



Microservice Security mit OAuth2



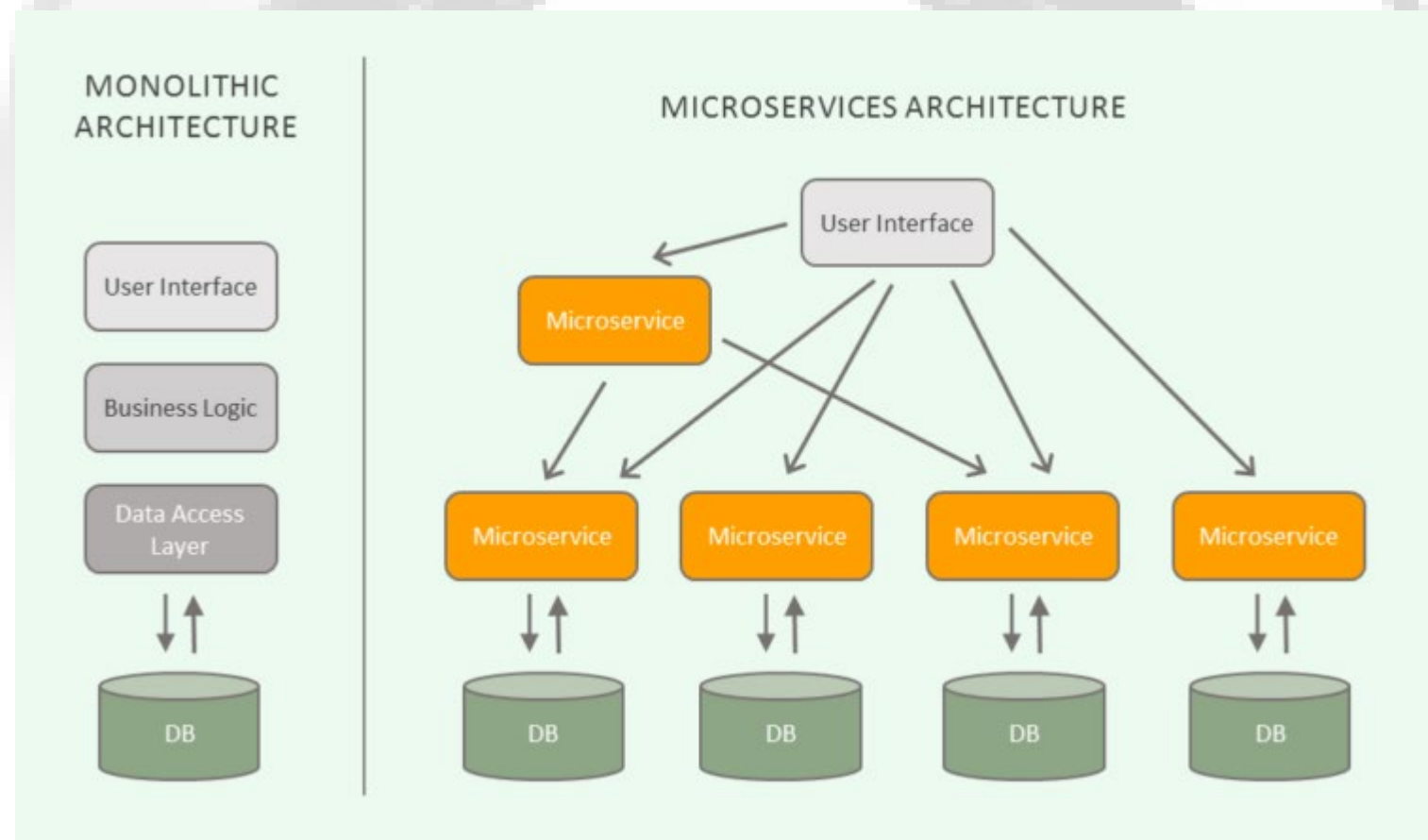
Microservice Security

Monolithisches System
Microservices

-> ein System

-> ein Zugangspunkt

-> mehrere Systeme -> mehrere Zugangspunkte



[Nordström 2016]

Was ist OAuth2?

- OAuth **O**pen **A**uthorization
- Version 2 von OAuth Protokoll
- OAuth Applikationen können Standard verwenden, um Client Anwendungen einen “sicheren delegierten Zugang” zur Verfügung zu stellen
- OAuth arbeitet über HTTP und autorisiert Geräte, APIs, Server und Anwendungen mit Access Tokens im Gegensatz zu Credentials
- OAuth Protokoll, unterstützt **Autorisierungsworkflow**
bietet Möglichkeit zur Prüfung, ob Client berechtigt ist, etwas Bestimmtes zu tun.

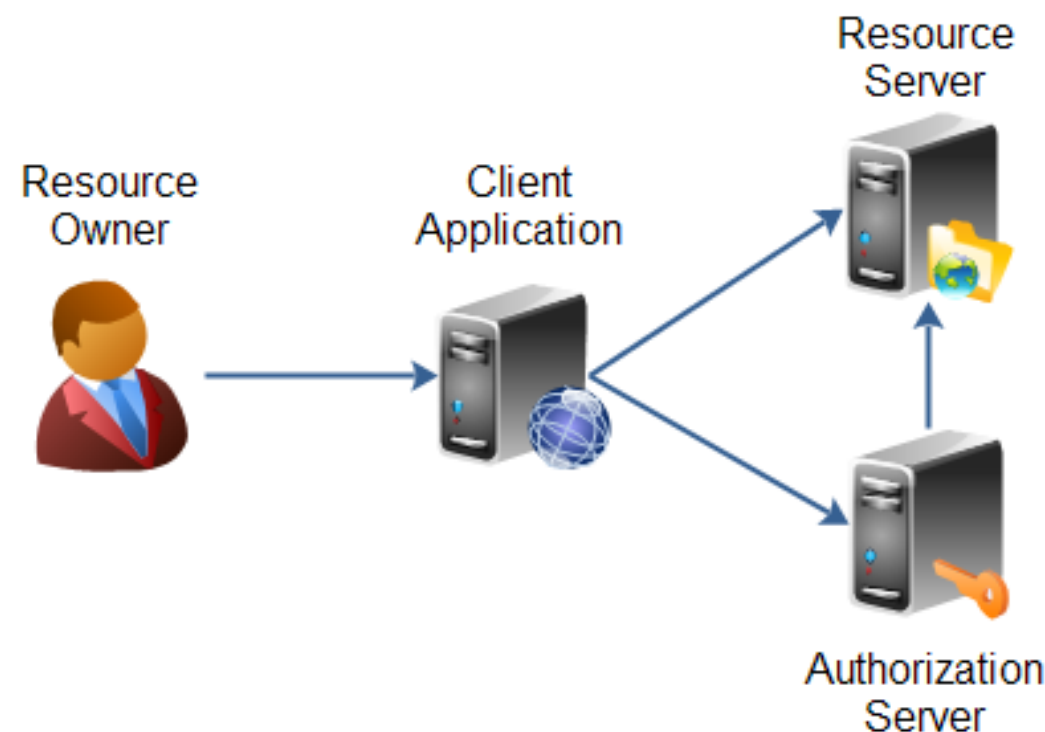
Inhalt – Microservice Security

- OAuth2 Protokoll
 - OAuth2 Rollen
 - OAuth2 Grant Types
 - Funktionsweise OAuth2
 - Authorization Code Grant Type
 - Implicit Code Grant Type
 - Resource Owner Password Credentials Grant Type
 - Client Credentials Grant Type
 - OAuth2 Endpoints
- Überblick Spring Security
- Beispiel Implementierung der verschiedenen Komponenten
- Laboraufgabe

OAuth2 Rollen

Rollen:

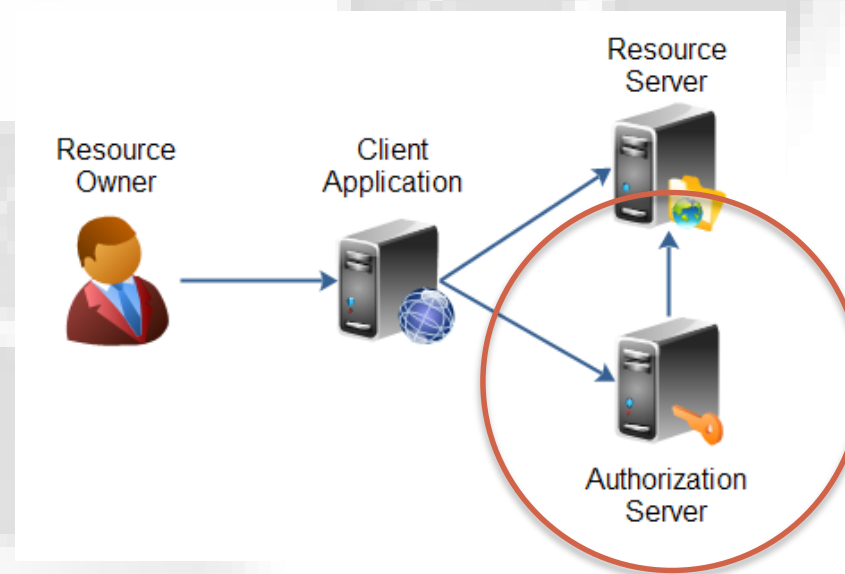
- Resource Owner: Besitzer der geschützten Daten
- Client Application: Anwendung, die die geschützten Daten anfordert
- Authorization Server: Server, der die Autorisierung durchführt
- Resource Server: Server, auf dem die Daten gespeichert sind



[Jenkov 2014]

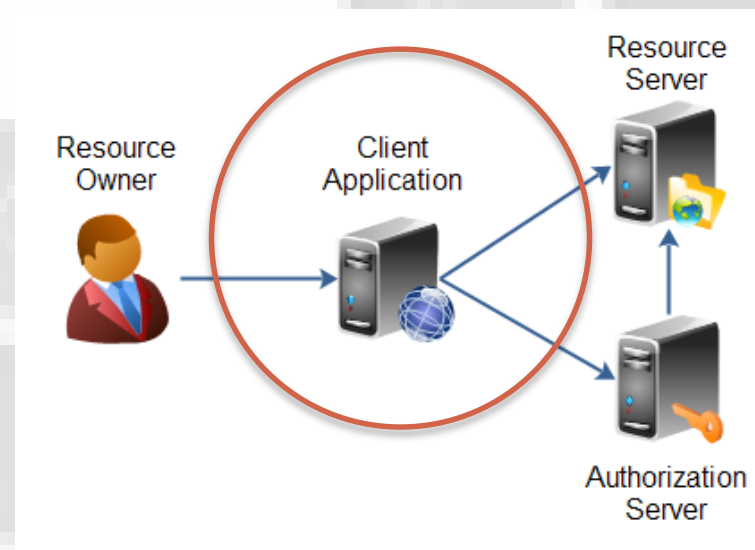
Rolle des Autorisierungsservers

- Client Application authentifizieren
- Interface für User (Resource Owner), um Client Application zu erlauben auf Daten zugreifen zu dürfen
- Erzeugt Token und liefert Token an Client Application aus
- User (Resource Owner) authentifizieren



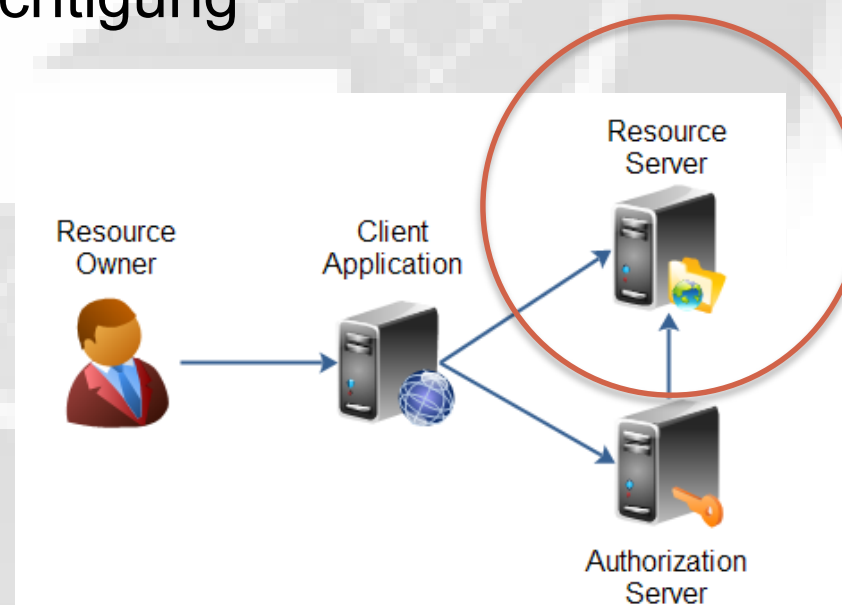
Rolle der Client Anwendung

- Erhält ein Token vom Autorisierungsserver
- Benutzt das Token, um auf die Ressourcen auf dem Resource Server zuzugreifen
- Kennt keine User Credentials, abhängig von Authorisierungsflow
- Registrieren am Autorisierungsserver notwendig, d.h. Anwendung hat client_id, client_secret, redirect URL



Rolle des Resource Servers

- Jeder Request muss Token enthalten
- Extrahiert das Token aus dem Request
- Entscheidet über Datenzugriffe
- Sendet 403 (FORBIDDEN), falls Token nicht ausreichend
- Sendet Daten bei ausreichender Berechtigung



OAuth2 Grant Types

Grant Type:

Modus zum Erzeugen von Tokens

4 unterschiedliche Modi:

für unterschiedliche Anwendungen, je nach zu implementierendem Service

- **Authorization Code Grant Type**
für third-party Anwendungen
- **Implicit Grant Type**
ähnlich wie oben, für Browser basierte oder mobile Anwendungen
- **Client Credentials Grant Type**
für Anwendungen von Maschine zu Maschine
- **Resource Owner Password Credentials Grant Type**
nur für vertrauenswürdige Anwendungen

Funktionsweise OAuth2

- Client muss sich beim Anbieter der API registrieren
- Anbieter weist eine Client Id und ein Client Secret zu

z.B. Client Infos für Facebook:

client:

clientId: 233668646673605

clientSecret: 33b17e044ee6a4fa383f46ec6e28ea1d

z.B. für eigene Testanwendung:

client:

clientId: my-clientId

clientSecret: my-client-secret

Beispiel Ablauf Registrierung:

Facebook:

<https://auth0.com/docs/connections/social/facebook>

Microsoft:

<https://docs.microsoft.com/de-de/azure/active-directory/develop/v1-protocols-oauth-code>

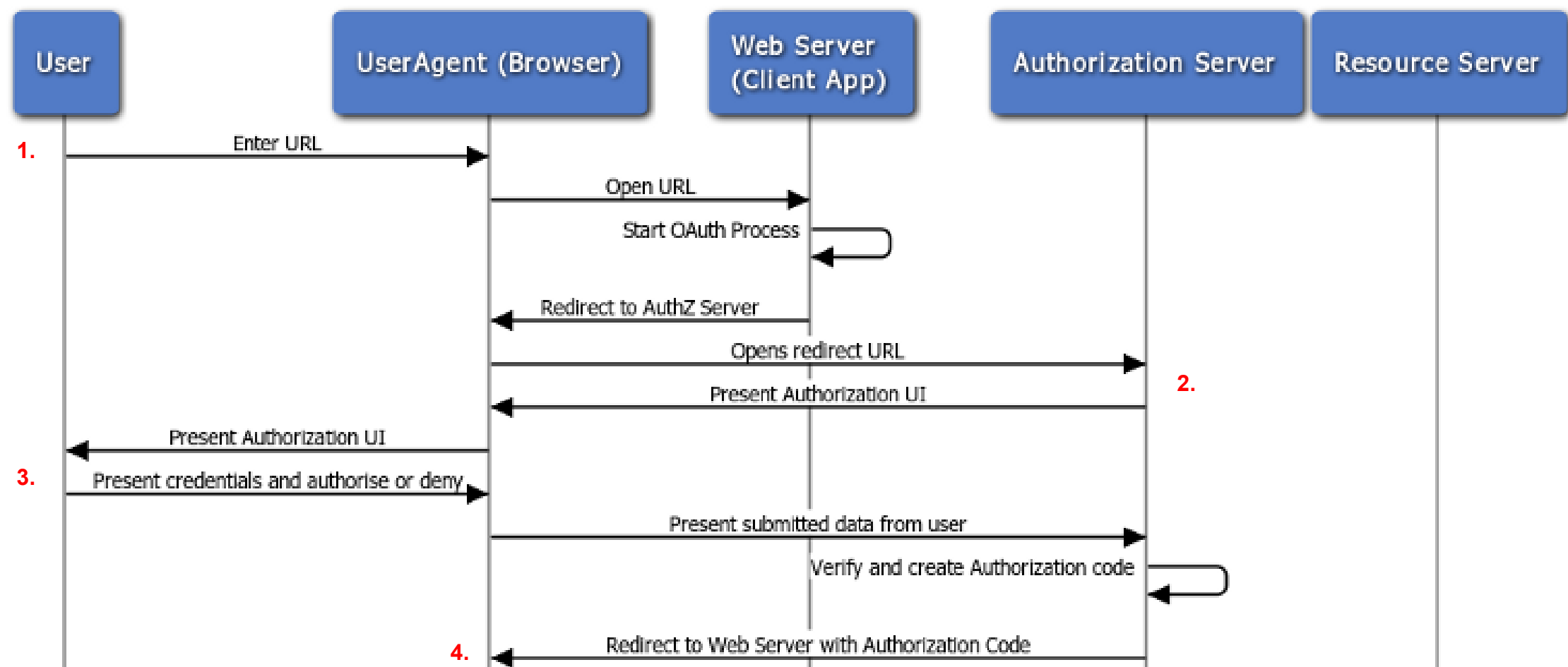
Google:

<https://developers.google.com/identity/protocols/OAuth2>

Authorization Code Grant Type

Ablauf Teil 1:

1. User besucht die Webseite
(Client App fordert User auf, sich über einen Autorisierungsserver anzumelden)
2. Redirect des Users auf die Seite des Autorisierungsserver
Validierung der Parameter durch Autorisierungsserver
anzeigen der Autorisierungsseite,
3. Einloggen am Autorisierungsserver und der Anwendung den Zugriff erlauben
4. Bei Zustimmung, redirect auf die Redirect URI der Client App

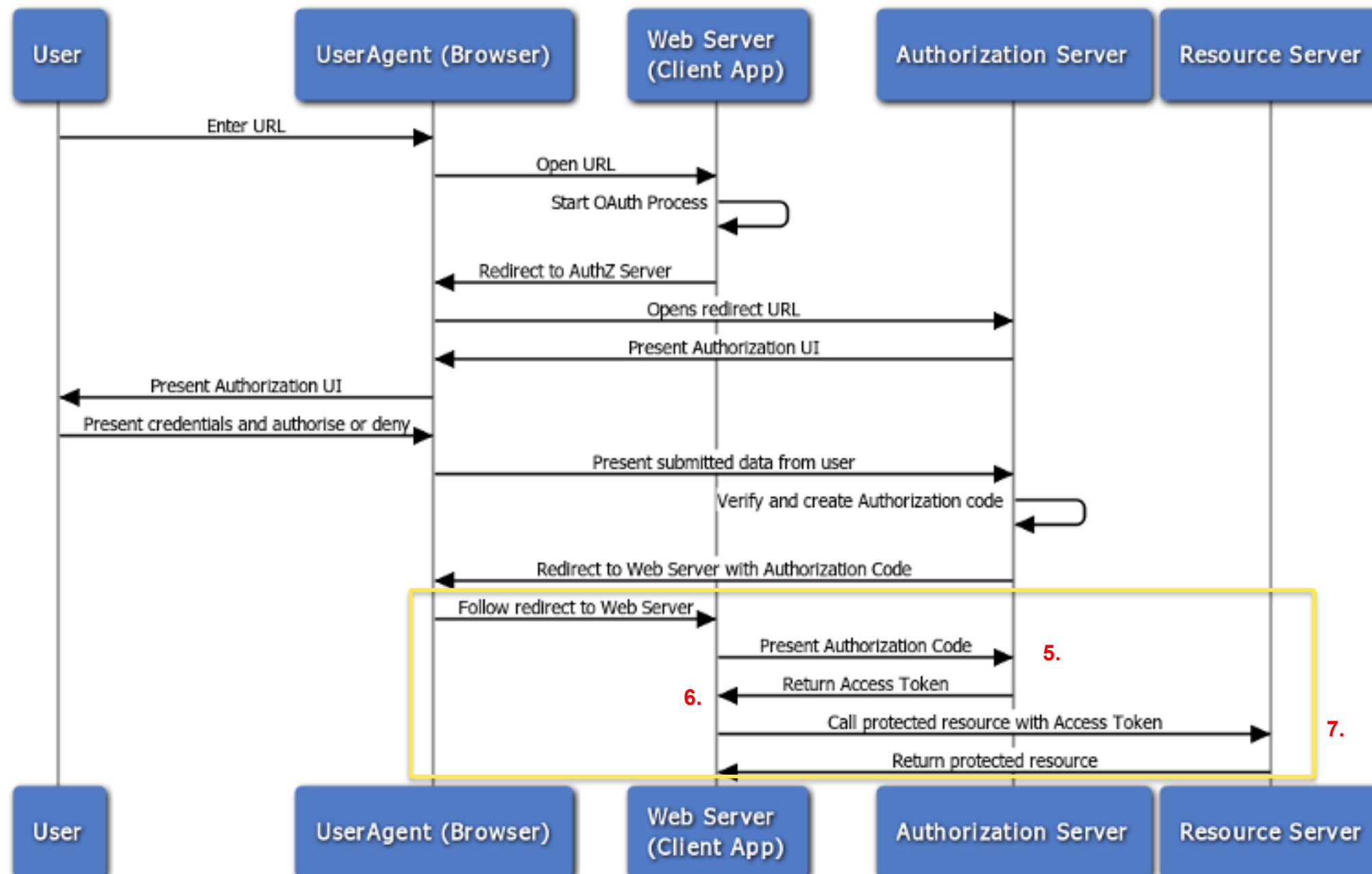


[APIOAuth]

Authorization Code Grant Type

Ablauf Teil 2:

5. Client App sendet einen POST-Request an den Autorisierungsserver
6. Autorisierungsserver antwortet mit JSON Objekt mit Access Token
7. Client App sendet Request mit Access Token im Header an Resource Server

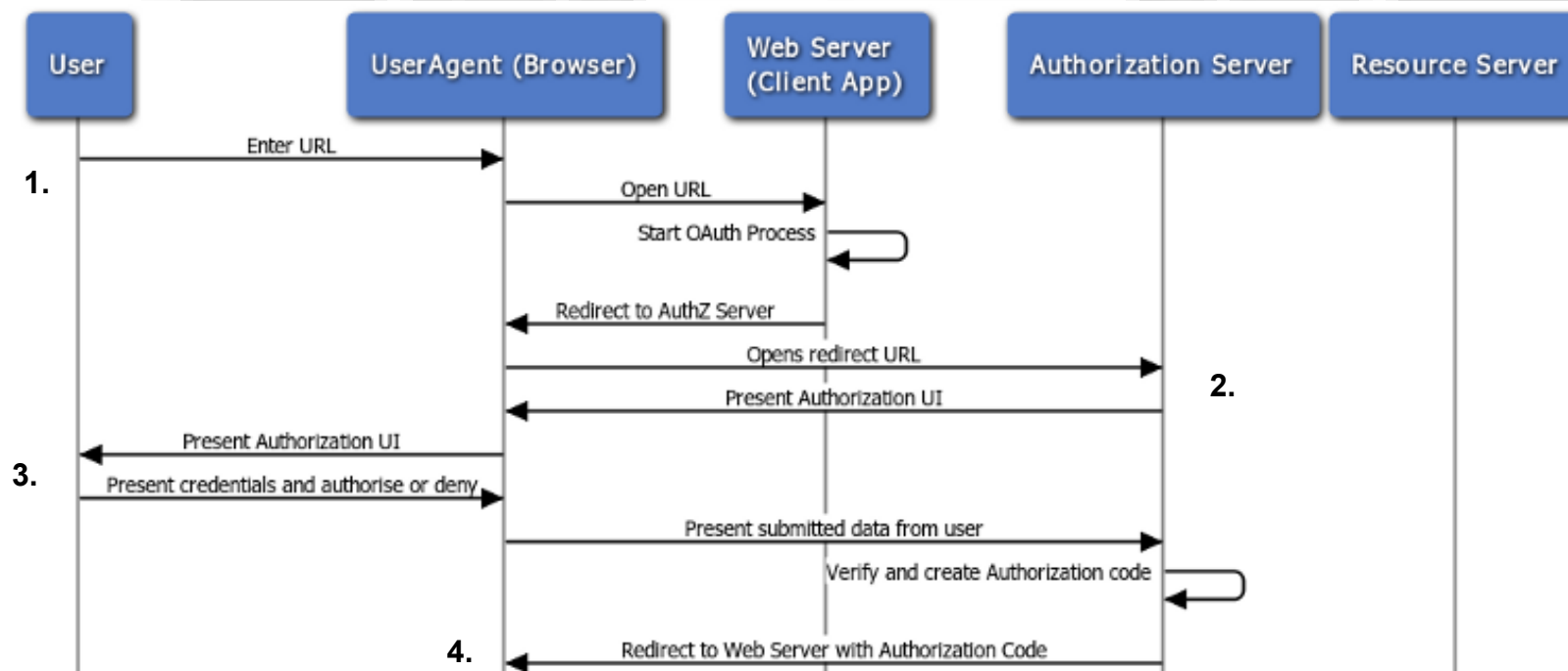


[APIOAuth]

Authorization Code Grant Type

Ablauf Teil 1:

1. User besucht die Webseite
(Client App fordert User auf, sich über einen Autorisierungsserver anzumelden)
 2. Redirect des Users auf die Seite des Autorisierungsserver, Parameter im Query String
 - **response_type** Wert: **"code"**
 - **client_id** Wert: zugewiesene client_id
 - **redirect_uri** optional Wert: registrierte URL
 - **scope** optional Wert: möglicher Scope des Requests
 - **state** optional, empfohlen Wert: Client Token
- Validierung der Parameter durch Autorisierungsserver,
anzeigen der Autorisierungsseite
3. Einloggen des Users am Autorisierungsserver und der Client App den Zugriff erlauben
 4. Bei Zustimmung, redirect auf die Redirect URI des Clients, Parameter im Query String:
 - **code** Wert: Autorisierungscode
 - **state** Wert: State Parameter aus dem ursprünglichen Request

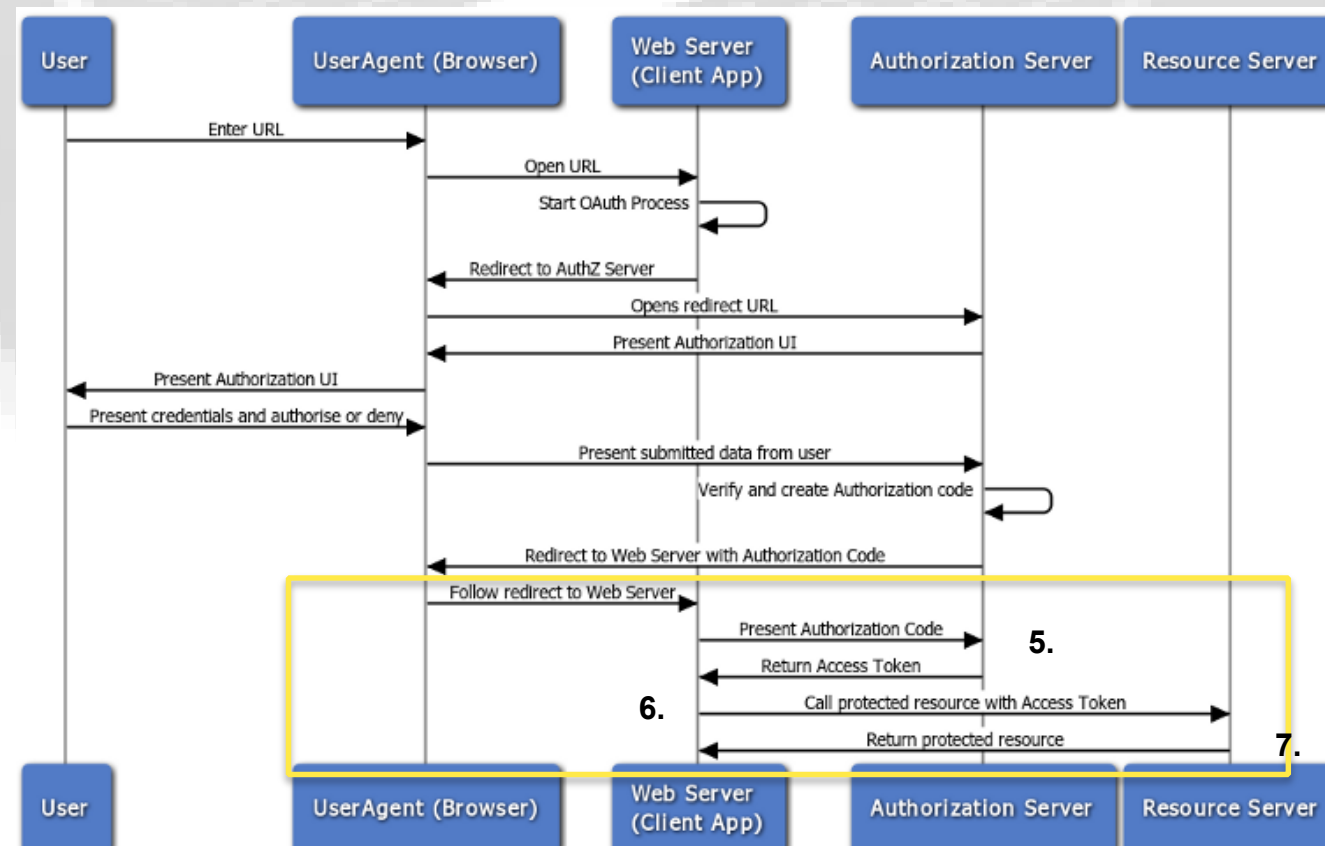


[APIOAuth]

Authorization Code Grant Type

Ablauf Teil 2:

5. Client App sendet einen POST-Request an den Autorisierungsserver, Parameter im Query-String
 - **grant_type** Wert: **“authorization_code”**
 - **client_id** Wert: Client Identifier
 - **client_secret** Wert: Client Secret
 - **redirect_uri** Wert: die selbe Redirect URI , wie in Teil1
 - **code** Wert: Autorisierungscode aus dem Query String
6. Autorisierungsserver antwortet mit JSON Objekt mit Properties
 - **token_type** Wert: meistens **“Bearer”**
 - **expires_in** Wert: Integer (TTL Time of Life des Access Tokens)
 - **access_token** Wert: Access Token
 - **refresh_token** Wert: Refresh Token, kann benutzt werden, um neues Access Token anzufordern
7. Client App sendet Request mit Access Token im Header an Resource Server

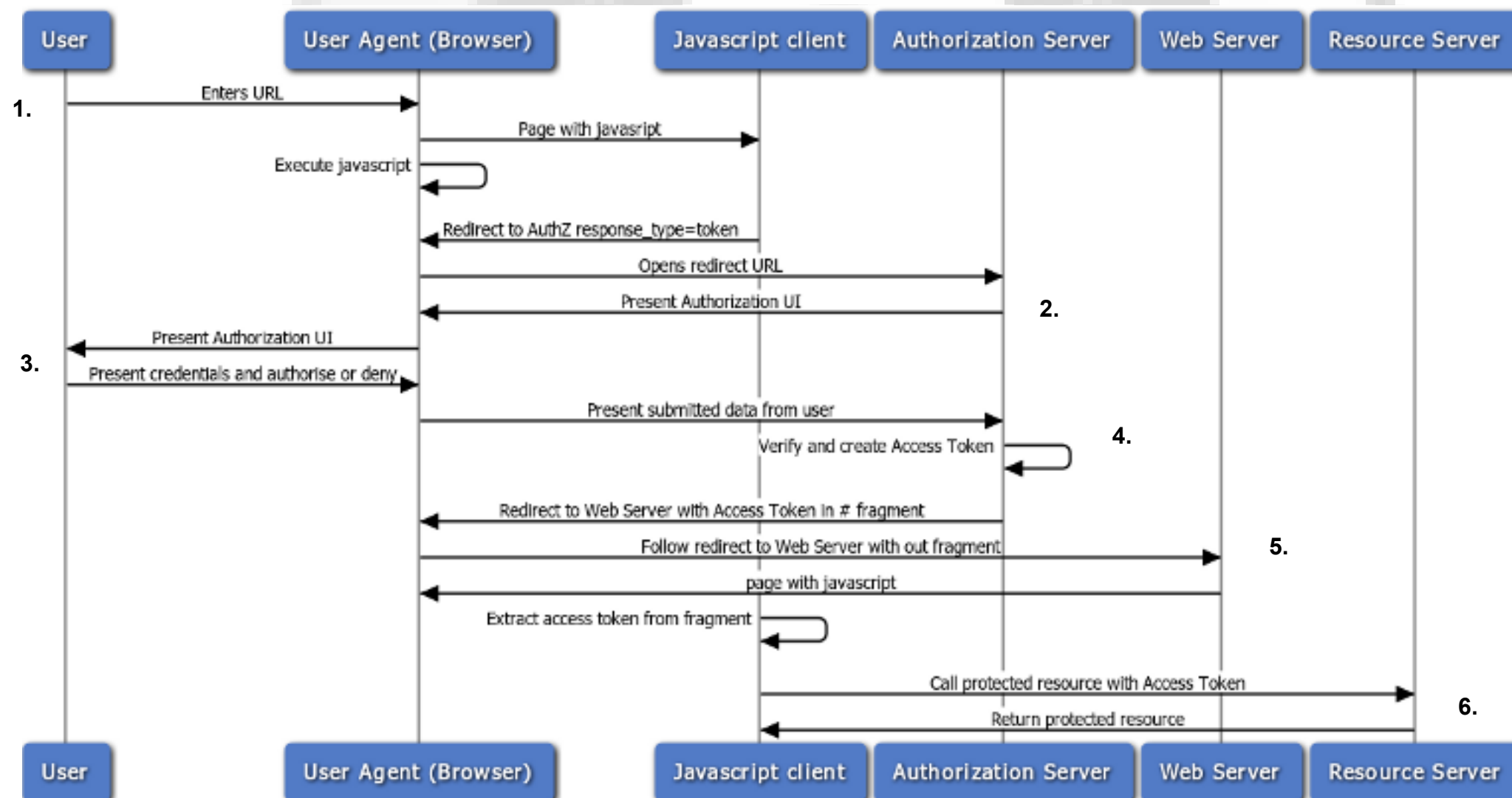


[APIOAuth]

Implicit Code Grant Type

Ablauf:

1. User besucht die Webseite
2. Redirect des Users auf die Seite des Autorisierungsserver
3. Einloggen am Autorisierungsserver und der Anwendung den Zugriff erlauben
4. Validierung der Parameter durch Autorisierungsserver
5. Bei Zustimmung antwortet Autorisierungsserver mit JSON Objekt mit Access Token, redirect auf den Client
6. Client sendet Request mit Access Token im Header an Resource Server

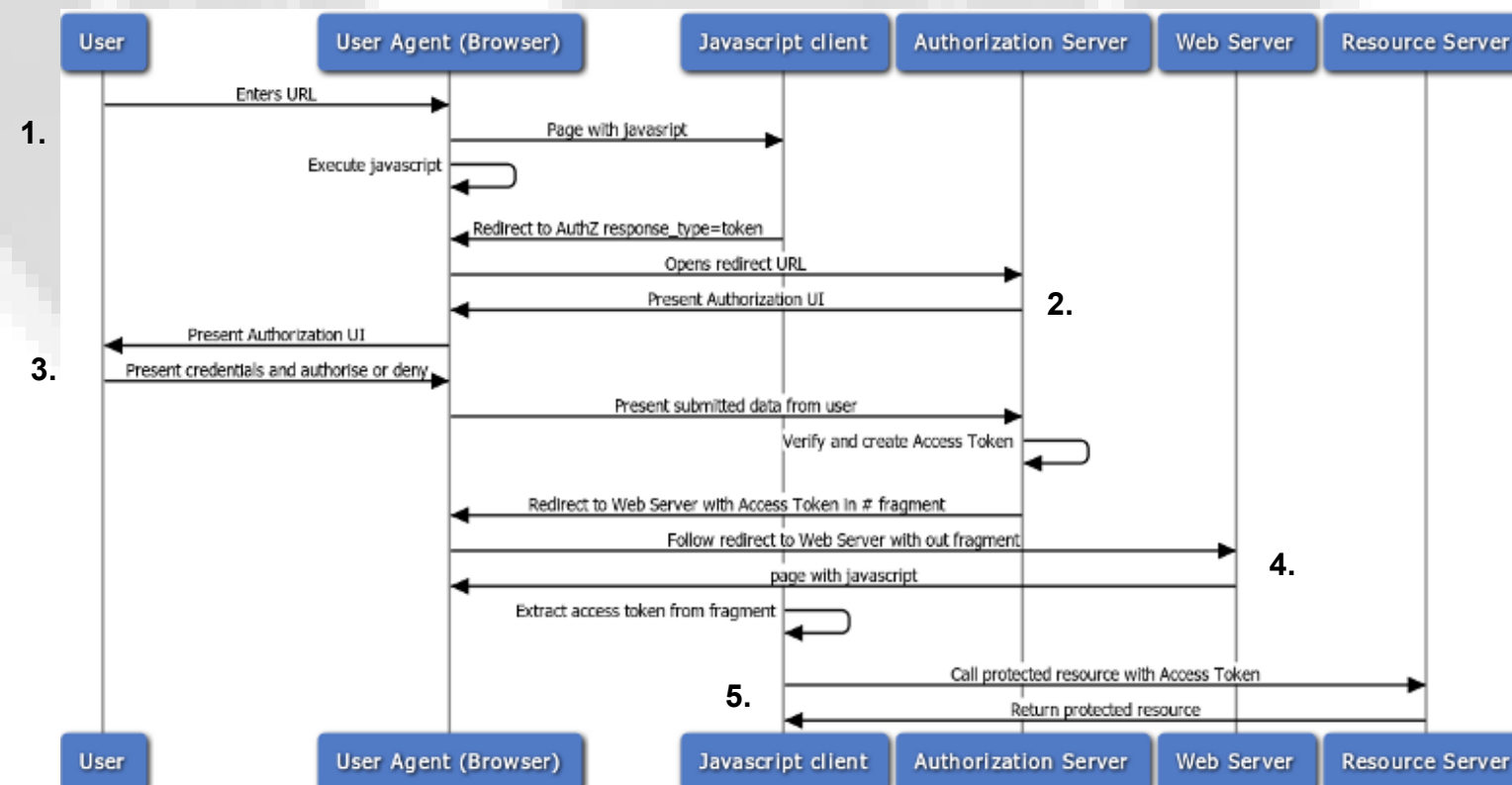


[APIOAuth]

Implicit Code Grant Type

Ablauf:

1. User besucht die Webseite
2. Redirect des Users auf die Seite des Autorisierungsserver, Parametern im Query String
 - **response_type** Wert: **"token"**
 - **client_id** Wert: zugewiesene client_id
 - **redirect_uri** Wert: Client redirect URI (optional), falls nicht, redirect an vorregistrierte URI
 - **scope** Wert: Liste von Scopes
 - **state** Wert: Client token(optional)
- Validierung der Parameter durch Autorisierungsserver, anzeigen der Autorisierungsseite
3. Einloggen des Users am Autorisierungsserver und der Client-Anwendung den Zugriff erlauben
4. Bei Zustimmung, redirect auf den Client mit Parametern im Query String:
 - **token_type** Wert: **"Bearer"**
 - **expires_in** Wert: Integer (TTL Time of Life des Access Tokens)
 - **access_token** Wert: Access Token
 - **state** Wert: State Parameter aus dem ursprünglichen Request
5. Client sendet Request mit Access Token im Header an Resource Server

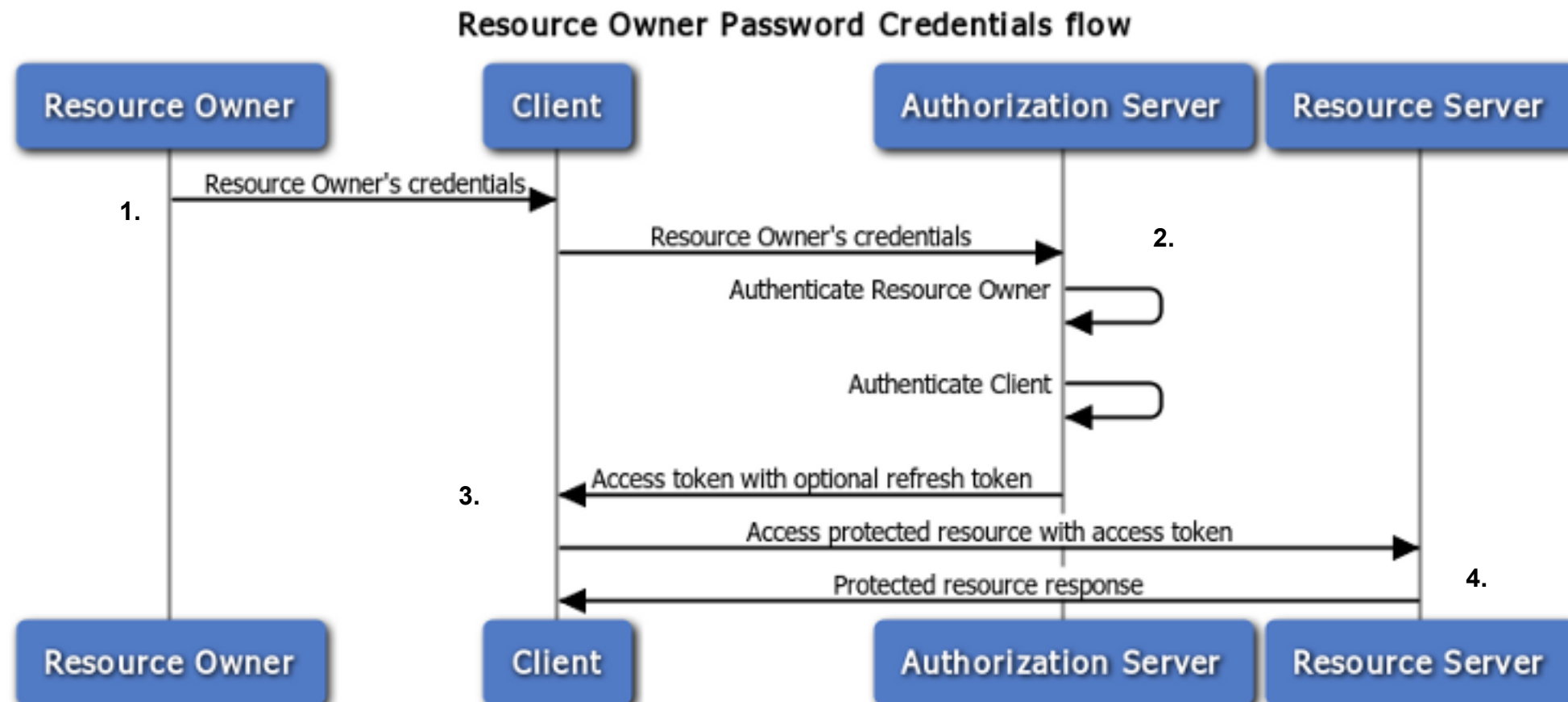


[APIOAuth]

Resource Owner Password Credentials Grant Type

Ablauf:

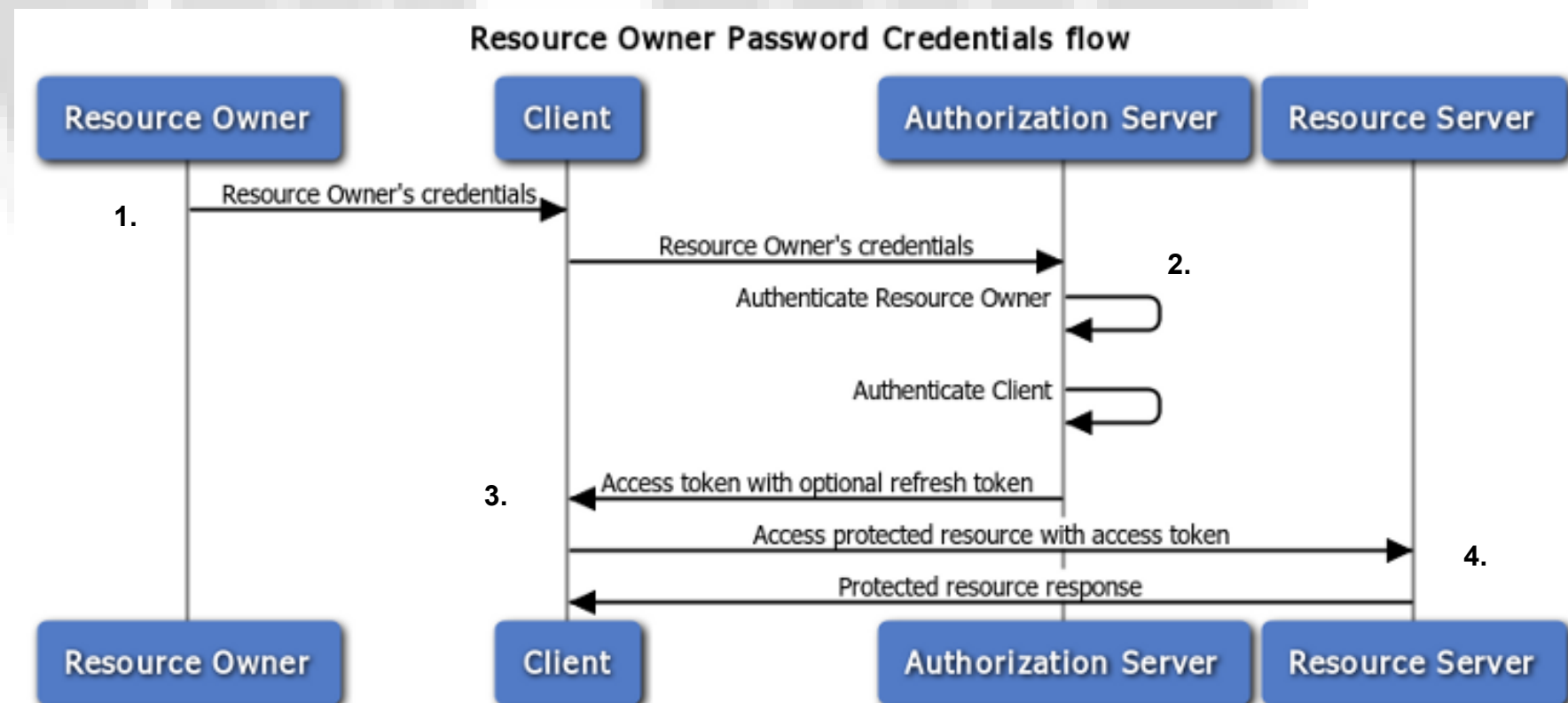
1. Client fordert username/password vom Resource Owner
2. Client sendet einen POST-Request an den Autorisierungsserver
3. Autorisierungsserver antwortet mit JSON Objekt mit Properties
4. Client sendet Request mit Access Token im Header an Resource Server



Resource Owner Password Credentials Grant Type

Ablauf:

1. Client fordert username/password von User
2. Client sendet einen POST-Request an den Autorisierungsserver, Parameter im Query-String
 - **grant_type** Wert: **"password"**
 - **client_id** Wert: Client Id
 - **client_secret** Wert: Client Secret
 - **scope** Wert: Liste von Scopes
 - **username** Wert: Username
 - **password** Wert: Passwort
3. Autorisierungsserver antwortet mit JSON Objekt mit Properties
 - **token_type** Wert: **"Bearer"**
 - **expires_in** Wert: Integer (TTL Time of Life des Access Tokens)
 - **access_token** Wert: Access Token selbst
 - **refresh_token** Wert: Refresh Token, zur Anforderung eines neuen Access Tokens
4. Client sendet Request mit Access Token im Header an Resource Server

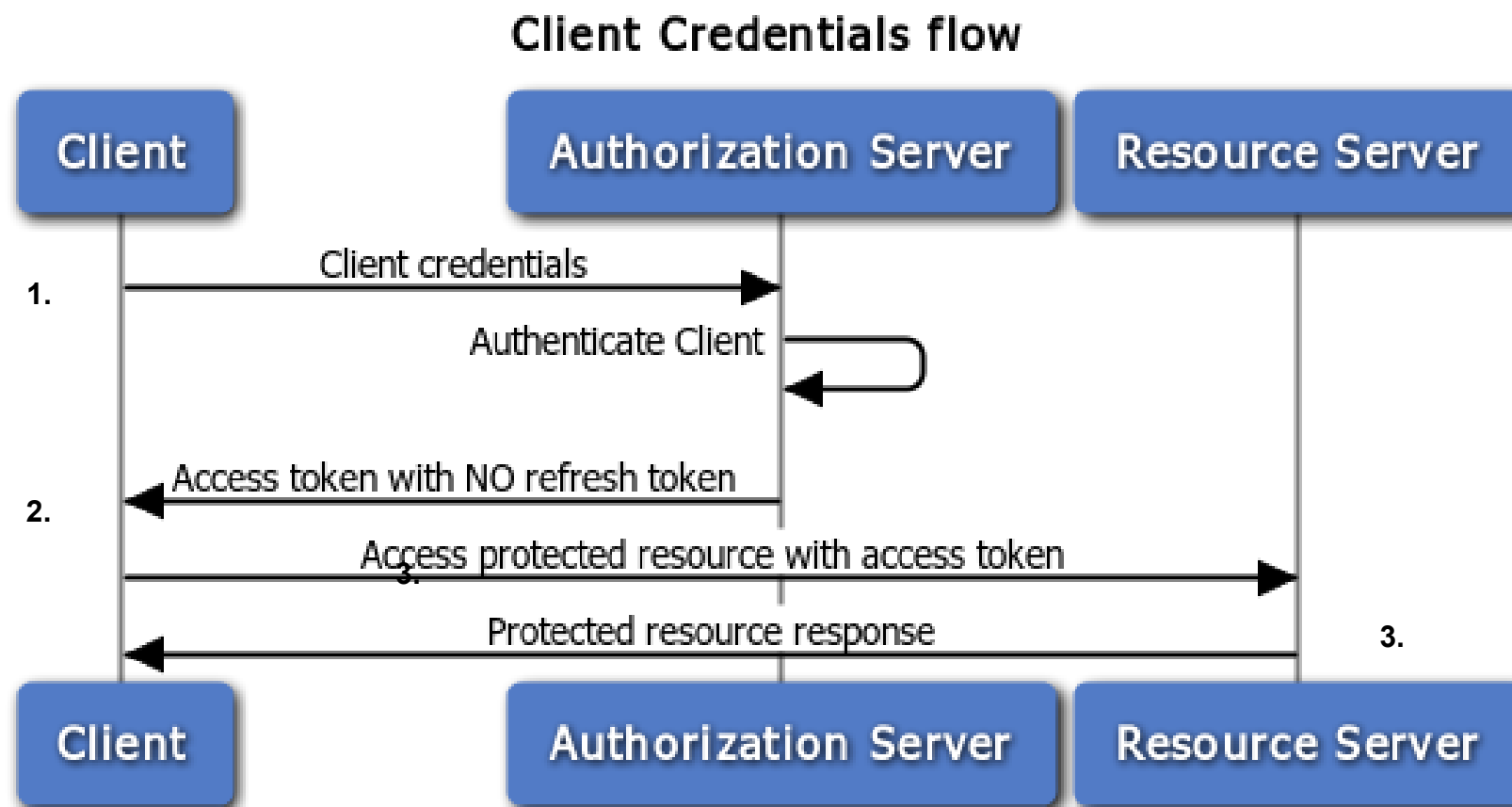


[APIOAuth]

Client Credentials Grant Type

Ablauf:

1. Client sendet einen POST-Request an den Autorisierungsserver
2. Autorisierungsserver antwortet mit Access Token
3. Client sendet Request mit Access Token an Resource Server

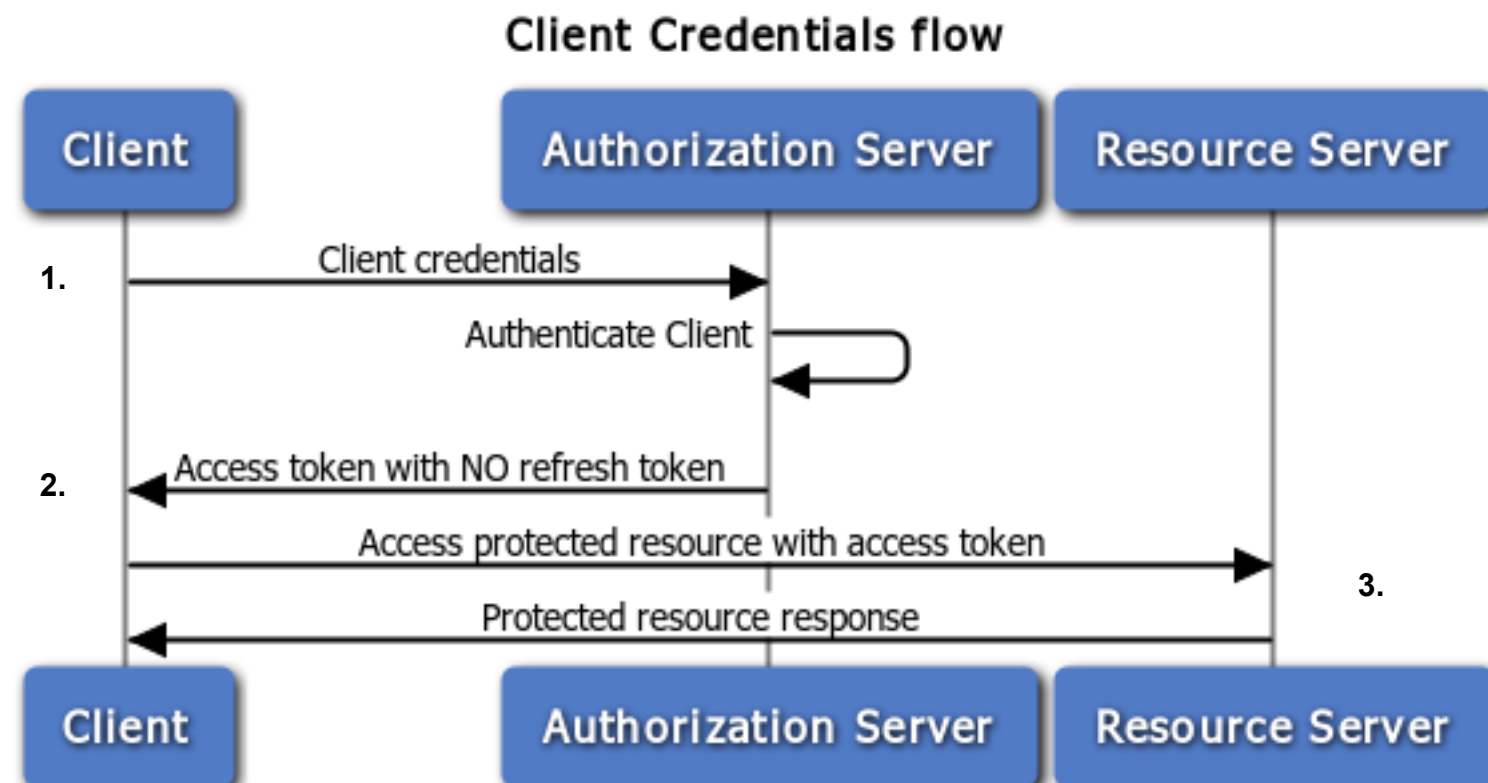


[APIOAuth]

Client Credentials Grant Type

Ablauf:

1. Client sendet einen POST-Request an den Autorisierungsserver mit Parametern im Query-String
 - **grant_type** Wert: “**client_credentials**”
 - **client_id** Wert: Client Id
 - **client_secret** Wert: Client Secret
 - **scope** Wert: optional Liste von Scopes
2. Autorisierungsserver antwortet mit JSON Objekt mit Properties
 - **token_type** Wert: “**Bearer**”
 - **expires_in** Wert: Integer (TTL Time of Life des Access Tokens)
 - **access_token** Wert: Access Token selbst
3. Client sendet Request mit Access Token im Header an Resource Server



[APIOAuth]

OAuth2 Endpoints

OAuth2 Workflow benötigt verschiedene Endpoints (URLs)

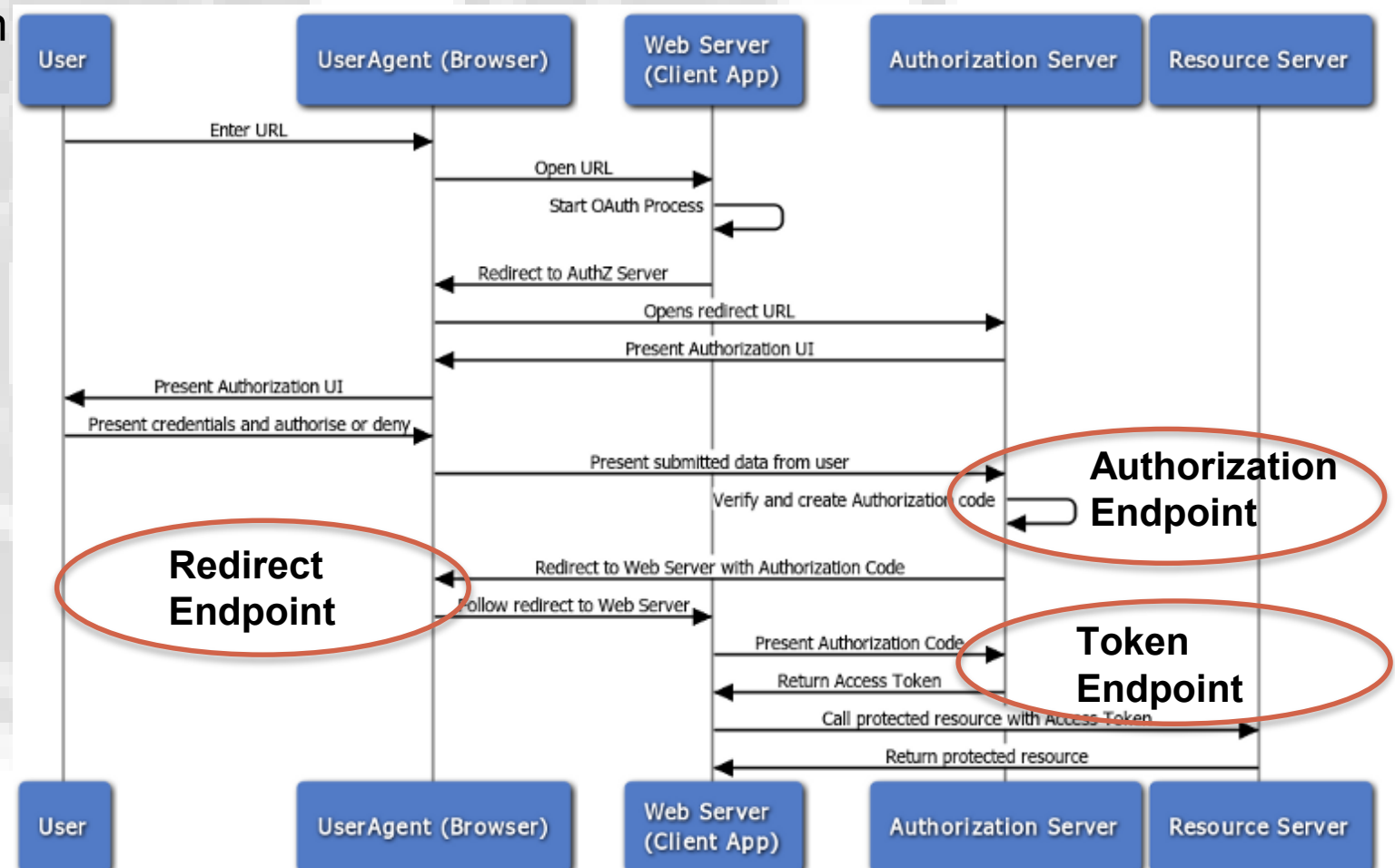
Autorisierungsserver

- Authorization Endpoint
akzeptiert Requests für Autorisierung
default: `https://host:port/oauth/authorize`
- Token Endpoint
akzeptiert Requests für Access Tokens
default: `https://host:port/oauth/token`

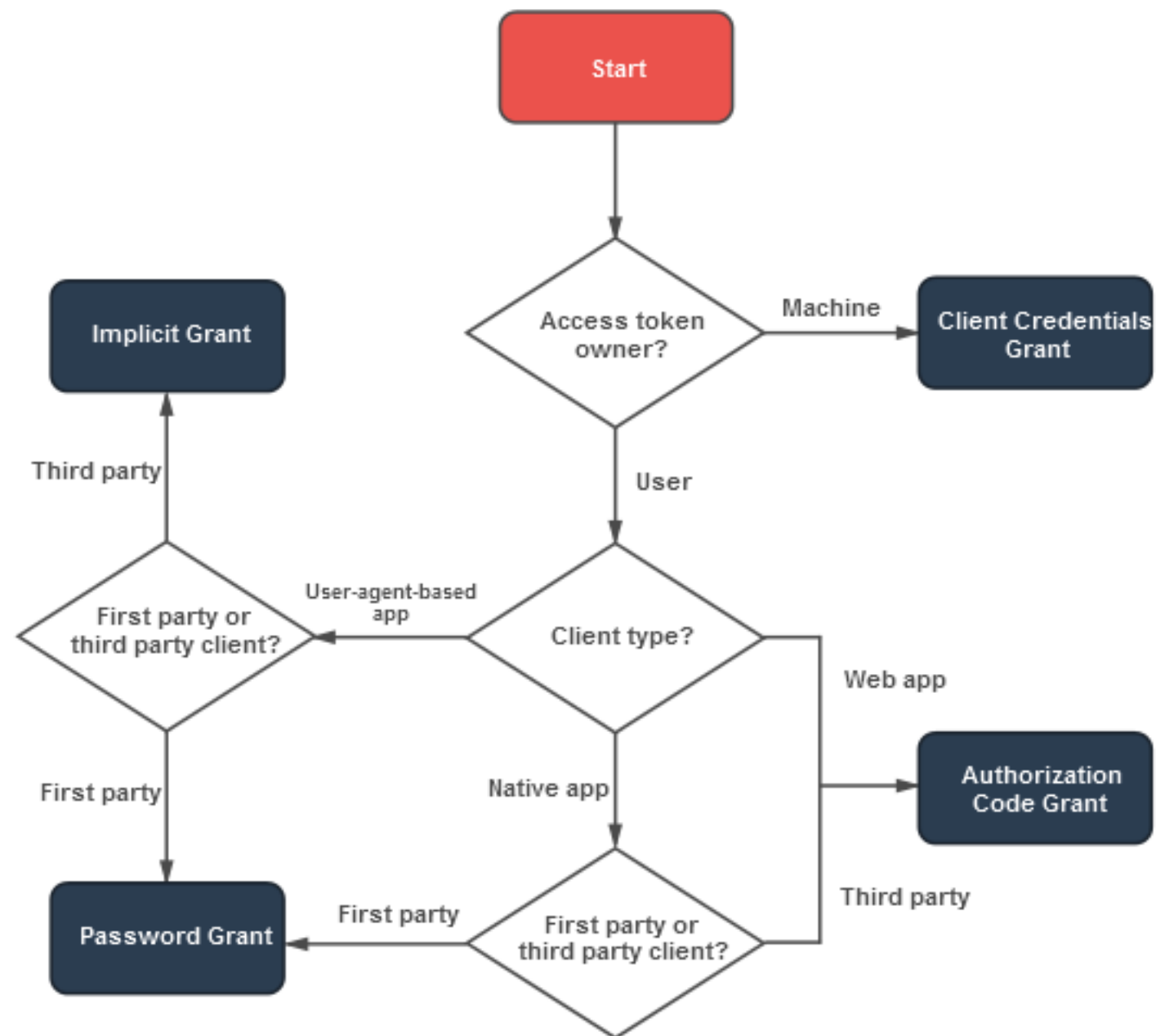
Client Applikation

- Redirect Endpoint

[APIOAuth]



Welcher Grant Type sollte verwendet werden?



[Bilbie 2016]

Spring Security Architektur und OAuth2

Eine Implementierung von OAuth2 → Projekt Spring Security OAuth2

<https://spring.io/projects/spring-security-oauth/>

Spring und Spring Security

Framework stellt Standard Programmiermodelle und Konstrukte zur Verfügung

<https://spring.io/guides/topicals/spring-security-architecture/>

Spring Security Architektur trennt:

- Authentifizierung
- Autorisierung

Spring Security im Classpath → alle Web Anwendungen durch 'Basic' Authentifizierung geschützt

Basic Authentifizierung → mit jedem Request werden Credentials im Authorization Header mitgeschickt

Spring Security Authentifizierung

```
public interface AuthenticationManager {  
    Authentication authenticate(Authentication authentication)  
                           throws AuthenticationException;  
}
```

Returnwert:

- Authentication (normalerweise mit authenticated=true)
- AuthenticationException

Default Implementierung AuthenticationManager → Default user (username : 'user' und random Passwort, ausgegeben auf INFO Level bei Anwendungsstart)

Spring Security Authentifizierung

Default Implementierung

UserDetailsService → Default user (username : 'user' und random Passwort, ausgegeben im Log auf INFO Level bei Anwendungsstart)
InMemoryStore für User

Using generated security password: 78fa095d-3f4c-48b1-ad50-e24c31d5cf35

Default ändern durch Angabe von

```
spring.security.user.name  
spring.security.user.password
```

in application.properties

Abschalten der Default WebApplication Security Konfiguration →

Klasse implementieren, die WebSecurityConfigurerAdapter erweitert

Beispiel UserDetailsService

```
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Bean
    public UserDetailsService users() {
        UserDetails user = User.withDefaultPasswordEncoder()
            .username("user1")
            .password("password")
            .roles("USER")
            .build();
        return new InMemoryUserDetailsManager(user);    }    }
```

```
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {

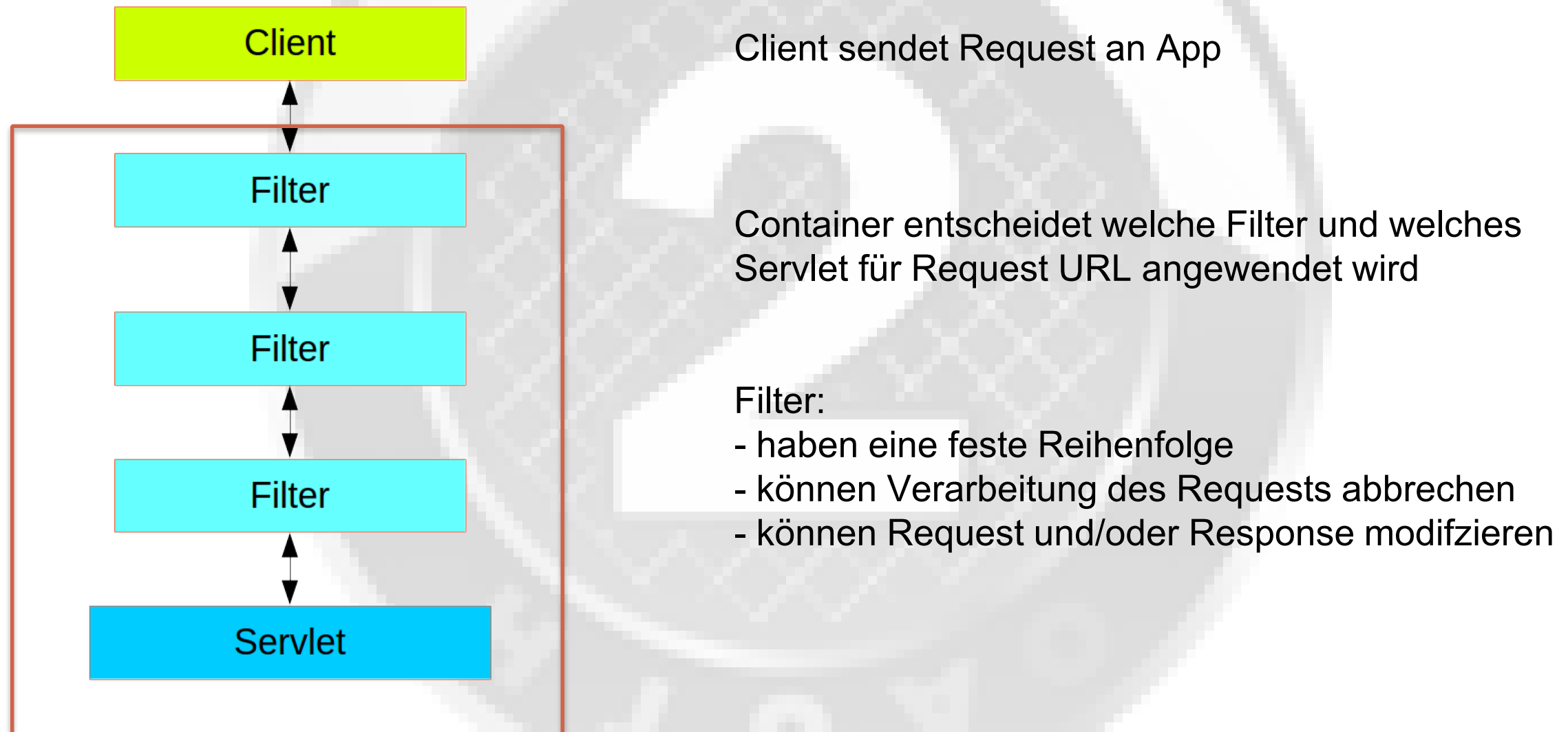
    @Bean
    public UserDetailsService users() throws Exception {
        @SuppressWarnings("deprecation")
        User.UserBuilder users = User.withDefaultPasswordEncoder();
        InMemoryUserDetailsManager manager = new InMemoryUserDetailsManager();
        manager.createUser(users.username("oauthuser").password("oauthpassword")
            .roles("USER").build());
        manager.createUser(users.username("admin").password("password")
            .roles("USER", "ADMIN").build());

        return manager;    }    }
```

InMemoryUserDetailsManager,
JdbcUserDetailsManager,
LdapUserDetailsManager,

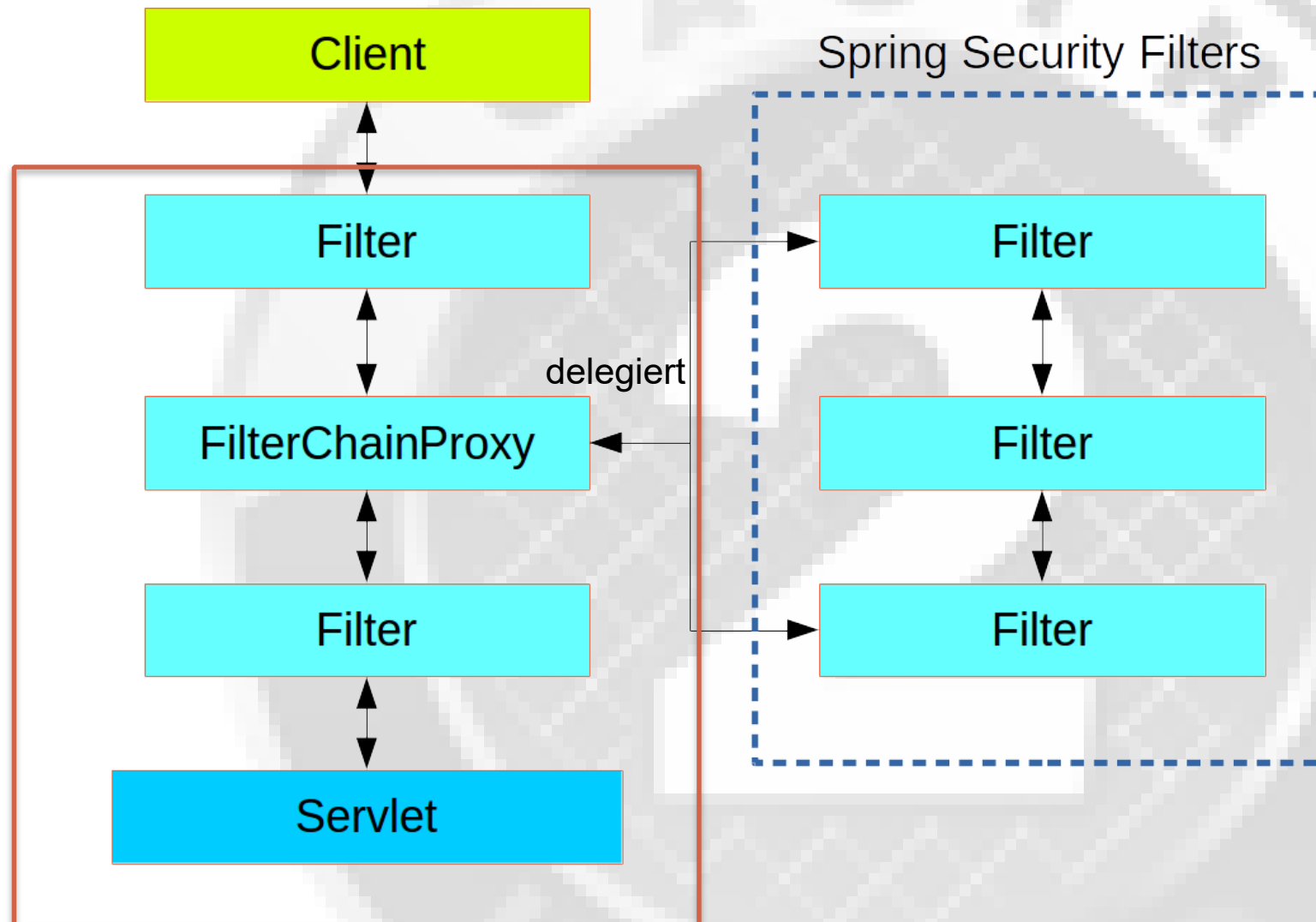
Spring Security Autorisierung

Spring Security für Web Backends basiert auf Servlet Filtern



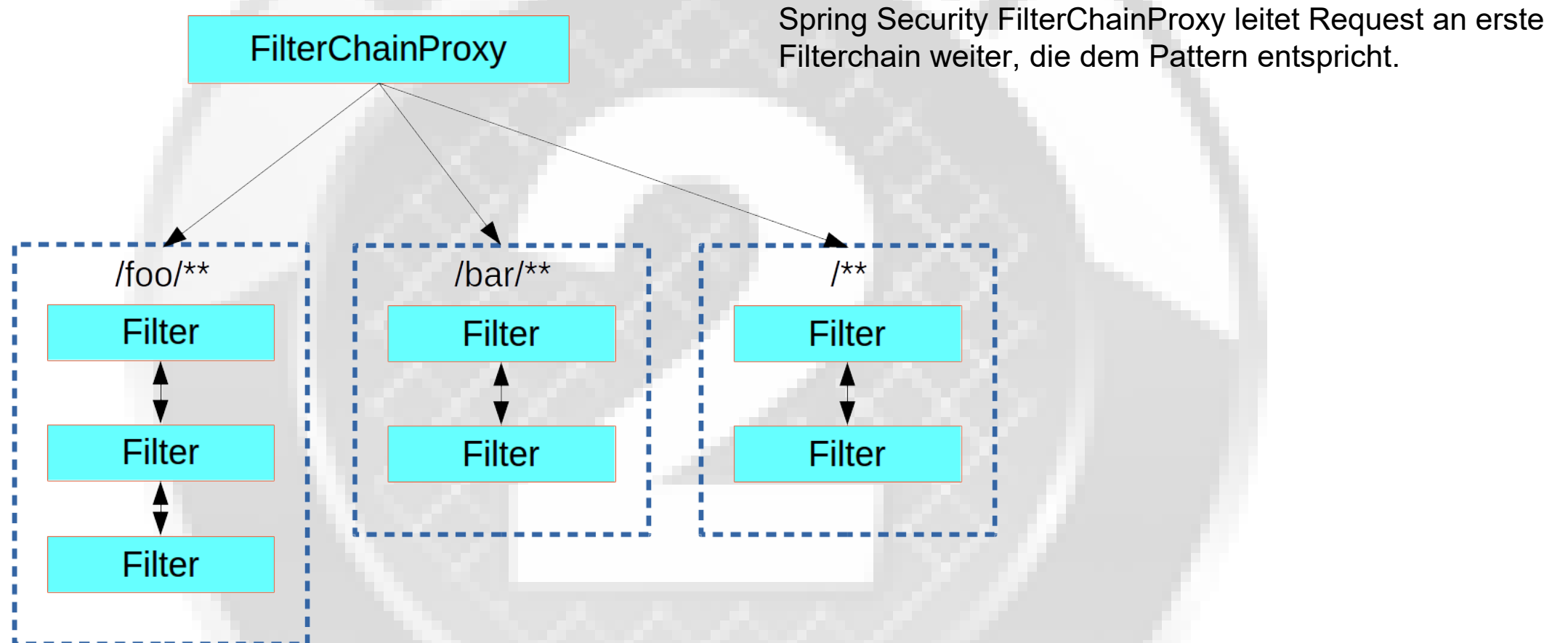
<https://spring.io/guides/topicals/spring-security-architecture/>

Spring Security FilterChainProxy



<https://spring.io/guides/topicals/spring-security-architecture/>

Spring Security FilterChainProxy



<https://spring.io/guides/topicals/spring-security-architecture/>

Spring Security FilterChain Konfiguration

definieren der Regeln

```
@EnableWebSecurity  
@Configuration
```

```
public class SecurityConfiguration extends WebSecurityConfigurerAdapter {  
  
    @Override  
    protected void configure(HttpSecurity http) throws Exception {  
        http.antMatcher("/foo/**")  
            .authorizeRequests()  
                .antMatchers("/foo/bar").hasRole("BAR")  
                .antMatchers("/foo/spam").hasRole("SPAM")  
                .anyRequest().authenticated();  
    }  
}
```

OAuth2 Maven Dependency

```
<dependency>
  <groupId>org.springframework.security.oauth</groupId>
  <artifactId>spring-security-oauth2</artifactId>
  <version>2.4.0.RELEASE</version>
</dependency>

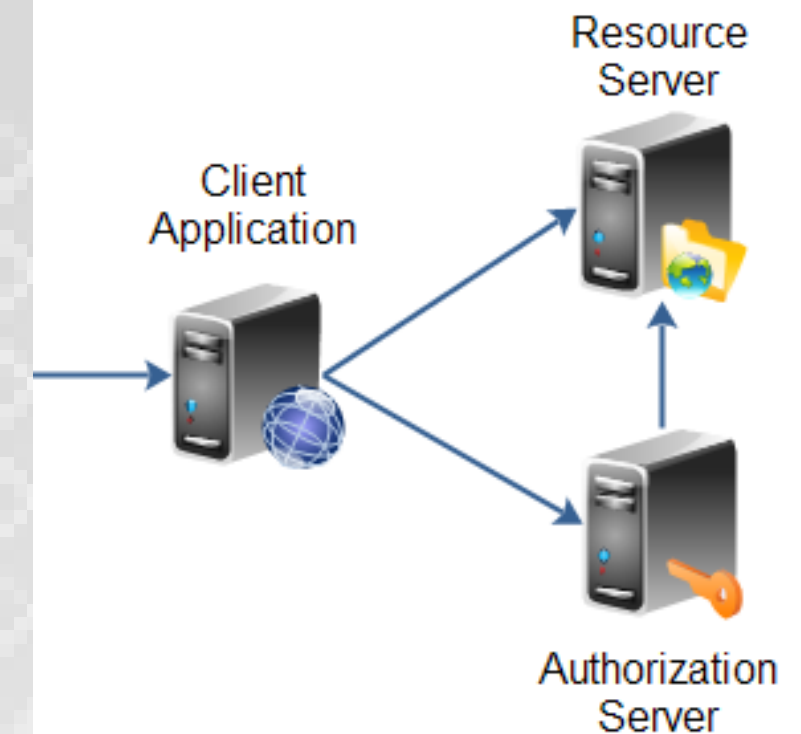
<dependency>
  <groupId>org.springframework.security.oauth.boot</groupId>
  <artifactId>spring-security-oauth2-autoconfigure</artifactId>
  <version>2.2.1.RELEASE</version>
</dependency>
```

spring-security-oauth2 enthält Autokonfigurationen um Autorisierungs- und Resource Server zu implementieren

OAuth2 Komponenten

Implementierung, Konfigurieren der Komponenten

- Authorization Server
 - Client Details Service
 - User Details Service
 - Token Service
- Resource Server
- Client Application



Beispielprojekt <https://iz-gitlab-01.hs-karlsruhe.de/IWI-I/vsmlab-oauth-demo>

Autorisierungsserver

@Configuration

@EnableAuthorizationServer

Stellt Endpoints zur Verfügung

- AuthorizationEndpoint (/oauth/authorize)
- TokenEndpoint (/oauth/token)

Authorizationserver Konfiguration

```
@Configuration
@EnableAuthorizationServer
public class AuthorizationServerConfiguration extends AuthorizationServerConfigurerAdapter
{

    @Override
    public void configure(ClientDetailsServiceConfigurer clients) throws Exception {
        // defines the client details service
        // TODO Auto-generated method stub
        super.configure(clients);
    }

    @Override
    public void configure(AuthorizationServerEndpointsConfigurer endpoints) throws Exception
    {
        // defines the authorization and token endpoints and the token services
        // TODO Auto-generated method stub
        super.configure(endpoints);
    }
}
```

Authorizationserver Konfiguration

Konfiguration der Endpoints und Clients

```
@Configuration
@EnableAuthorizationServer
public class AuthorizationServerConfiguration extends AuthorizationServerConfigurerAdapter {

    @Override
    public void configure(ClientDetailsServiceConfigurer clients) throws Exception {
        clients.inMemory()
            .withClient("messaging-client").secret("{noop}secret")
            .authorizedGrantTypes("authorization_code", "client_credentials",
                                "password", "refresh_token")
            .redirectUri("http://localhost:8080/client/authorized")
            .scopes("message.read", "message.write");
    }

    @Override
    public void configure(AuthorizationServerEndpointsConfigurer endpoints) throws Exception {
        endpoints // injects the Spring Security authentication manager (set up in WebSecurityConfiguration )
            .authenticationManager(authenticationManager)
            .tokenStore(tokenStore())
            .userApprovalHandler(superApprovalHandler())
            .accessTokenConverter(accessTokenConverter());
    }
}
```

Authorizationserver Konfiguration

- **ClientDetailsServiceConfigurer:** definiert den Client Details Service in-memory oder JDBC Implementierung der Clientdetails wichtige Attribute eines Clients:
 - clientId: (required)
 - clientsecret: (required für trusted Clients)
 - scope: Scopes (Zugriffsrechte) auf die Client beschränkt ist
 - authorizedGrantTypes: Grant types, die Client benutzen darf

- **AuthorizationServerEndpointsConfigurer:** definiert non-security Features der Endpoints
 - AuthenticationManager
 - Token Store
 - UserApprovalHandler
 - AccessTokenConverter

Authorizationserver Security Konfiguration

@EnableWebSecurity

```
public class WebSecurityConfiguration extends WebSecurityConfigurerAdapter {
```

@Override

```
protected void configure(HttpSecurity http) throws Exception {
```

```
    http
```

```
        .authorizeRequests()
```

```
        .antMatchers("/oauth2/keys").permitAll()
```

```
        .anyRequest().authenticated()
```

```
        .and()
```

```
        .formLogin();    }
```

@Bean

```
public UserDetailsService users() {
```

```
    UserBuilder users = User.withDefaultPasswordEncoder();
```

```
    InMemoryUserDetailsManager manager = new InMemoryUserDetailsManager();
```

```
                                // JdbcUserDetailsManager, LdapUserDetailsManager
```

```
    manager.createUser(users.username("user").password("password")
```

```
                        .roles("USER")
```

```
                        .build());
```

```
    manager.createUser(users.username("admin").password("password")
```

```
                        .roles("USER", "ADMIN")
```

```
                        .build());
```

```
    return manager;    }
```

@Bean

@Override

```
public AuthenticationManager authenticationManagerBean() throws Exception {
```

```
    return super.authenticationManagerBean(); }
```

Authorizationserver Token Konfiguration

Autorisierungsserver erzeugt Token → Resourceserver benutzt Token

Daten der Authentifizierung müssen beim Anlegen des Token gespeichert werden:

- InMemoryTokenStore
- JdbcTokenStore
- JSON Web Token

Verschiedene Arten:

Self-contained Access Token

enthält Information über Resource Owner, Client, Rechte

Random Access Token

enthält keine Information über Resource Owner, Client

JSON Web Token

besteht aus Header, Payload, Signature

Payload enthält Infos zu Client, Rechten

Authorizationserver Token Konfiguration

```
<dependency>
  <groupId>org.springframework.security</groupId>
  <artifactId>spring-security-jwt</artifactId>
  <version>1.1.0.RELEASE</version>
</dependency>
<dependency>
  <groupId>com.nimbusds</groupId>
  <artifactId>nimbus-jose-jwt</artifactId>
  <version>8.2.1</version>
</dependency>
```

Verifikation des JWT :

- Schnittstelle zum Abrufen des Keys
- Keys müssen für alle anderen Services bekannt sein

Authorizationserver Token Konfiguration

Im Beispiel Projekt wird Schnittstelle definiert:

```
@RestController
public class JwkController {

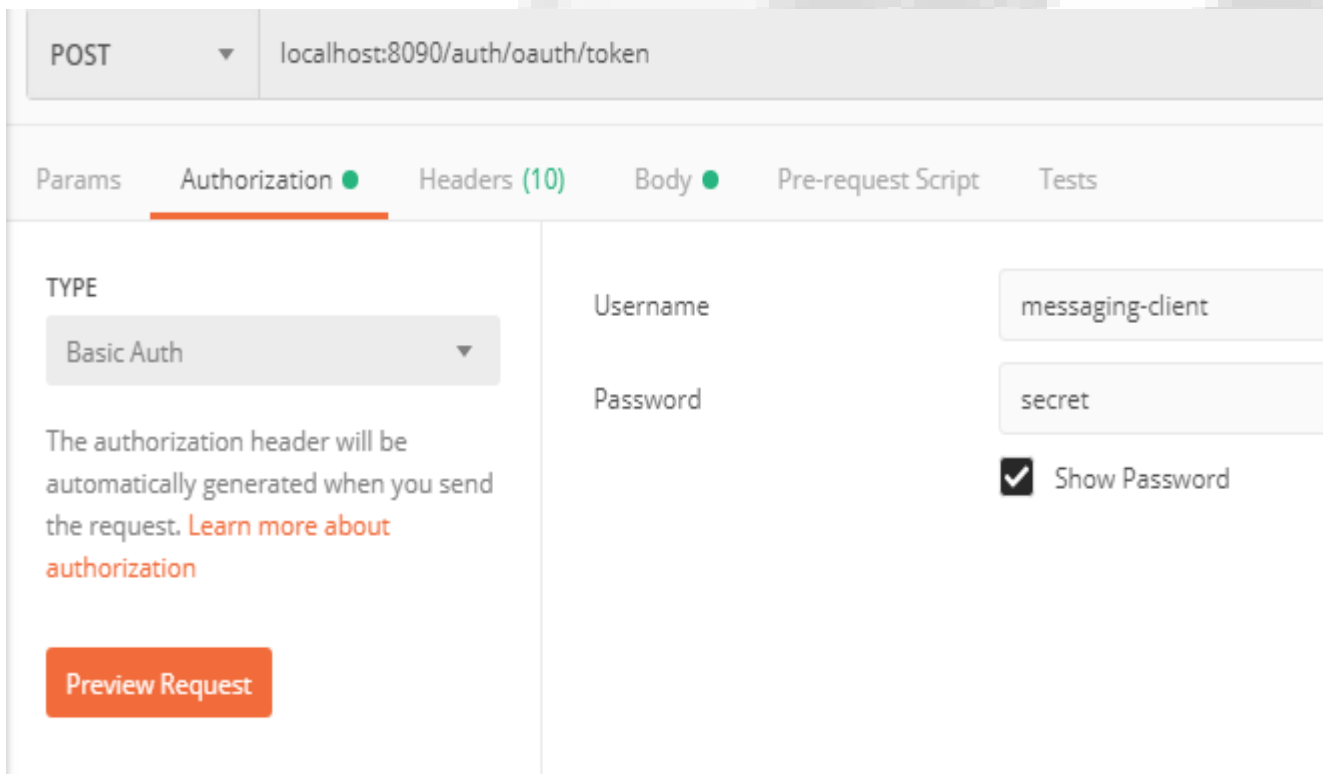
    // ..

    @GetMapping(value = "/oauth2/keys", produces = "application/json; charset=UTF-8")
    public String keys() {
        return this.jwkSet.toString();
    }
}
```


Authorizationserver testen

Mit curl, Postman oder ähnlichen Tools

```
$ curl messaging-client:secret@localhost:8090/auth/oauth/token
-d grant_type=password
-d username=oauthuser
-d password=oauthpassword
-d scope=message.read
```



POST localhost:8090/auth/oauth/token

Params Authorization Headers (10) Body Pre-request Script Tests

TYPE: Basic Auth

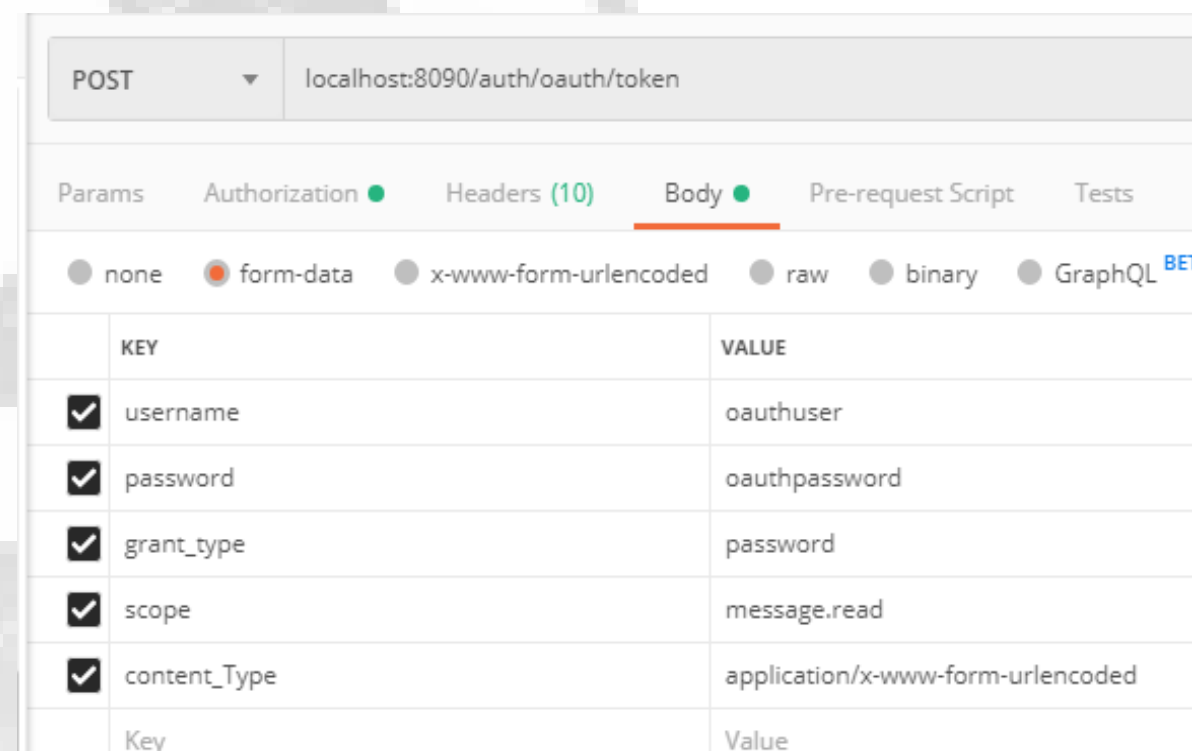
The authorization header will be automatically generated when you send the request. [Learn more about authorization](#)

Preview Request

Username: messaging-client

Password: secret

☒ Show Password



POST localhost:8090/auth/oauth/token

Params Authorization Headers (10) Body Pre-request Script Tests

☐ none ☒ form-data ☐ x-www-form-urlencoded ☐ raw ☐ binary ☐ GraphQL

KEY	VALUE
<input checked="" type="checkbox"/> username	oauthuser
<input checked="" type="checkbox"/> password	oauthpassword
<input checked="" type="checkbox"/> grant_type	password
<input checked="" type="checkbox"/> scope	message.read
<input checked="" type="checkbox"/> content_Type	application/x-www-form-urlencoded

Key Value

basic auth credentials für /token Endpoint: clientId und clientsecret (messaging-client/secret)
User credentials: falls default Spring Security User Details ("user", random password) oder
in UserDetailsService in Authorisierungsserver konfiguriert

Resource Server Konfiguration

Resource Server stellt Ressourcen zur Verfügung, die durch OAuth2 Tokens geschützt sind

@EnableResourceServer

@EnableResourceServer Konfiguration OAuth2 Resource Server, aktiviert Spring Security Filter, ankommende Requests werden über OAuth2 Token authentifiziert, d.h. bei jedem Zugriff auf einen Resource Server Endpoint verifiziert Spring das Access Token

application.properties

```
security.oauth2.resource.jwk.key-set-uri=http://localhost:8090/auth/oauth2/keys
```

Security auf Methodenebene hinzufügen durch Annotation:

@EnableGlobalMethodSecurity

Methode annotieren mit z.B.:

```
@PreAuthorize("hasRole('ROLE_USER')")
```

OAuth2 Client Application Konfiguration

@EnableOAuth2Client

Service wird OAuth2 Client, weiterleiten der Access Token bei Anfragen an Resource Server durch OAuth2RestTemplate

- OAuth2ClientContext Objekt
- OAuth2AccessToken Objekt
- AccessTokenRequest Objekt

OAuth2RestTemplate kann OAuth2-authentifizierte REST Requests durchführen

OAuth2RestTemplate(OAuth2ProtectedResourceDetails resource, OAuth2ClientContext context)

OAuth2ProtectedResourceDetails

Interface für die Granttypes

- AuthorizationCodeResourceDetails
- ClientCredentialsResourceDetails
- ImplicitResourceDetails
- ResourceOwnerPasswordResourceDetails

<https://docs.spring.io/spring-security/oauth/apidocs/org/springframework/security/oauth2/client/OAuth2RestTemplate.html>

Implementierung Resource Owner Password Credentials

In Client Application

```
ResourceOwnerPasswordResourceDetails resourceDetails =  
    new ResourceOwnerPasswordResourceDetails();  
resourceDetails.setUsername("oauthuser");  
resourceDetails.setPassword("oauthpassword");  
resourceDetails.setAccessTokenUrl(" http://localhost:8090/auth/oauth/token");  
resourceDetails.setClientId("messaging-client");  
resourceDetails.setClientSecret("secret");  
resourceDetails.setGrantType("password");  
resourceDetails.setScope(asList("message.read", "message.write"));  
  
OAuth2RestTemplate oauth2RestTemplate =  
    new OAuth2RestTemplate(resourceDetails, clientContext);  
  
String[] messages = this.  
    oauth2RestTemplate.getForObject("http://localhost:8092/resource/messages",  
        String[].class);
```

<http://docs.spring.io/spring-security/oauth/apidocs/org/springframework/security/oauth2/client/OAuth2RestTemplate.html>

Application.properties

OAuth-Server

application.properties

server.port=8090

server.servlet.context-path=/auth

Client Application

application.properties

server.port=8080

security.oauth2.client.client-id=messaging-client

security.oauth2.client.client-secret=secret

security.oauth2.client.access-token-uri=http://localhost:8090/auth/oauth/token

security.oauth2.client.grant-type=authorization_code

security.oauth2.client.user-authorization-uri=http://localhost:8090/auth/oauth/authorize

Resource-Server

application.properties

server.port=8092

security.oauth2.resource.jwk.key-set-uri=http://localhost:8090/auth/oauth2/keys

Aufgabe 4 (10.06.20 – 01.07.20) - Web Shop Implementierung

In der vierten und letzten Aufgabe soll die Struts-Anwendung des **Web Shops als REST-Client** an den in Aufgabe 3 erstellten Edge Server angebunden werden. Dabei soll eine **Autorisierung der Web Shop Anwendung mittels OAuth** vorgenommen werden. Konkret soll eine Lösung auf Basis von OAuth 2.0 realisiert werden.

Aufgabe 4.1 Implementieren Sie einen Authorisierungsserver mit OAuth2 und nutzen z.B. den ProductService als Resourceserver. Testen Sie den Ablauf, anfordern eines Tokens und auslesen der Resource mit Hilfe von curl oder Postman.

Binden Sie den Edgeserver ein, d.h. alle Anfragen gehen über den Edge-Server.

Aufgabe 4.2 Der Login soll so angepasst werden, dass damit eine Autorisierung der Web Anwendung gegenüber der Microservice Infrastruktur erfolgt.

Schreiben Sie die LoginAction des Webshops so um, dass der Authorisierungsserver bzw. Userservice mittels eines Restaufrufs benutzt werden kann und testen Sie den Zugriff.

Aufgabe 4.3 Passen Sie die Restaufrufe im Webshop an die Anforderungen für den Aufruf mit OAuthRestTemplate an.

Aufgabe 4.4 Als letzter Teil ist noch das Registrieren umzusetzen.

Links

Titelbild	http://www.thinkstockphotos.de
[Spring 2016a]	Tutorial Spring Boot and OAuth2 https://spring.io/guides/tutorials/spring-boot-oauth2/
[Spring 2016b]	OAuth 2 Developers Guide https://projects.spring.io/spring-security-oauth/docs/oauth2.html
[Spring 2017a]	Spring Security Architecture https://spring.io/guides/topicals/spring-security-architecture/
[Spring 2017b]	Securing a Web Application https://spring.io/guides/gs/securing-web/
[Spring 2019]	OAuth2 Migration Guide https://github.com/spring-projects/spring-security/wiki/OAuth-2.0-Migration-Guide
[Spring]	OAuth2 Boot version 2.2.2.RELEASE https://docs.spring.io/spring-security-oauth2-boot/docs/current/reference/html5/
[APIOAuth]	“API Gateway OAuth 2.0 Authentication Flows” https://docs.oracle.com/cd/E39820_01/doc.11121/gateway_docs/content/oauth_flows.html
[Bergamo 2018]	“Building Scalable Container-Ready and Secure Microservices Using Spring Boot, Netflix OSS, and Spring OAuth2” https://dzone.com/articles/building-scalable-container-ready-and-secure-micro
[Bilbie 2016]	Alex Bilbie “A Guide To OAuth 2.0 Grants” https://alexbilbie.com/guide-to-oauth-2-grants/
[Dhiraj 2018]	Dhiraj “Spring Boot Security OAuth2 Example” https://www.devglan.com/spring-security/spring-boot-security-oauth2-example
[Doerrfeld 2015]	Bill Doerrfeld “How To Control User Identity Within Microservices” http://nordicapis.com/how-to-control-user-identity-within-microservices/
[Fast]	Jürgen Fast “Tutorial Spring Security” https://labs.micromata.de/tutorial-spring-security/
[Gupta]	Lokesh Gupta “Spring Boot 2 – OAuth2 Auth and Resource Server “ https://howtodoinjava.com/spring-boot2/oauth2-auth-server/
[Jenkov 2014]	Jakob Jenkov “OAuth 2.0 Tutorial” http://tutorials.jenkov.com/oauth2/authorization.html

Links

- [Kürsten 2019] Philipp Kürsten "Rest-Schnittstellen absichern mit Java Spring, OAuth2.0 & JSON Web Token"
<https://blog.ordix.de/technologien/oauth-2-0-und-java-spring-rest-schnittstellen-absichern-mit-spring-oauth-2-0-json-web-token>
- [Larsson 2015] Magnus Larsson "Building Microservices, part3. Secure API's with OAuth 2.0"
<http://callistaenterprise.se/blogg/teknik/2015/04/27/building-microservices-part-3-secure-APIs-with-OAuth/>
- [Lea 2015] Graham Lea "Microservices Security: All The Questions You Should Be Asking"
<http://www.grahamlea.com/2015/07/microservices-security-questions/>
- [Nordström 2016] Jesper Nordström "Architecting for speed: How agile innovators accelerate growth through microservices"
<https://www.linkedin.com/pulse/architecting-speed-how-agile-innovators-accelerate-growth-nordström/>
- [Paraschiv 2018] Eugen Paraschiv „Security with Spring“
<http://www.baeldung.com/security-spring>
- [Parecki a] Aaron Parecki "OAuth 2.0"
<https://oauth.net/>
- [Parecki b] Aaron Parecki "OAuth 2.0 Servers"
<https://www.oauth.com/>
- [Reinke 2016] Johann Reinke, "Understanding OAuth2"
<http://www.bubblecode.net/en/2016/01/22/understanding-oauth2/>
- [Siriwardena 2016] Prabath Siriwardena "Securing Microservices (Part I)"
<https://medium.facilelogin.com/securing-microservices-with-oauth-2-0-jwt-and-xacml-d03770a9a838#.12cwu8zcl>
- [Spyna 2018] Lorenzo Spyna, "An OAuth 2.0 introduction for beginners"
<https://itnext.io/an-oauth-2-0-introduction-for-beginners-6e386b19f7a9>
- [Yankelevich 2016] Federico Yankelevich, "OAuth2 in depth: A step-by-step introduction for enterprises"
<http://www.swisspush.org/security/2016/10/17/oauth2-in-depth-introduction-for-enterprises>
- [Yegulap 2015] Serdar Yegulalp "Microservices: Simple servers, complex security"
www.infoworld.com/article/2984867/application-architecture/microservices-simple-servers-complex-security.html
- [Websystique 2016] "Secure Spring REST API using OAuth2"
<http://websystique.com/spring-security/secure-spring-rest-api-using-oauth2/>
- Demoprojekt Source: <https://iz-gitlab-01.hs-karlsruhe.de/IWI-I/vsmlab-oauth-demo>