



Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

Spring Security

Peter Andres, ISC-EJPD

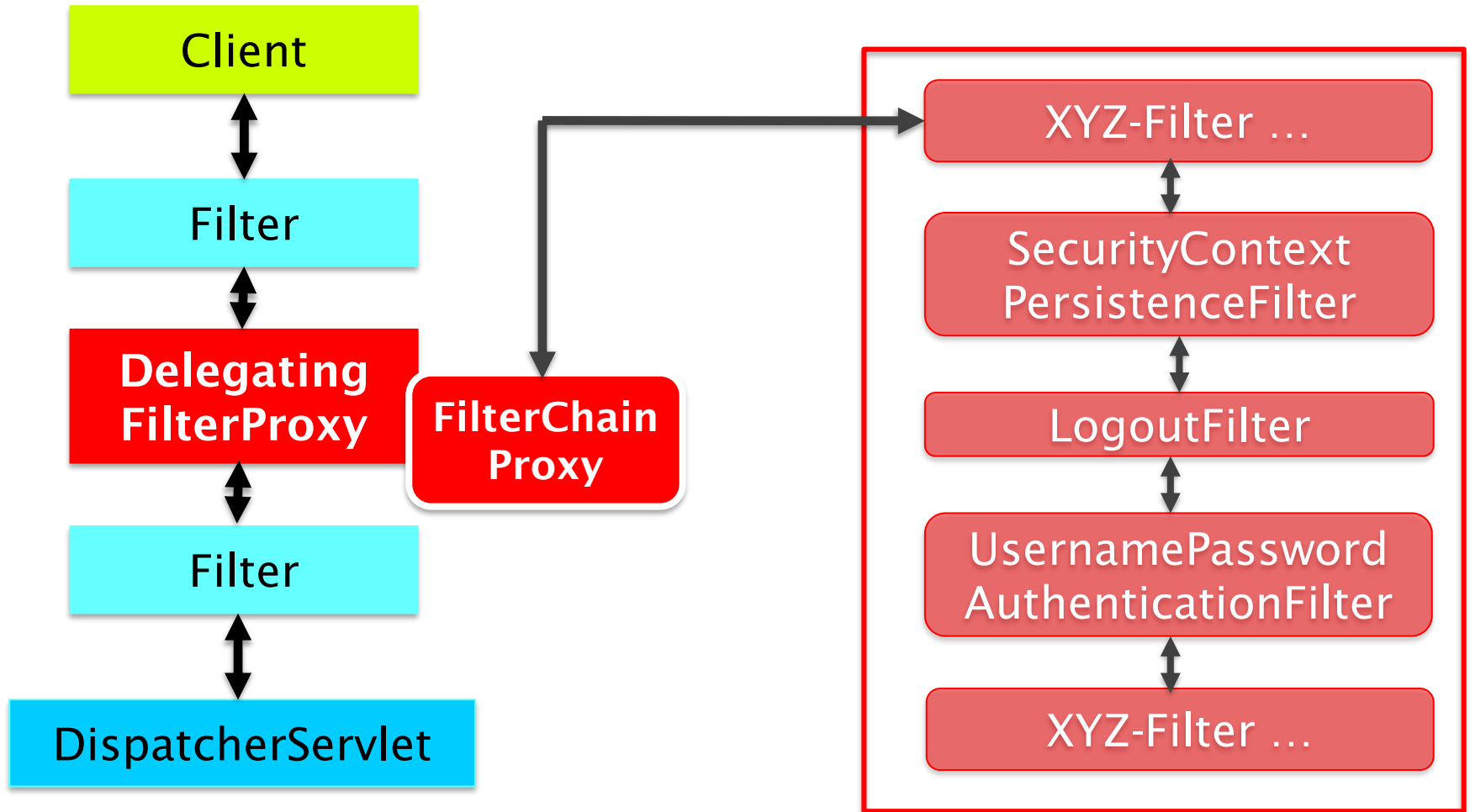


Architektur

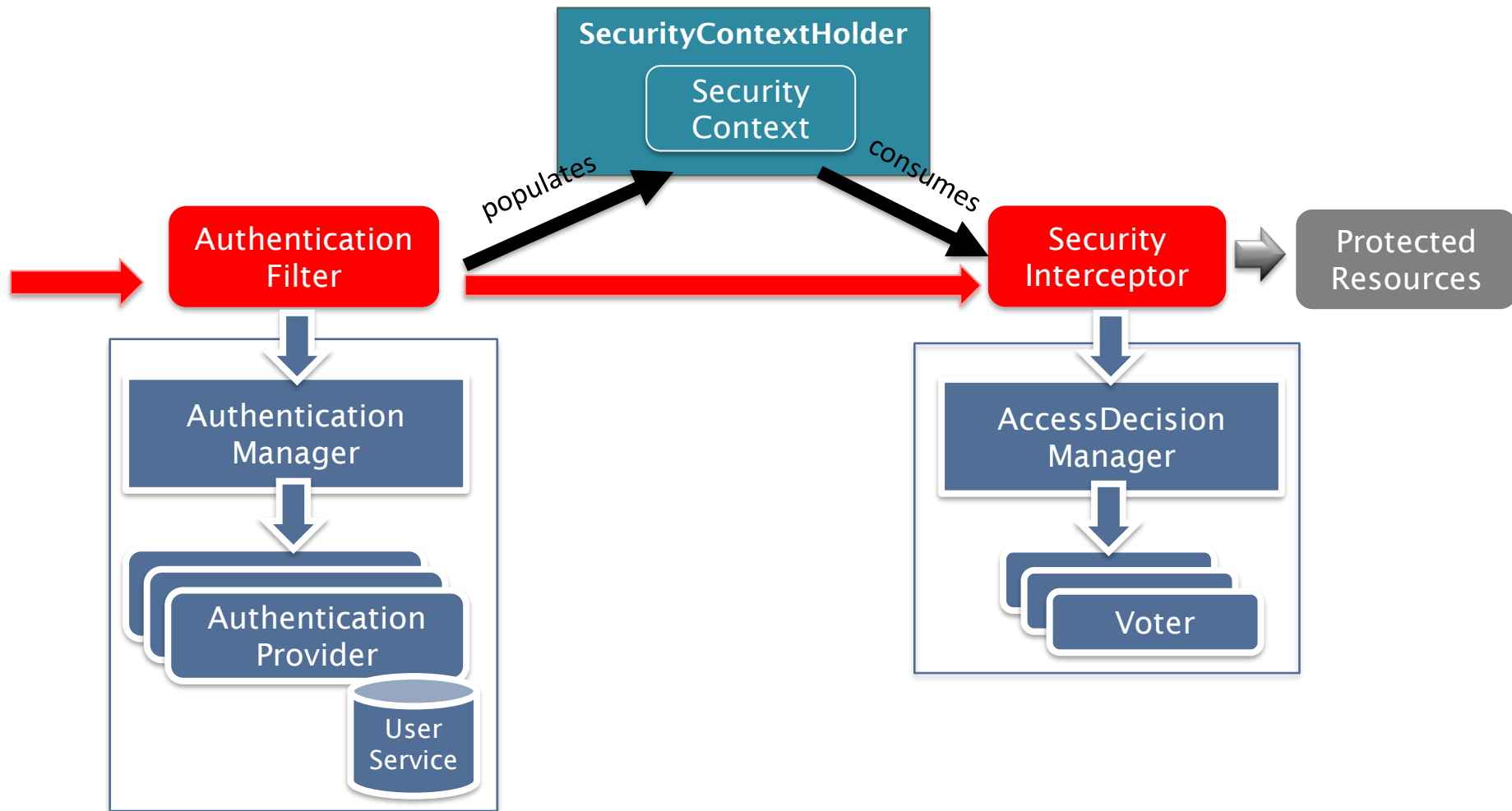
Einführung

- ▶ Spring Security wurde als Teilprojekt von Spring 2003 von Ben Alex ins Leben gerufen und 2008 als Spring Security 2.0 released
- ▶ Spring Security ist ein umfassendes, flexibles und mächtiges Framework und basiert auf der Servlet-Spezifikation
- ▶ Spring unterstützt eine grosse Anzahl von Authentifizierungsprotokollen wie Basic Authentication, LDAP, OAuth2 und OpenID Connect
- ▶ Spring implementiert verschiedene Autorisierungsverfahren:
 - ▶ URL-basierter Zugriffsschutz (für Webanwendungen)
 - ▶ Methoden-basierter Zugriffsschutz
 - ▶ Access Control-Listen (ACL) für einen feingranularen Zugriffsschutz auf Ressourcen oder Objekte

Security-Filterkette



Spring Security-Architektur



Security-Kontext



```
SecurityContext context = SecurityContextHolder.getContext();
Authentication authentication = context.getAuthentication();

// get principal username
User user = (User) authentication.getPrincipal();
String name = user.getUserName();
```

Konfiguration

Maven-Konfiguration

- ▶ Der Security Starter aktiviert die `SecurityAutoConfiguration`-Klasse mit der Default Security-Konfiguration:
 - ▶ sichert alle Endpunkte der Applikation, sodass nur authentifizierte Requests zugelassen sind
 - ▶ generiert einen (in-memory) User mit Name *user* und einem Random-Passwort (Ausgabe auf der Konsole)
 - ▶ setzt die wichtigsten Security-HTTP-Headers
 - ▶ aktiviert CSRF-Tokens

```
<dependency>  
  <groupId>org.springframework.boot</groupId>  
  <artifactId>spring-boot-starter-security</artifactId>  
</dependency>
```


Default Security-Konfiguration

- ▶ Die Default Security-Konfiguration verwendet folgende Application Properties, welche angepasst werden können:

```
spring.security.user.name=user      # default user name  
spring.security.user.password=     # password of the default user  
spring.security.user.roles=        # granted roles of the default user
```

Custom Security-Konfiguration

```
@Configuration
public class SecurityConfig {

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        // configure authentication method and URL authorization
    }

    @Bean
    public UserDetailsService userService() {
        // create user details service
    }

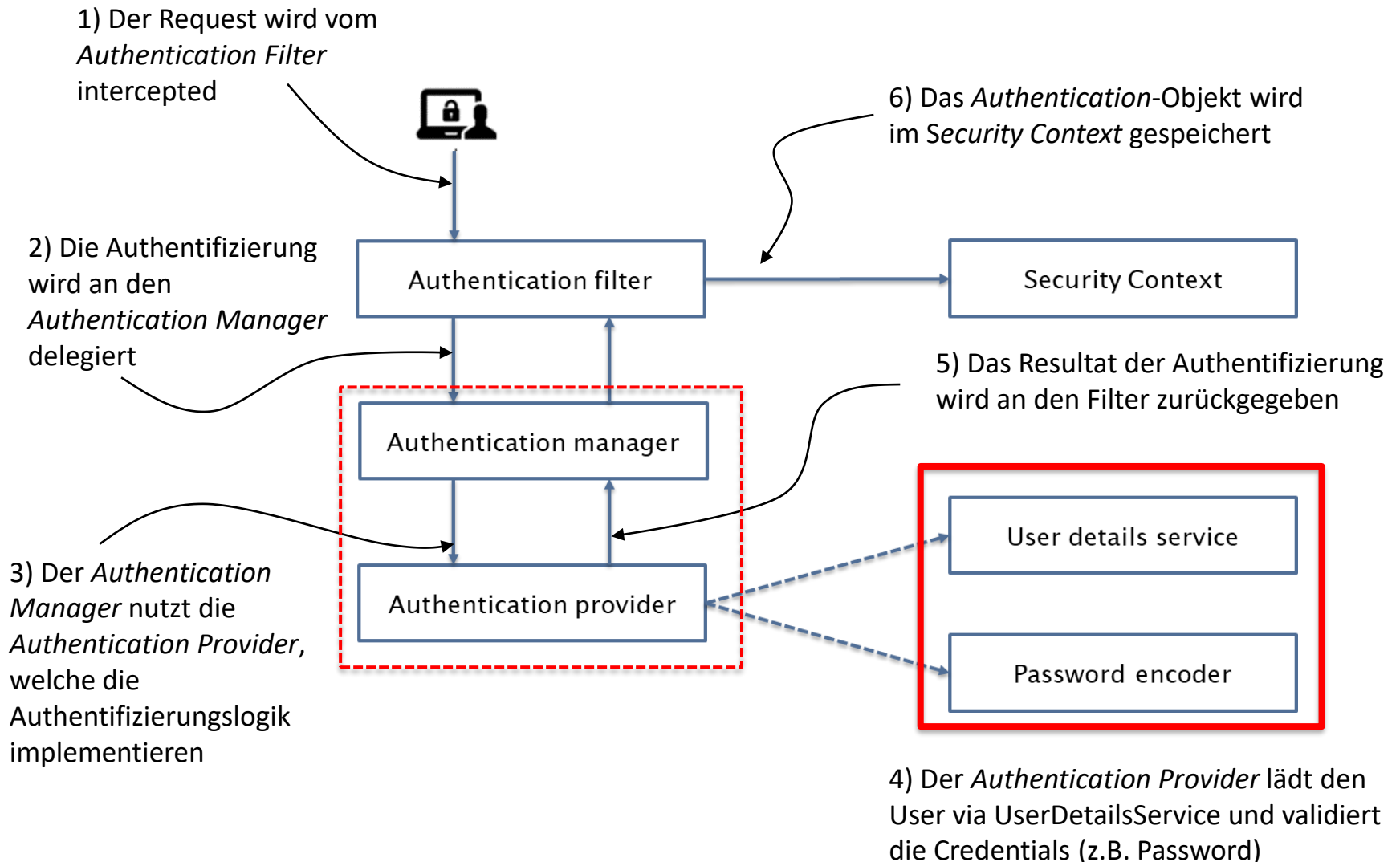
    @Bean
    public PasswordEncoder passwordEncoder() {
        // create password encoder
    }
}
```

Authentifizierung

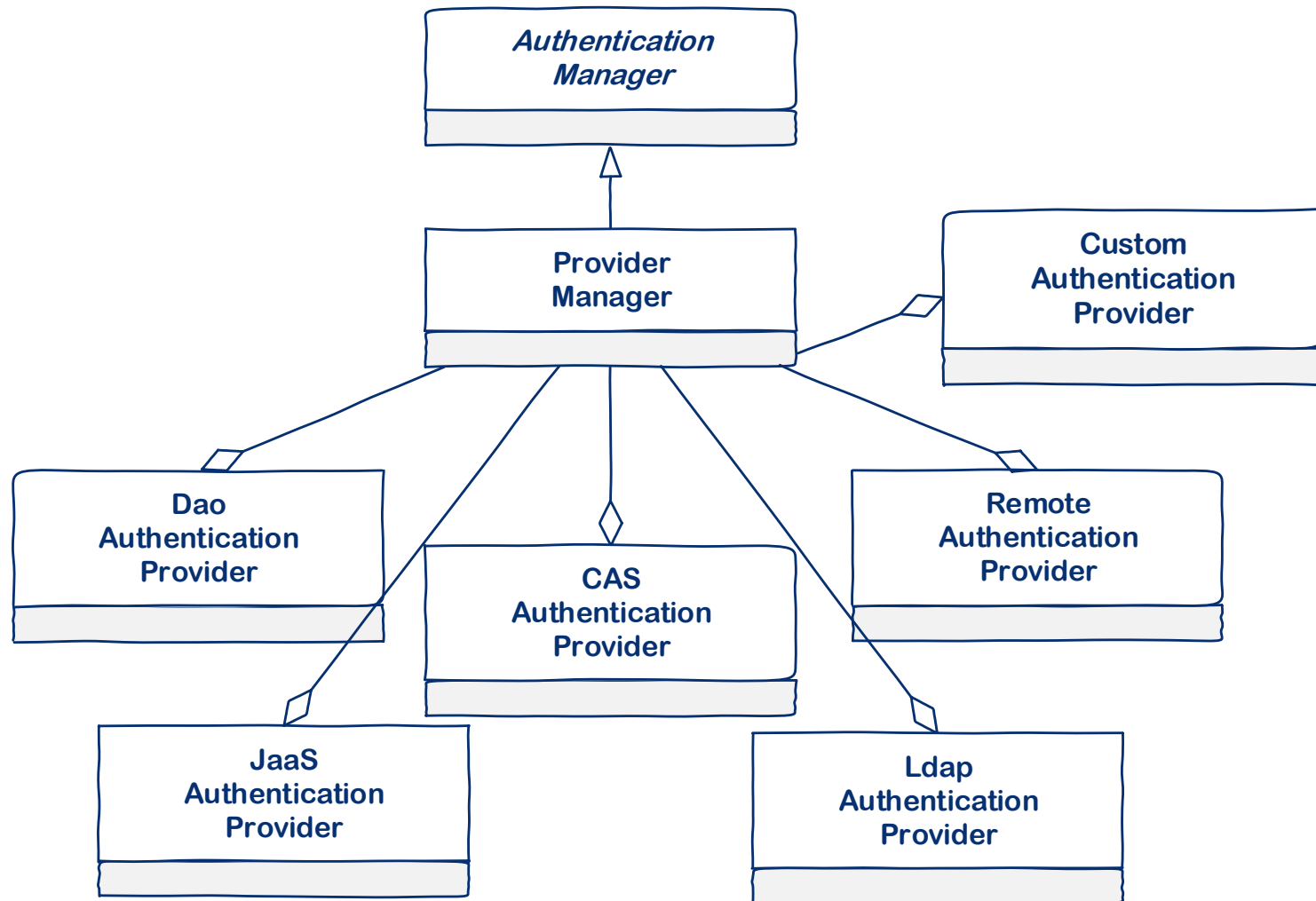
Benutzerverwaltung

- ▶ Soll ein Softwaresystem von mehreren Benutzern benutzt werden, so muss es über eine Benutzerverwaltung (Identity Store) verfügen, in dem die Benutzerattribute und Credentials abgelegt sind
- ▶ Identity Stores können sein:
 - ▶ Benutzer-Datenbank
 - ▶ LDAP / Active Directory
 - ▶ Filesystem
- ▶ Die Authentifizierung kann auch über einen externen Service erfolgen:
 - ▶ Identity-Provider (z.B. Google, GitHub, usw.)
 - ▶ CAS (Central Authentication Service)
 - ▶ RSA-Server (SecureID)

Ablauf Authentifizierung



Authentication Manager



Authentication Provider

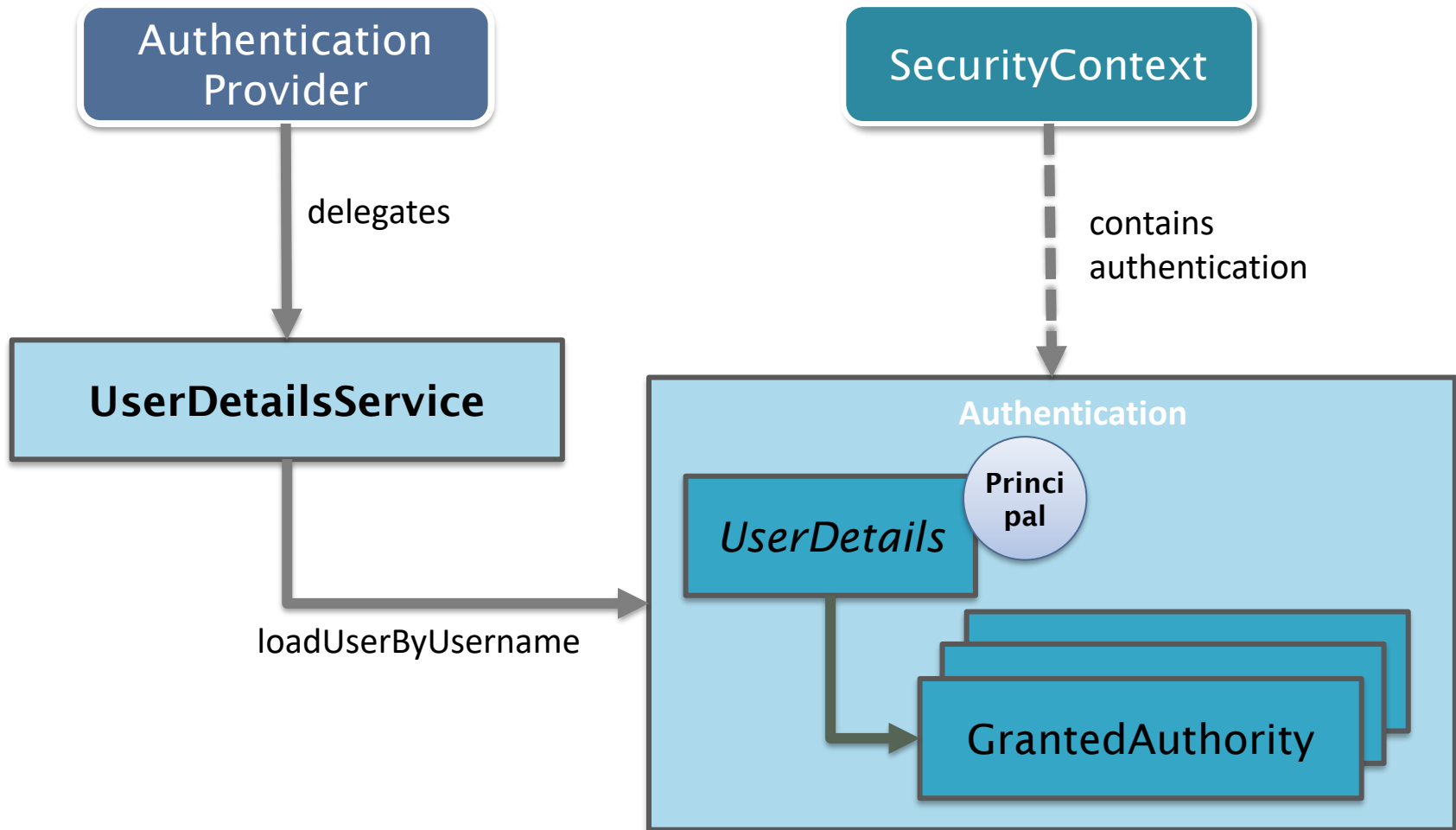
- ▶ Ein Authentication-Provider ist zuständig für die Authentifizierung und kann
 - ▶ ein `Authentication`-Objekt zurückgeben, wenn er die Credentials des Benutzers validieren konnte
 - ▶ eine `AuthenticationException` werfen, wenn die Validierung fehlschlägt
 - ▶ null zurückgeben, wenn er sich nicht entscheiden kann

```
public interface AuthenticationProvider {  
    boolean supports(Class<?> clazz);  
    Authentication authenticate(Authentication auth)  
        throws AuthenticationException;  
}
```

User Details Service

- ▶ Der Authentication Provider verwendet einen `UserDetailsService`, um Detailinformationen über Benutzer zu laden
- ▶ Dazu ruft er die Methode `loadUserByUsername` auf, welche ein `UserDetails`-Objekt zurückgibt oder im Fehlerfall eine `UsernameNotFoundException` wirft
- ▶ Der `UserDetailsService` kann zum Beispiel so implementiert werden, dass Benutzer von einer Datenbank oder einem Legacy System geladen werden

User Details Service



In-Memory User Details Service

```
@Configuration
public class SecurityConfig {

    @Bean
    public UserDetailsService inMemoryUsers(PasswordEncoder encoder) {
        UserDetails user = User.withUsername("user")
            .password(encoder.encode("12345"))
            .roles("USER").build();
        UserDetails admin = User.withUsername("admin")
            .password(encoder.encode("secr3t"))
            .roles("ADMIN").build();
        return new InMemoryUserDetailsManager(user, admin);
    }

    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder(10);
    }
}
```

Custom User Details Service

```
@Service
public class CustomUserDetailsService implements UserDetailsService {

    @Autowired
    private UserRepository userRepository;

    @Override
    public UserDetails loadUserByUsername(String username)
        throws UsernameNotFoundException {
        UserInfo userInfo = userRepository.findByUsername();
        if (userInfo == null)
            throw new UsernameNotFoundException("User not found");
        return User.withUsername(username)
            .password(userInfo.getPassword())
            .roles(userInfo.getRoles())
            .build();
    }
}
```

Password Encoder

- ▶ Spring verwendet Hash-Algorithmen mit einem Salt, um Passwörter mit hoher Güte sicher zu speichern
- ▶ Spring verwendet standardmässig den `DelegatingPasswordEncoder`, welcher
 - ▶ mehrere konfigurierbare Hashverfahren ermöglicht
 - ▶ das Format *{hashID}passwordHash* unterstützt
 - ▶ einen `DefaultEncoder` (in der Regel BCrypt) verwendet, falls die Hash-ID fehlt

Authentifizierungsmethode

- ▶ Die Authentifizierungsmethode definiert, welche Art Credentials verwendet werden, um Benutzer zu authentifizieren
- ▶ Die Authentifizierungsmethode wird in der `SecurityFilterChain` festgelegt, welche in einer Bean-Methode mit dem `HttpSecurity`-Objekt erzeugt werden kann
- ▶ Über das `HttpSecurity`-Objekt können zudem die URL-basierte Autorisierung von Requests, das Session-Management und die Verwendung von CSRF-Tokens konfiguriert werden

Authentifizierungsmethode

```
@Configuration
public class SecurityConfig {

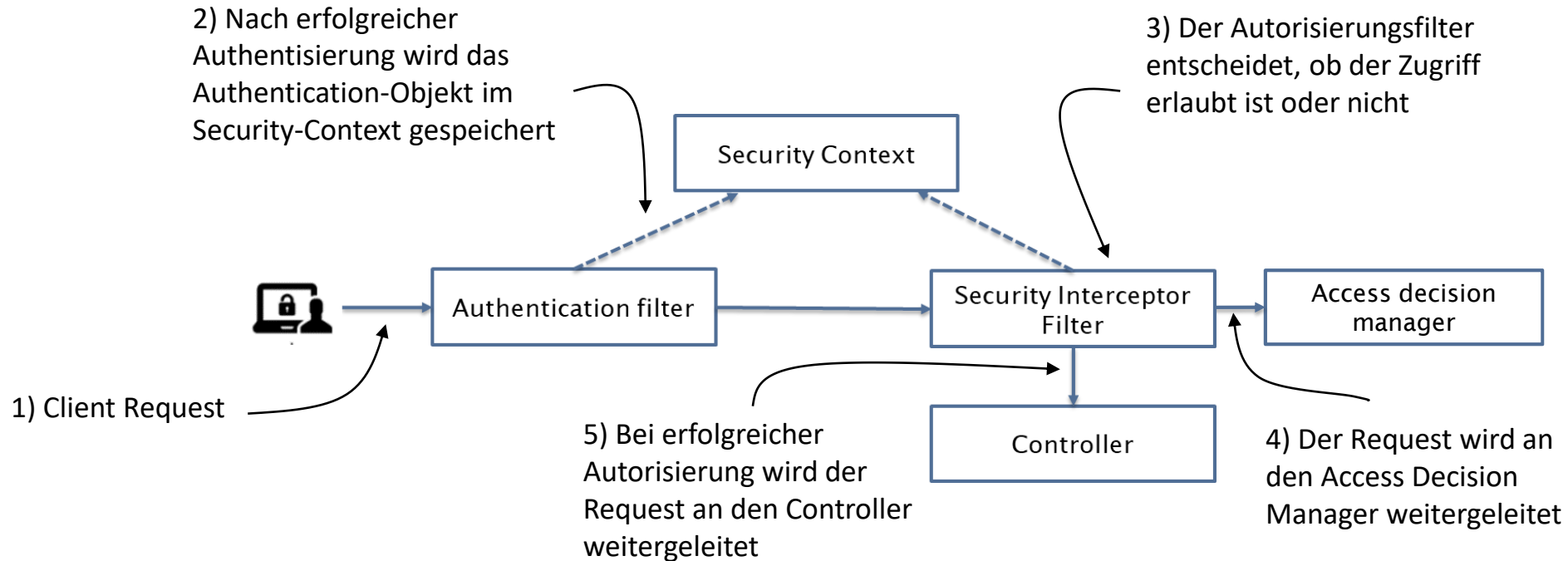
    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        http.httpBasic();
        http.authorizeRequests().anyRequest().authenticated();
        http.sessionManagement()
            .sessionCreationPolicy(SessionCreationPolicy.STATELESS);
        http.csrf().disable();
        return http.build();
    }
}
```

Autorisierung

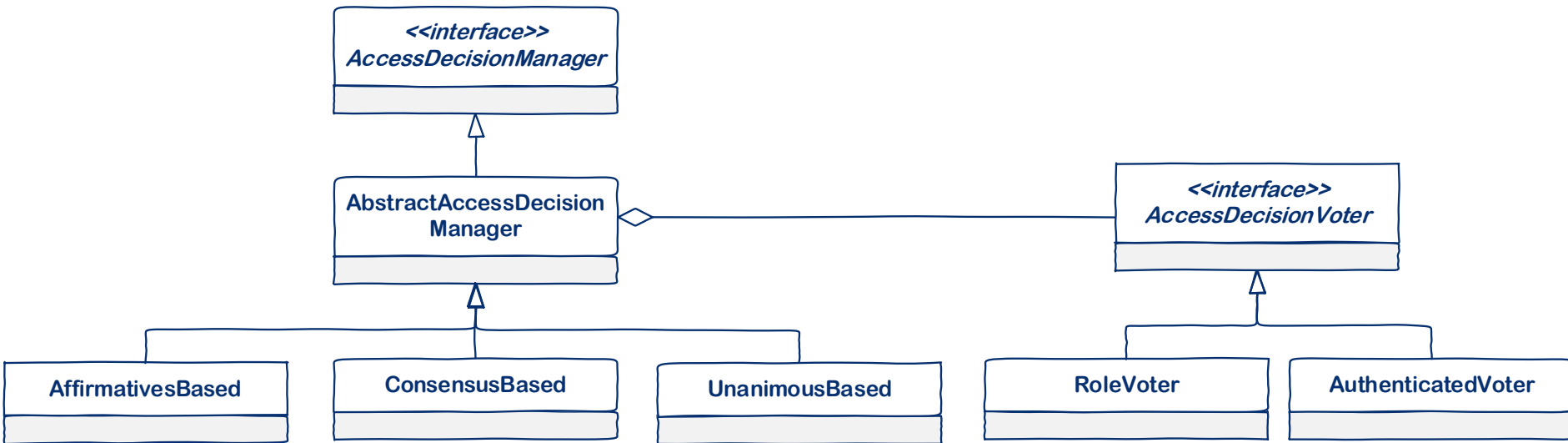
Grundlagen

- ▶ Nach der Authentisierung muss ein Benutzer (oder Client) für den Zugriff auf geschützte Ressourcen autorisiert (berechtigt) werden
- ▶ Autorisierungen können erfolgen auf Stufe
 - ▶ URL (mit Servlet-Filter)
 - ▶ Methode (mit dynamischen Proxies)
 - ▶ Objekt (mit ACL-Listen)
- ▶ Das `UserDetails`-Objekt enthält eine Liste von `GrantedAuthority`-Objekten, die Berechtigungen beschreiben als
 - ▶ grobgranulare Rollen (mit Präfix *ROLE_*)
 - ▶ feingranulare Berechtigungen
 - ▶ Access Control List (ACL) bezogen auf ein Objekt

Ablauf Autorisierung



URL-basierte Zugriffskontrolle



URL-basierte Zugriffskontrolle

- ▶ Die URL-basierte Zugriffskontrolle kann über das `HttpSecurity`-Objekt konfiguriert werden:
 - ▶ Mit der Methode `authorizeRequests` werden Zugriffsregeln für URLs erstellt (Security-Policy)
 - ▶ Mit der Methode `requestMatchers` wird definiert, für welche URLs und welche HTTP-Methode eine Konfiguration gilt (Scope)
 - ▶ Mit den Methoden `permitAll`, `hasRole` usw. werden Zugriffe erlaubt
- ▶ Die Konfiguration muss von fein nach grob erfolgen, da die erste passende Regel verwendet wird

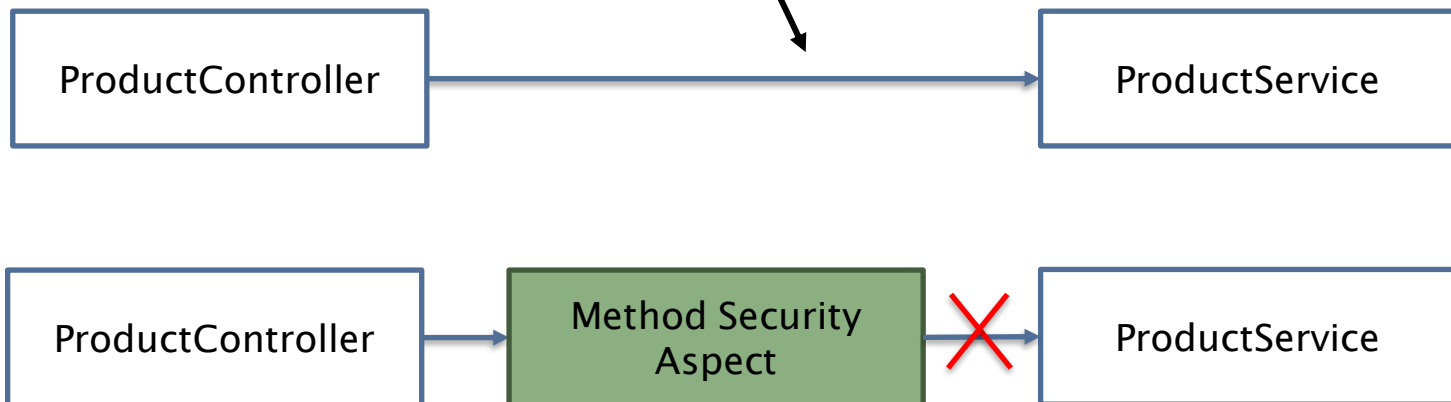
URL-basierte Zugriffskontrolle

```
@Configuration
public class SecurityConfig {

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        http.httpBasic();
        http.authorizeRequests()
            .requestMatchers("/login/**").permitAll()
            .requestMatchers("/view/**").authenticated()
            .requestMatchers("/admin/**").hasRole("ADMIN")
            .requestMatchers(HttpMethod.GET, "/monitor/**").hasRole("ADMIN")
            .anyRequest().denyAll();
        ...
    }
}
```

Methodenbasierte Zugriffskontrolle

Ohne Method-Security wird der Request vom Controller direkt an den Service weitergeleitet



Mit Method-Security wird der Request vom Security-Aspect intercepted (AOP) und der Request nur weitergeleitet, wenn die Sicherheitskonfiguration (Policy) es zulässt

Methodenbasierte Zugriffskontrolle

- ▶ Mit der Annotation `@EnableMethodSecurity` auf einer Konfigurationsklasse wird die Methodensicherheit aktiviert
- ▶ Über die Attribute `prePostEnabled` (Default), `jsr250Enabled` und `securedEnabled` können folgende Annotationen aktiviert werden:
 - ▶ `@PreAuthorize`, `@PostAuthorize`, `@PreFilter`, `@PostFilter` (SPEL-Ausdrücke)
 - ▶ `@PermitAll`, `@DenyAll`, `@RolesAllowed`, `@RunAs` (Servlet Security)
 - ▶ `@Secured` (ursprüngliche Spring-Annotation)
- ▶ Methodensicherheit kann in einem Controller oder Service auf Stufe Methode oder Klasse verwendet werden

Methodenbasierte Zugriffskontrolle

```
public class PostService {  
  
    @PreAuthorize("hasAuthority('POST')")  
    public void doPost(Post post) { ... }  
  
    @PreAuthorize("hasRole('USER')")  
    public void doPost(Post post) { ... }  
  
    @PreAuthorize("#post.name == authentication.name")  
    public String updatePost(Post post) { ... }  
  
    @PostAuthorize("returnObject.name = authentication.name")  
    public Post readPost(Long id) { ... }  
}
```

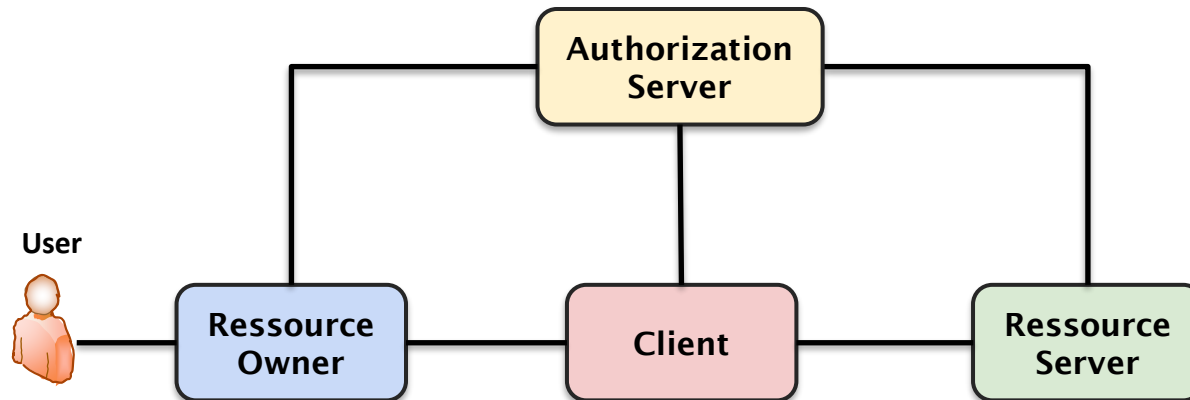
OAuth2 und OpenID Connect

Einführung

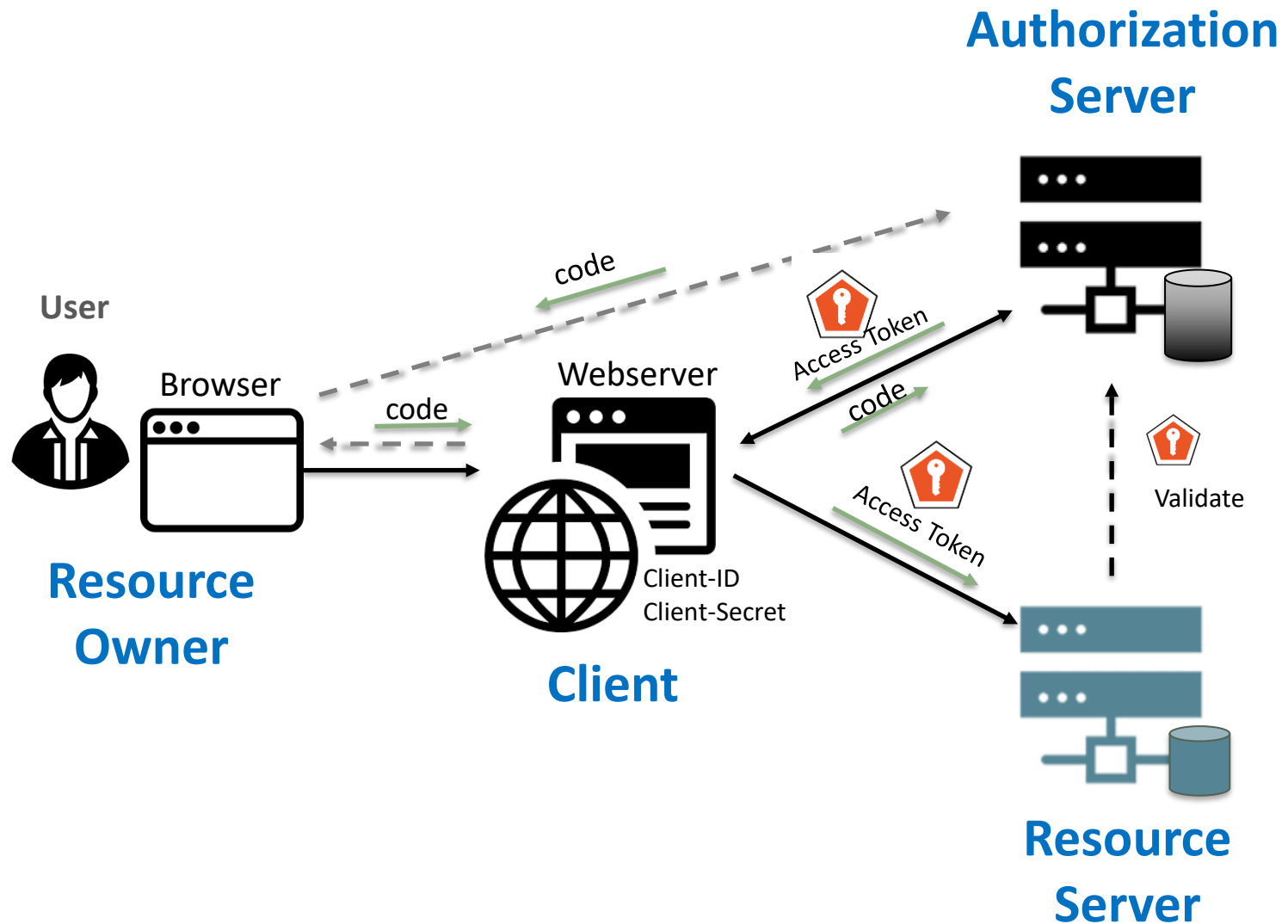
- ▶ OAuth 2.0 ist ein Autorisierungsprotokoll und -framework
- ▶ Endbenutzer können einer Zielanwendung erlauben, auf ihre Daten eines Dienstes zuzugreifen, ohne ihre Credentials preisgeben zu müssen (delegierte Autorisierung)
- ▶ OpenID Connect (OIDC) ergänzt OAuth2 mit Identitätsdaten und ermöglicht so einer Zielanwendung, die Identität von Benutzern zu prüfen (delegierte Authentifizierung)

OAuth2 Parteien

- ▶ Der **Ressource Owner** (Benutzer) kann den Zugriff auf eine Ressource (Daten oder Dienste) gewähren
- ▶ Der **Client** (Zielapplikation) möchte auf eine geschützte Ressource des Benutzers zugreifen
- ▶ Der **Autorisierungsserver** authentisiert den Benutzer und stellt dem Client eine Erlaubnis (Token) für den Zugriff auf die Ressource aus
- ▶ Der **Ressource-Server** enthält die Ressourcen des Benutzers und liefert diese bei erfolgreicher Autorisierung aus

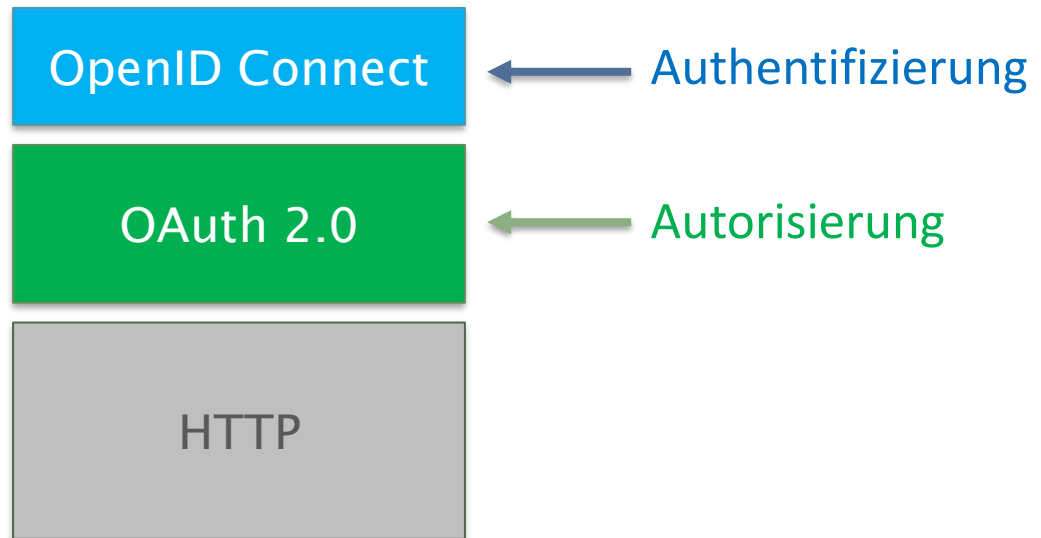


OAuth2 Ablauf



OpenID Connect (OIDC)

- ▶ OpenID Connect ist ein Authentifizierungsprotokoll, das auf dem OAuth2-Framework aufbaut
- ▶ Es ermöglicht Drittanwendungen, die Identität eines Benutzers anhand eines ID-Tokens zu überprüfen und grundlegende Benutzerinformationen abzurufen

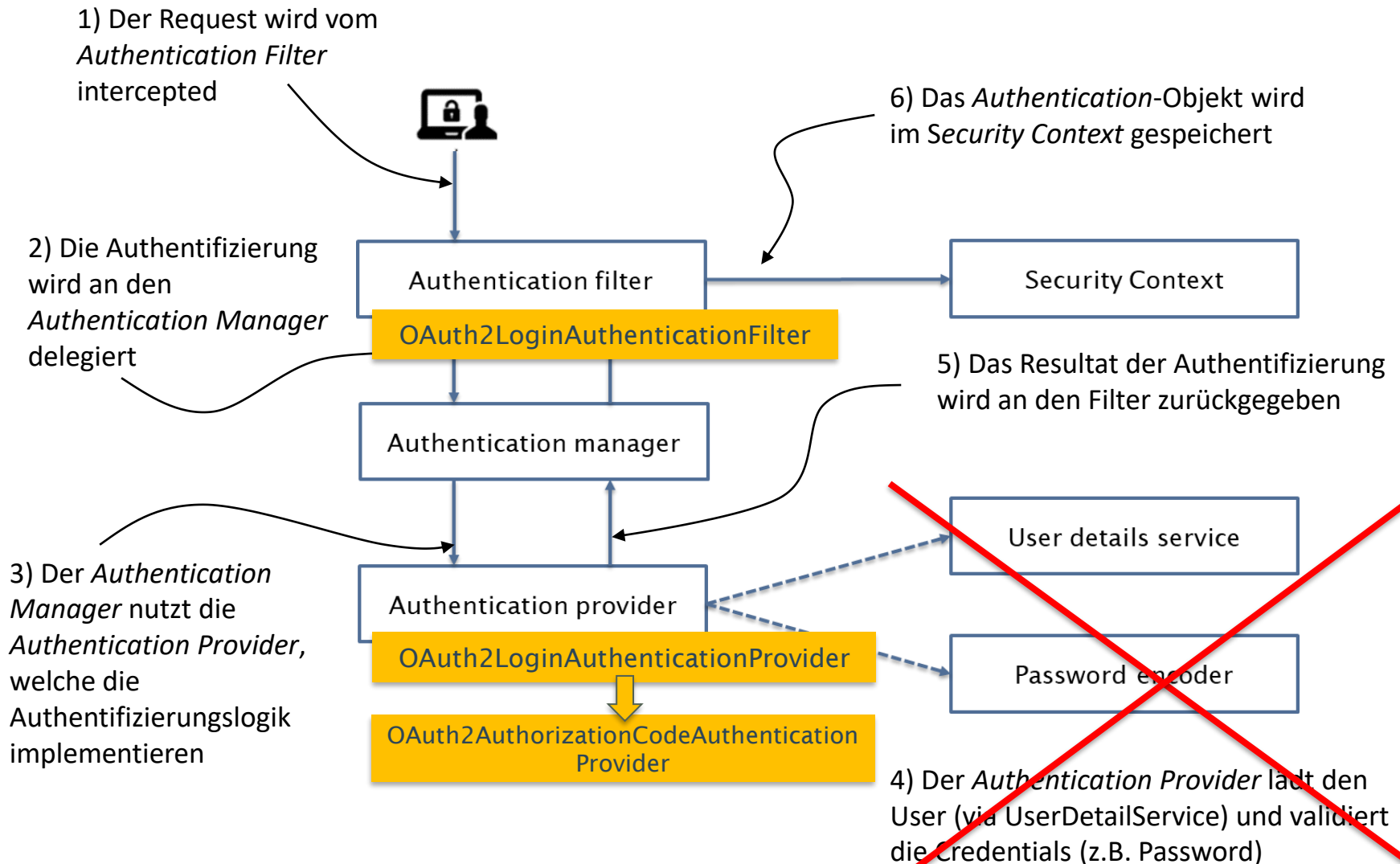


ID-Token

- ▶ Ein OAuth2-Client kann die OpenID-Erweiterung verwenden, indem er den Scope **openid** zu einem Autorisierungs-Request hinzufügt
- ▶ Der Client erhält dann ein ID-Token, das ein signiertes JSON-Web-Token (JWT) ist und verschiedene Claims enthalten kann:
 - ▶ Registrierte Claims (sub, iss, exp, iat)
 - ▶ Standard-Claims (name, given_name, email)
 - ▶ Adressen-Claims (country, street_address)
 - ▶ Selbstdefinierte Claims

Ressourcen-Server

Ablauf Authentifizierung



Maven-Konfiguration

- ▶ Ein OAuth2 Resource-Server kann sehr einfach mit dem entsprechenden Starter konfiguriert werden

```
<dependency>  
  <groupId>org.springframework.boot</groupId>  
  <artifactId>spring-boot-starter-oauth2-resource-server</artifactId>  
</dependency>
```


Konfiguration Autorisierungsserver

- ▶ Der Ressource-Server muss prüfen können, ob ein Access Token (in der Regel ein JWT-Token) gültig ist oder nicht
- ▶ Dazu muss der entsprechende Autorisierungsserver, welcher das Token ausgestellt hat, im Ressourcen-Server konfiguriert werden

```
spring.security.oauth2.resourceserver.jwt.issuer-uri=  
https://example.org/auth/realms/foo
```

Web Security Konfiguration

- ▶ Die Security-Konfiguration eines Ressource-Servers kann folgendermassen aussehen:

```
@Configuration
public class SecurityConfig {

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        http.oauth2ResourceServer().jwt();
        http.authorizeRequests()
            ...
        http.sessionManagement()
            .sessionCreationPolicy(SessionCreationPolicy.STATELESS);
        http.csrf().disable();
    }
}
```

Custom Claims Mapping

- ▶ Spring Boot Security bietet Converter an, um JWT-Tokens individuell auszuwerten:

```
@Configuration
public class SecurityConfig {

    @Bean
    public JwtAuthenticationConverter jwtConverter() {
        JwtAuthenticationConverter jwtConverter =
            new JwtAuthenticationConverter();
        jwtConverter.setPrincipalClaimName("username");
        JwtGrantedAuthoritiesConverter authConverter =
            new JwtGrantedAuthoritiesConverter();
        authConverter.setAuthoritiesClaimName("roles");
        authConverter.setAuthorityPrefix("ROLE_");
        jwtConverter.setJwtGrantedAuthoritiesConverter(authConverter);
        return jwtConverter;
    }
}
```

Authentication-Objekt

- ▶ Das Authentication-Objekt ist ein Objekt der Klasse `JwtAuthenticationToken`
- ▶ Der User, die Authorities und das Token selbst können einfach abgefragt werden

```
@GetMapping(path = "/authInfo")
public AuthInfo getAuthInfo(Authentication auth) {
    String name = auth.getName();
    List<String> authorities = auth.getAuthorities().stream()
        .map(GrantedAuthority::getAuthority).toList();
    String token = ((JwtAuthenticationToken) auth).getToken().getTokenValue();
    return new AuthInfo(name, authorities, token);
}
```

Zusammenfassung

Microservices Architektur

