTTPS
  - Traffic is cacheable as no HTTPS is required
- Disadvantages:
  - No default implementation
  - PKI needed to provide certificates (revocation problem)
  - Certificates have to be distributed over a secure connection
  - No encryption, just source validation!
- Alternative: JWT (JWS/JWE)

API keys

- Everyone has heard about so called "API keys"
- There’s no default mechanism to create an API key
- There may be just one hard coded API key, one API key per user or multiple API keys per user (with different access rights/roles/claims)
- You could use public-private keys to generate APIs or so many other procedures…
- We won’t look any further as it depends on your use case, technical infrastructure any more how you could implement an API key and there are other ways to solve this which are standardized (OAuth/OAuth2, OpenID Connect)

Confused deputy problem


Browser-based UI
SSO gateway
Shopping service
Online shop service
Order service

Problem description

- So you have authenticated a user by its username and password
- The username interacts with your online shop service – till now all is fine
- but now you online shop service has to contact the shipping service and the order service to get some additional information (you’re right, that’s evil! But let’s assume it for a moment.)

Problem description

- How should the shipping service or order service authenticate themselves the user when the information about the principal is gone in the SSO gateway or the scope of the token (we’ll have a look at scopes later on) does not include the shipping service and the order service?
- There’s no easy solution for this! You might choose to trust the online shop service, you might pass the access token through the SSO gateway or do a kind of cross service call

Advisories

- If you have to do password hashing and validation on your own, do it right. Use algorithms that are meant to be used for that:
  - PBKDF2: is designed to be slow on CPUs, unfortunately not on GPUs
  - BCrypt is not just optimized to be slow on CPUs but also to be memory-intensive so it should be harder to be implemented on GPUs and ASICs
- If you have to do encryption use the currently recommended encryption algorithms (AES256) and don’t implement on your own! If you are not sure what is recommended have a look here

Advisories

- If you have to do cryptographic signing (you also got it, don’t it on your own) if possible use elliptic-curve * cryptography! Otherwise RSA4096 is told to be safe for the next years (but not post-quantum) If you have to store certificates, have a look what your platform is offering you (e.g. Windows cert store, TPM,…)

Technische Hochschule Rosenheim
Technical University of Applied Sciences

Service-to-Service authentication & authorization - Microservices

HOW DO WE AUTHENTICATE OUR SERVICES IN X-SERVICE CALLS?
IP ADDRESS FILTER
HTTP BASIC
OPENID CONNECT TOKENS
imgflip.com

Single-Sign On (SSO)

- SSO is great!...but what is it?
- Most of the time you’re speaking about SSO you’re meaning ticket based SSO (there’s a second variant called local SSO but that’s more like a password safe than a real protocol)
- SSO means that you’re proving that you are who you are to one central party and that party hands you over a token
- Whenever you’re accessing another resource and you’re required to authenticate you’re doing it with the help of the SSO token to avoid passing your credentials to another resource than the central party

SSO - protocols

There are a few of SSO protocols which are well known (and some not so well known):

- OAuth
- OAuth2
- OpenID Connect
- Security Assertion Markup Language (SAML)
- Kerberos

OAuth2

- As OAuth has a few design flaws it was abandoned and replaced by OAuth2
- OAuth2 was developed as an authorization protocol (not for authentication!)
- OAuth2 is based on JSON (in the opposite to SAML which is based on XML and SOAP – is anyone remembering SOAP?)
- OAuth2 was originally developed by Twitter (2006) to delegate access rights to other applications
- Since 2012 it is standardized in the RFC 6749

OAuth2 - Roles

There are 4 roles in OAuth2:

- Resource owner
- Resource server
- Client
- Authorization server

Resource owener

Definition of the spec:

> An entity capable of granting access to a protected resource. When the resource owner is a person, it is referred to as an end-user.

So the resource owner is the one who’s authorizing an application to access his data – you’re well aware of this grant process as you all did it already multiple times!

Note that there may be edge-cases where a resource owner isn’t asked to grant any access. E.g in enterprise environments where the data is not owned by a single user but by an company and the company decides to share the data.

Resource server

Definition of the spec:

> The server hosting the protected resources, capable of accepting and responding to protected resource requests using access tokens.

Every application which stores data of you is a resource server. E.g. the internal GitLab instance stores your students contact data, your activity, your repositories and many more. If you’re installing the LabCoat app on your mobile the app needs access to your GitLab account to be able to display your repositories and so on, so the GitLab server is the resource server. But when you’re importing a GitHub repository to GitLab, GitLab needs access to your GitHub account so a resource server may also be a client at the same.

Client

Definition of the spec:

> An application making protected resource requests on behalf of the resource owner and with its authorization. The term „client“ does not imply any particular implementation characteristics (e.g., whether the application executes on a server, a desktop, or other devices).
>
> - e.g., whether the application executes on a server, a desktop, or other devices

- According to the spec a client may be any application that requests data.
- As already mentioned a resource server may also be a client at the same time.

Authorization server

Definition of the spec:

> The server issuing access tokens to the client after successfully authenticating the resource owner and obtaining authorization.

- The authorization server is the central party handing you over an access token (the term access token is special in OAuth2)
- The authentication process is not specified so it may be with username and password, username, password and OTP, smartcard,…

Scopes

- Scopes are defining a kind of roles on APIs
- A client has to require scopes to be able to access the APIs grouped in the scope
- Multiple resource servers may share one scope but it’s also possible to have multiple scopes on one resource server

See also GitHub docs about scopes for OAuth Apps.

OAuth2 - Tokens

OAuth2 differs between the following two types of tokens:

- Access tokens
- Refresh tokens

Access tokens

- Access tokens are short living (between a few minutes and a few hours)
- Access tokens are required by the client to access resources served by resource servers
- The access token contains the granted scopes
- The OAuth2 spec nor does specify how a access token should look like (JWT, Bearer,…) neither does it specify what exactly should be contained in it. So it depends on the provider of the token
- But when this token exceeds within a few minutes, why isn’t the client repeatedly asking for my credentials you ask?

REFRESH TOKENS
imgflip.com
H HD
STORY.COM

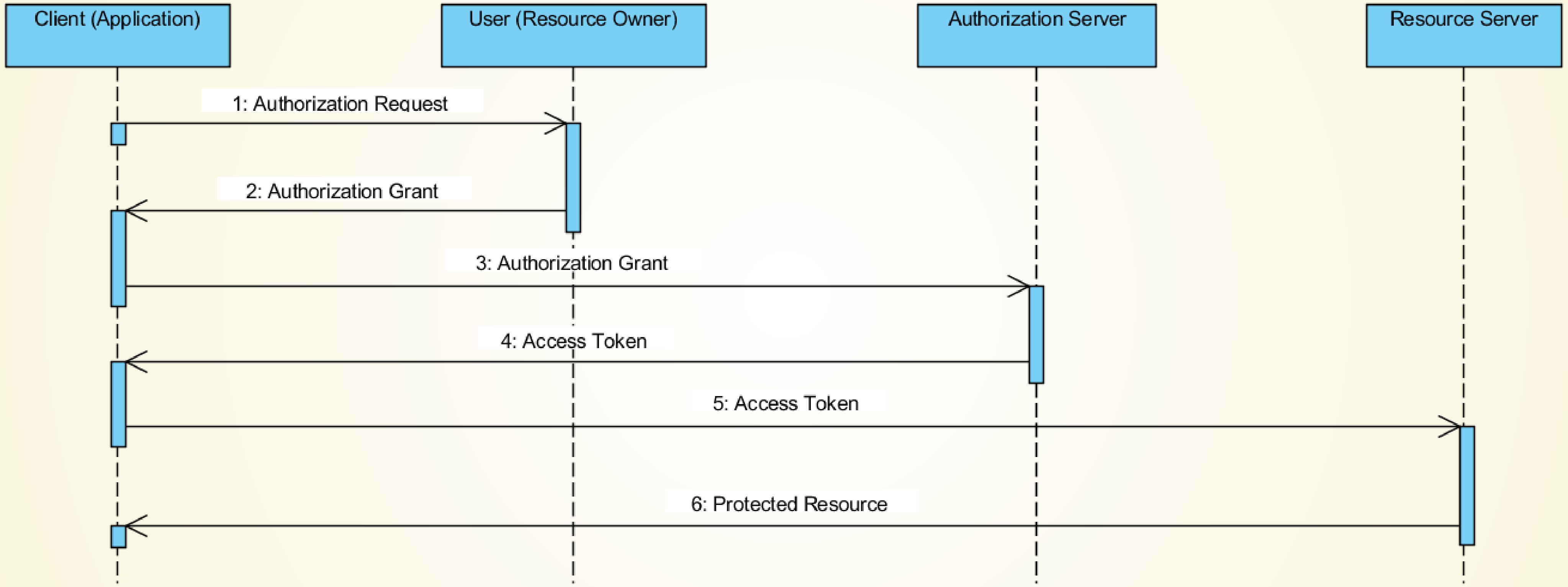
Refresh tokens

- Depending on the flow a client receives with the access a second token called refresh token
- In contrast to the access token the refresh token a very long lifetime (can be over a year)
- The refresh token also contains the information which scopes where granted and for which client the access was granted
- The refresh token has to be stored secured in the client
- When the access token exceeds the client is able to fetch a new one by passing the refresh token to the authorization server
- On the one hand that’s pretty nice because whenever a token is lost an attacker can’t do much with it because of the short lifetime but on the other hand if the refresh token is stolen you have real problem!

OAuth2 - Client registration

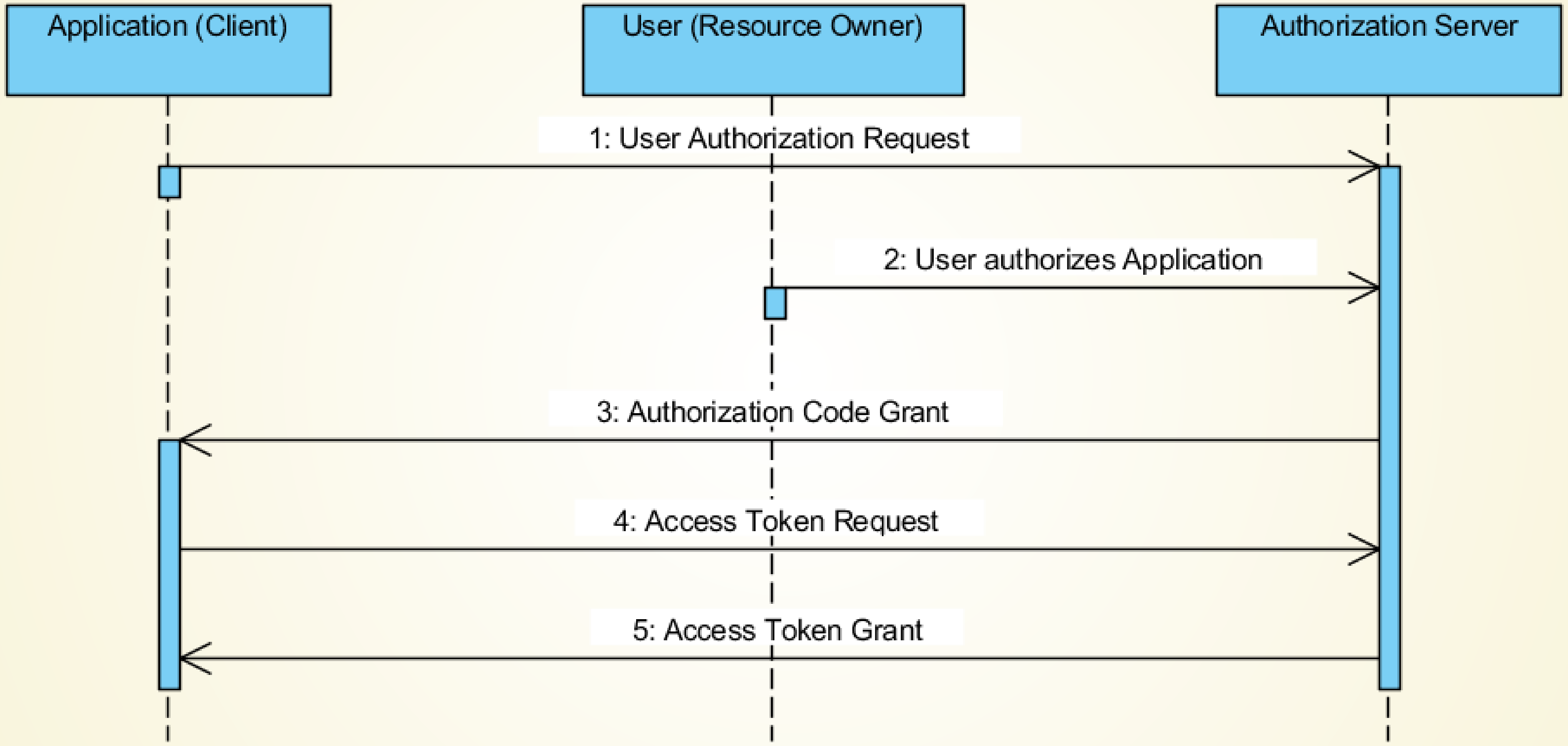
- To be able to interact with an authorization server it’s not enough to just know its address but the client has to be registered within the authorization server
- The registration at an authorization server typically requires two parts:
  - The client type (confidential or public)
  - A redirect URI
- OAuth2 providers are allowed to require more information for the registration, e.g.:
  - Name of the application
  - Description
  - Logo
  - License terms
  - …
- When the registration is completed you’re getting a client id and a client secret where the client id is kind of a username and the client secret is kind of a password (and they should be handled equivalently)

OAuth2 - flows


Client (Application)
User (Resource Owner)
Authorization Server
Resource Server
1: Authorization Request
2: Authorization Grant
3: Authorization Grant
4: Access Token
5: Access Token
6: Protected Resource

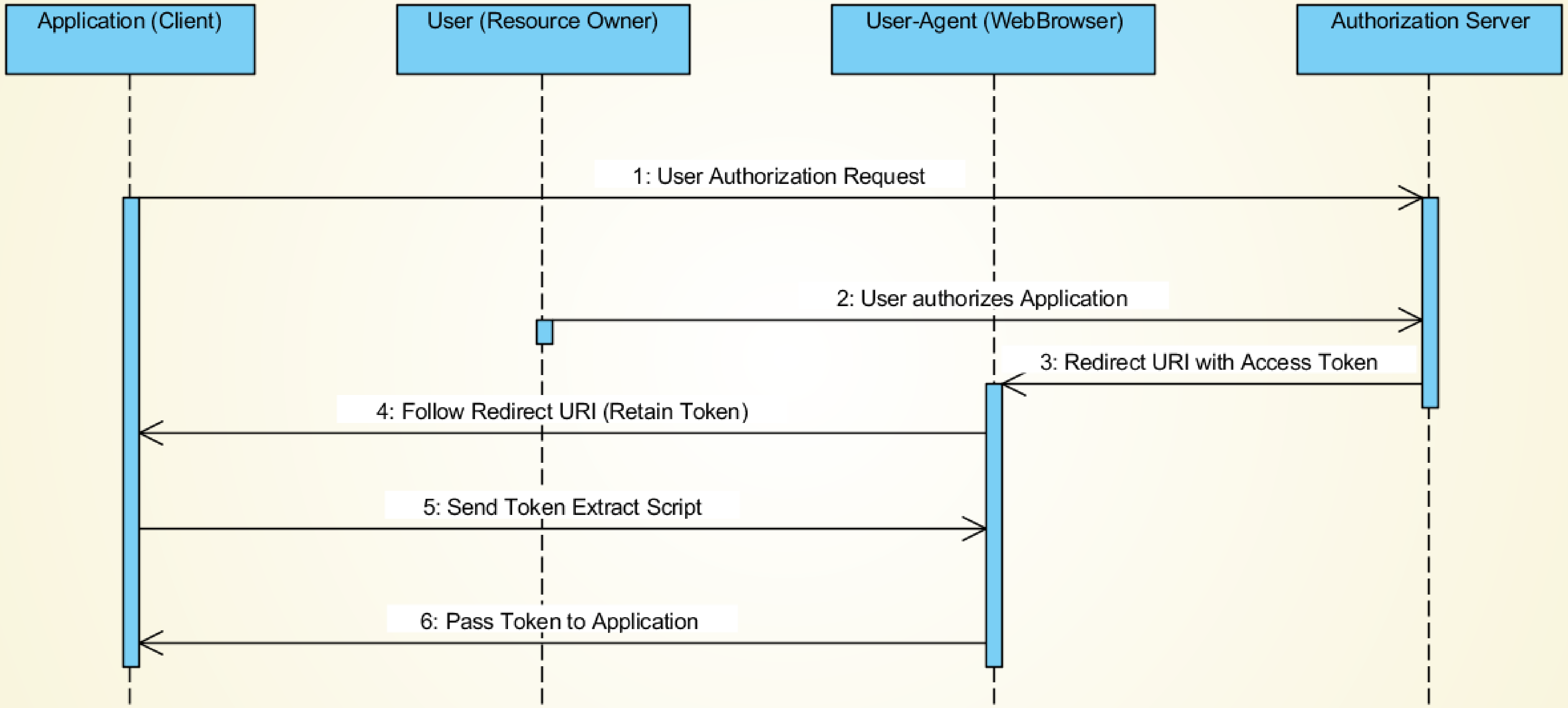
Overview

- Flows are describing the interaction between the different roles specified in OAuth2
- The abstract flow is meant as an orientation and can be interpreted likes this:
    1. The application (or client) asks for access to resources/scopes of the user
    2. User grants access (application gets an so called authorization grant)
    3. The application claims for an access token by sending the authorization grant and its identity (client id and client secret) to the authorization server
    4. If client id, client secret and authorization grant are valid the application get’s an access token
    5. The application can now access resources of the user at the resource server
- OAuth2 is aware of the following Flows:
    - Authorization Code
    - Implicit
    - Resource Owner Password Credentials
    - Client Credentials


Application (Client)
User (Resource Owner)
Authorization Server
1: User Authorization Request
2: User authorizes Application
3: Authorization Code Grant
4: Access Token Request
5: Access Token Grant

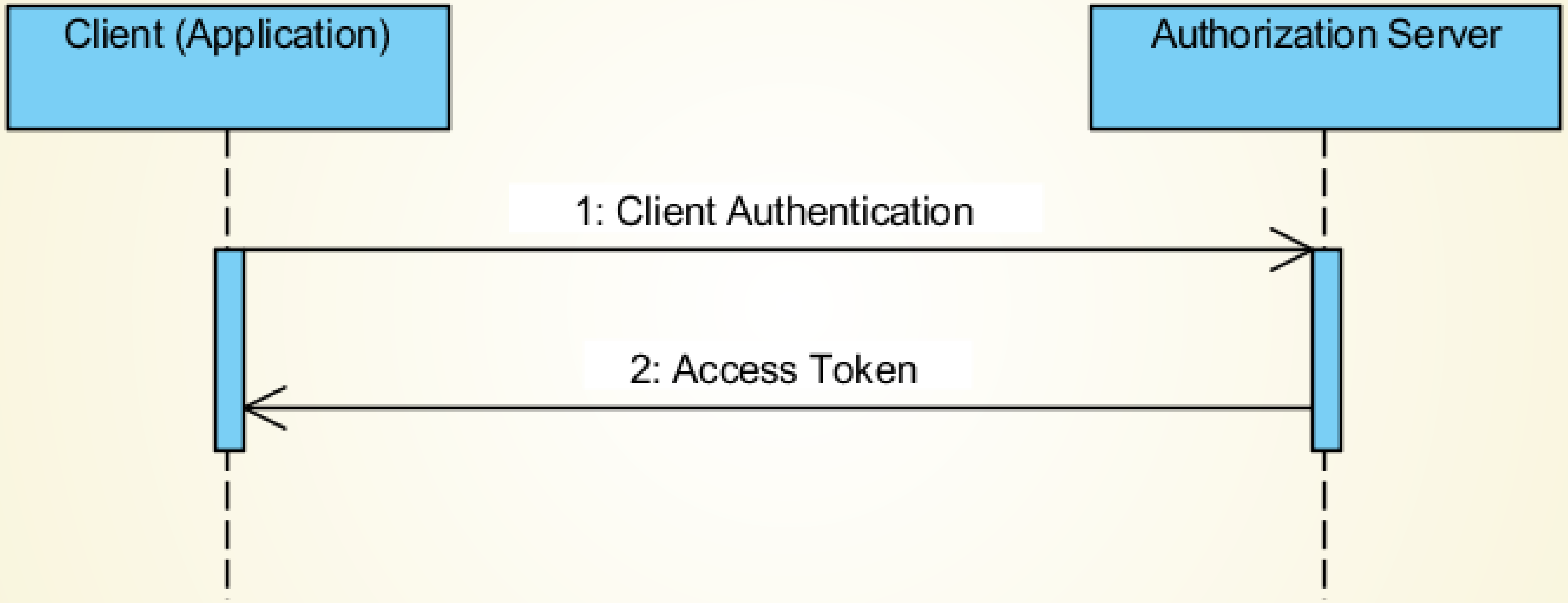
Authorization code flow

- The Authorization Code flow requires the client to include its client secret when it claims for an access token. Because of that this flow is mostly used within classical web applications like JSP/JSF or ASP.NET MVC where the backend can store the client secret securely
- A prerequisite for this flow is that the application can force the user agent to follow redirects to route the user agent at first to the authorization server and afterwards back to the client application including the authorization grant included as query param
- The flow is proceeded like this:
    1. The client claims for access to user resources by routing the user to the authorization server
    2. User (resource owner) grants access
    3. Authorization grant gets redirected to a previously registered callback URL
    4. Client can acquire an access token by sending the authorization grant, the client id and the client secret to the authorization server
    5. If all params were valid the client gets an access token (and a refresh token)


Application (Client)
User (Resource Owner)
User-Agent (WebBrowser)
Authorization Server
1: User Authorization Request
2: User authorizes Application
3: Redirect URI with Access Token
4: Follow Redirect URI (Retain Token)
5: Send Token Extract Script
6: Pass Token to Application

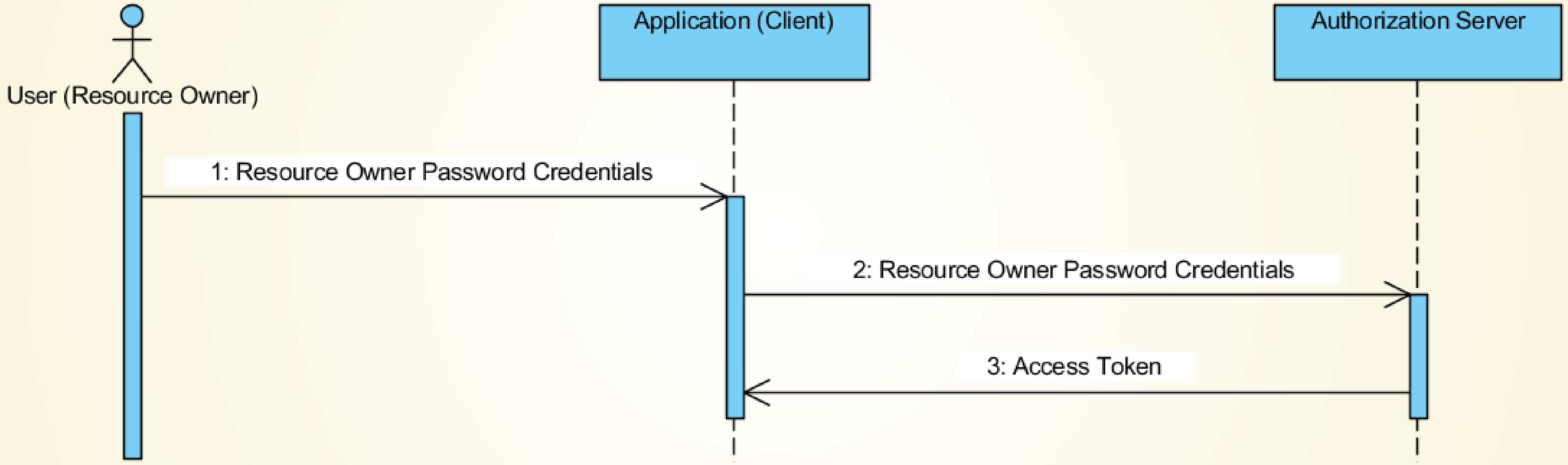
Implicit flow

- The Implicit flow is designed for mobile and browser apps where confidentiality can not be guaranteed
- Within the Implicit flow the access token is directly handed over to the user agent of the user
- Because of the missing confidentiality there’s no refresh token when the Implicit flow is used
- The flow works like this:
  1. The client redirects the user agent to the authorization server for claiming access
  2. User has to grant the access
  3. User agent is redirected to an previously registered callback URL including the access token (no authorization grant!)
  4. User agent follows the redirect
  5. Script in the browser is triggered by the URL and extracts the access token
  6. Script passes token to the application


Client (Application)
Authorization Server
1: Client Authentication
2: Access Token

Client credentials flow

- The client credentials flow is designed for cross service calls
- Furthermore this workflow can be used if the service is updating its metadata stored at the authorization server
- Within this flow service A creates an access token with his own client credentials (client id and client secret)
- The flow works like this:
    1. Client sends his own client id and client secret to the authorization server
    2. Client receives access token and can access his own API


User (Resource Owner)
Application (Client)
Authorization Server
1: Resource Owner Password Credentials
2: Resource Owner Password Credentials
3: Access Token

Resource owner password credentials flow

- The resource owner password credentials flow is based on letting the user pass his credentials to the client and the client sending his credentials to the authorization server.
- Because most OAuth2 providers don’t want their users to expose their credentials to 3rd party apps this flow is often not implemented
- The flow works like this:
  1. User passes his credentials to the client
  2. Client passes the credentials to the authorization server and applies for a access token
  3. If the credentials were valid the client receives an access token

OAuth2 - conclusion

- Scopes are a nice abstraction to tell the user which permissions he's granting the client
- The predefined flows are containing everything you need for a microservice application
- OAuth2 is not an identity framework but an authorization framework
- It can be used as an identity framework too but a developer doing this has to know what he’s doing as there may occur security leaks if he doesn’t

OpenID Connect (OIDC)

- OpenID Connect was designed a an identity framework
- It is based on OAuth2
- OpenID connect is much more restrictive than OAuth2 as it specifies a token format, some default claims a token has to contain, API endpoints with special routes that have to exist and much more
- Special attention was paid on interoperability within different providers

Components/specs

OpenID Connect consists of the following three parts:

- Core (token specs, flows, endpoints, claims, implementation advises and security remarks)
- Discovery (assists users to log on to an application with an account of a provider completely unknown prior to the first log on of the user, relies on WebFinger, exposes the endpoints /.well-known/webfinger and /.well-known/openid-configuration for dynamic configuration of a new application)
- Dynamic client registration (without this concept a discovery of a new OpenID Connect provider would be useless. Enables applications to register themselves on the fly if required)

Tokens

- As already mentioned OpenID Connect is much more restrictive than OAuth2 and defines the concrete format and even the content of all tokens
- OpenID Connect uses JWT (but you can choose between JWE and JWS) for all tokens
- OpenID Connect is aware of the following tokens:
  - Identity token
  - Access token
  - Refresh token

Identity token

- The identity token is kind of a passport of a user
- It has to contain 20 claims e.g.:
  - Sub
  - Name
  - Nickname
  - Prefered_username
  - Profile
  - Email
  - …
- The identity token can be used for many use cases, e.g.:
  - Stateless sessions
  - Identification at 3rd party apps
  - Key exchange (to get an access token)

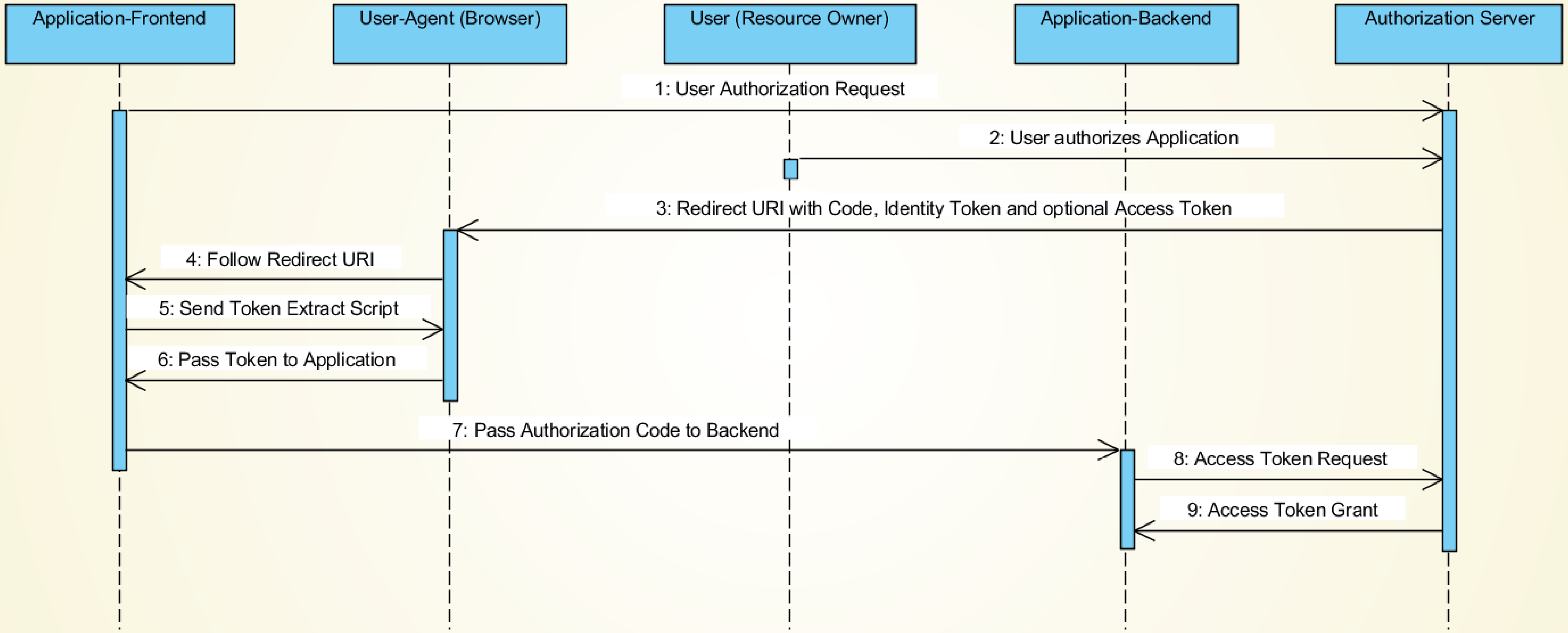
Access token

- Like OAuth2 OpenID Connect specifies an access token
- The functionality is equivalent to OAuth2
- If the app only claims for the scope "openid" the response will only contain an identity token but no access token

Flows

OpenID Connect specifies just 3 flows:

- Authorization Code (is like in OAuth2)
- Implicit (is like in OAuth2)
- Hybrid (this one is a little bit tricky)


Application-Frontend
User-Agent (Browser)
User (Resource Owner)
Application-Backend
Authorization Server
1: User Authorization Request
2: User authorizes Application
3: Redirect URI with Code, Identity Token and optional Access Token
4: Follow Redirect URI
5: Send Token Extract Script
6: Pass Token to Application
7: Pass Authorization Code to Backend
8: Access Token Request
9: Access Token Grant

Hybrid flow

- The Hybrid flow is mixture of Authorization Code and Implicit flow
- It’s try to compensate the disadvantages of the Implicit flow by added a refresh token which is only available in the backend of an application but not in the (unsecure) frontend
- The flow works like this:
    1. Frontend applies for access to resources of the resource owner
    2. User grants access
    3. Authorization server redirects user agent to previously registered callback URI
    4. User agent follows redirect including the encoded identity token and optionally encoded access token and authorization code
    5. Frontend extracts identity token and optionally the access token and authorization code
    6. Script passes extracted token(s) to frontend application
    7. Optionally: backend retrieves refresh token with with the authorization code
    8. If params are valid backend retrieves refresh token

Technische Hochschule Rosenheim
Technical University of Applied Sciences

OpenID Connect flows overview

Property	Authorization code flow	Implicit flow	Hybrid flow
All tokens from authorization endpoint	✓	✓	✗
All tokens from token endpoint	✓	✗	✗
Tokens exposed to user agent	✓	✗	✗
Client can be authenticated	✓	✗	✓
Refresh token possible	✓	✗	✓
Communication in one roundtrip	✗	✓	✗
Communication mostly server to server	✓	✗	Varies