Bachelor of Science (BSc) in Informatik

Modul Advanced Software Engineering 1 (ASE2)

LE 04 – Spring Framework

Spring MVC

Spring Security

Institut für Angewandte Informationstechnologie (InIT)

Walter Eich (eicw) / Matthias Bachmann (bacn)

https://www.zhaw.ch/de/engineering/institute-zentren/init/

Agenda

- Teil LE04-1: Spring MVC
- Teil LE04-2: Spring Security

Einführung Spring MVC

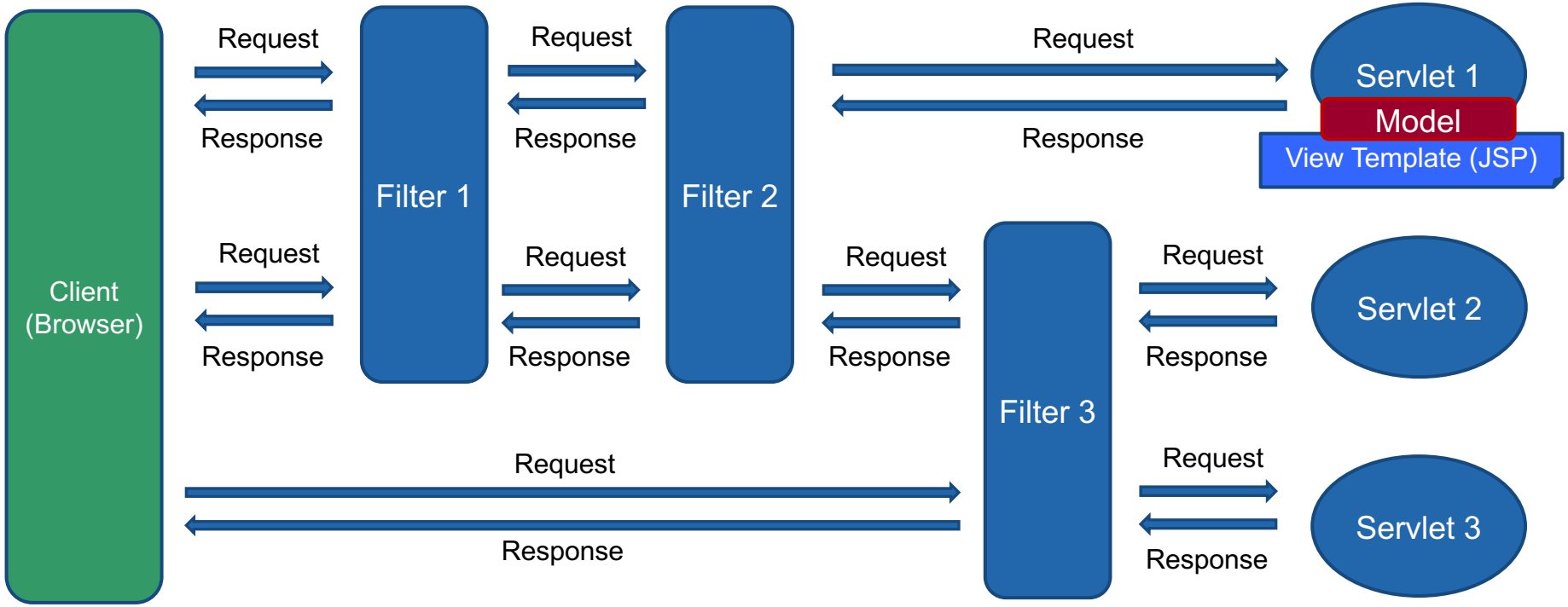
- Einführung MVC
  - Dispatcher Servlet
  - Application Context
- Setup
- Controller inkl. Annotationen
  - RequestParam
  - HttpStatusCode und MediaTypes
  - ResponseBody
- RestController
  - ResponseStatus
  - ErrorHandler und Controller Advice
- Rest Client mit RestTemplate

Lernziele LE 04-1 – Spring MVC

- Die Studierenden..
  - können einen Controller bzw. RestController erstellen
  - Können mit Hilfe von `@RequestMapping`, `@RequestParam` oder `@PathVariable` die URL konfiguriere bzw. Parameter aus dem Request extrahieren
  - Können mit `@ResponseBody` und `@ResponseStatus` eine Antwort an den Client zurücksenden
  - Verstehen die Fehlerbehandlung mit `@ErrorHander` oder `@ControllerAdvice`
  - Können mit Hilfe von `RestTemplate` einen RestClient erstellen

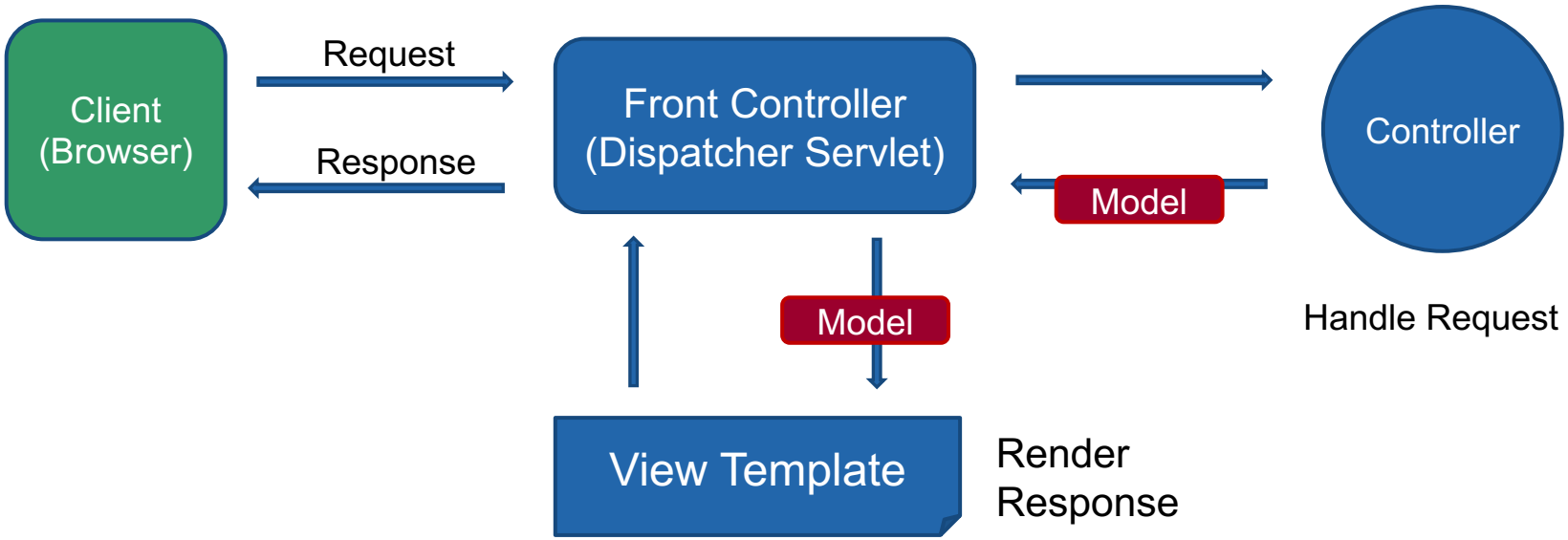
Was sind Standard Java Servlet's?

- Filter können Request und Response abfangen (intercept - URL basiert)
- Ein Servlet kann http-Methoden behandeln (URL basiert)
- Annotationen: @WebServlet, @WebFilter, @WebListener, @WebInitParam



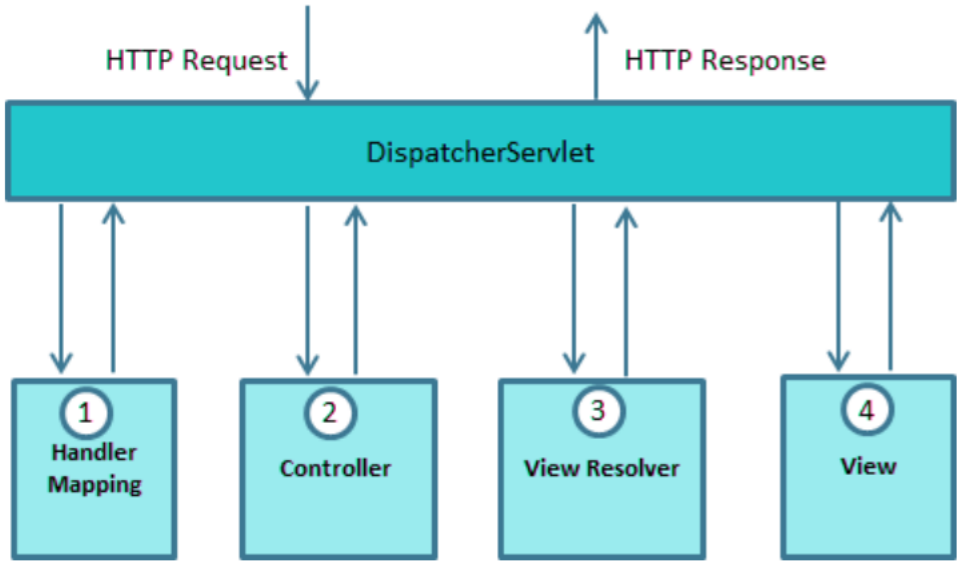
Was ist Spring MVC? (1/2)

- Basiert auf Java Servlets
- Ein (Request) anfrage-basiertes Web Framework
- Implementiert das MVC-Pattern
- Ein *Front-Controller* (*Dispatcher Servlet*) delegiert die Anfragen an die entsprechenden *Controller*



Was ist Spring MVC? (2/2)

- `Das DispatcherServlet` behandelt die HTTP-Requests und Responses
  1. Nach dem Empfang eines HTTP-Requests konsultiert das `DispatcherServlet` das **`HandlerMapping`** um den entsprechenden Controller aufzurufen
  2. Der **`Controller`** ruft die entsprechende Service Methode (GET, POST, etc) auf. Die Service-Methode fügt Model-Daten hinzu und gibt den Namen einer View zurück.
  3. Das `DispatcherServlet` nimmt die Hilfe des **`ViewResolvers`** in Anspruch um den definierten View aufrurufen
  4. Das `DispatcherServlet` übergibt die Modeldaten and den **`View`**, welcher die Daten gerendert an den Browser übergibt



https://docs.spring.io/spring/docs/current/spring-framework-reference/html/mvc.html

Setup für Spring MVC (1/3)

- Hinzufügen der *Spring MVC Starter Dependency* zur pom/build.gradle-Datei ergibt: *Spring MVC*, *Jackson* (Mapping), *Hibernate Validator* und *embedded Tomcat*.
- *Thymeleaf Starter* stellt eine *Templating Engine* um HTML Seiten zu rendern zur Verfügung (Alternative zu *JSP*).

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

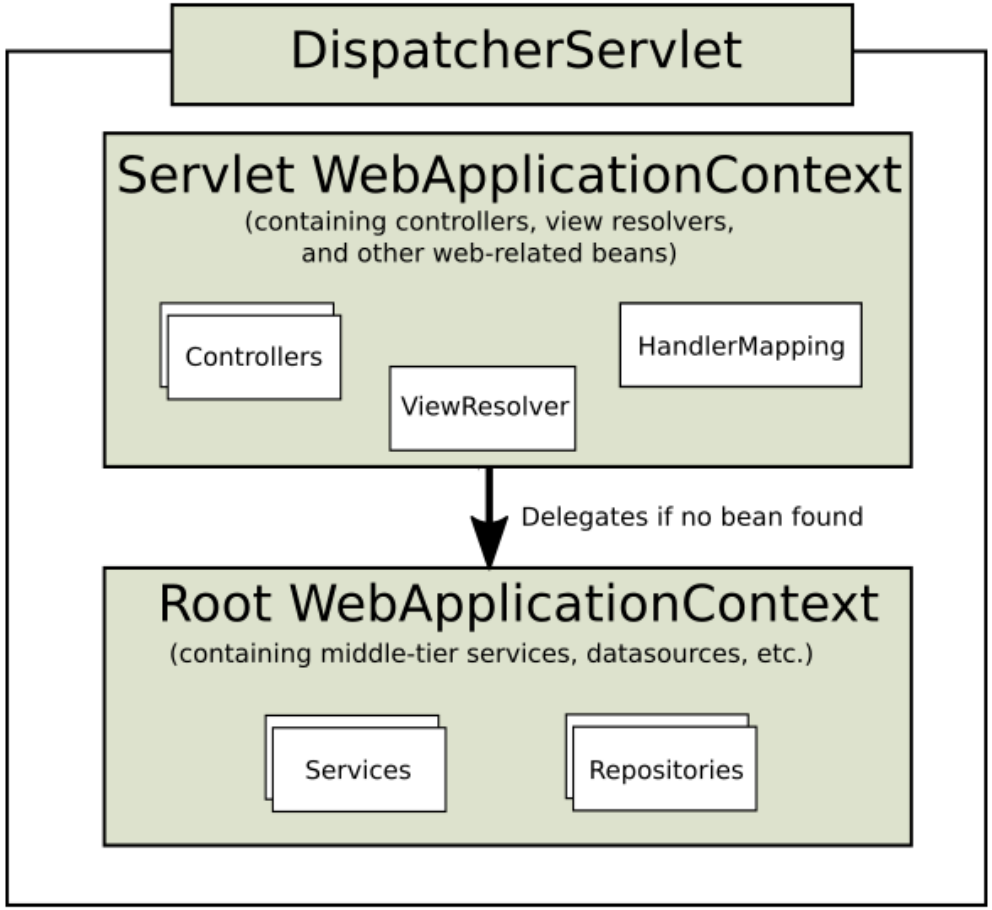
```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-thymeleaf</artifactId>
</dependency>
```

Setup für Spring MVC (2/3)

- Starter Projekte konfigurieren die Infrastruktur-Spring-Beans automatisch
  - Thymeleaf
    - Templating Engine (für Auslieferung von HTML)
    - Caching enabled -> Spring Boot Devtools
  - Spring MVC
    - Enthält ContextLoaderListener und DispatcherServlet
    - `@EnableWebMvc` mit Formatters, Converters und Validators
    - Static resources werden ausgeliefert aus resources `/static`, `/public`, oder `/META-INF/resources`
    - Templates (Thymeleaf/JSP/..) werden ausgeliefert von `resources/templates`
    - Stellt ein Default `/error` mapping zur Verfügung

Setup für Spring MVC (3/3)
Context-Hierarchie

- Controller
  - @Controller oder
  - @RestController
- View
  - Template Engines
    - FreeMarker
    - Groovy
    - Thymeleaf
    - Mustache
  - JSP
  - Grails (Groovy)
  - JSF



https://docs.spring.io/spring-framework/docs/current/reference/html/web.html#mvc

Einfacher Controller mit einer View

- Get-Request auf */*
- Model model enthält die anzuzeigenden Daten
- Der Return-Wert "home" bezieht sich auf den Template-Namen

```
@Controller
public class SimpleController {

    @Value("${spring.application.name}")
    private String appName;

    @GetMapping("/")
    public String homePage(Model model) {
        model.addAttribute("appName", appName);
        return "home";
    }
}
```

Request Mappings

- Generic
  - `@RequestMapping`
- HTTP default methods
  - `@GetMapping`
  - `@PostMapping`
  - `@PutMapping`
  - `@DeleteMapping`
  - `@PatchMapping`

Return Werte von @RequestMapping

- Die mit @RequestMapping annotierte Methode kann verschiedene Rückgabewerte haben. Einige davon sind unten beschrieben
- **Return-Werte-Beschreibung**
  https://docs.spring.io/spring-framework/docs/current/reference/html/web.html#mvc-ann-return-types
  - ModelAndView — Enthält Model and View-Information
  - String — Repräsentiert den View-Name
  - View — Repräsentiert das View Objekt
  - Model/Map — Enthält die Model Daten für den View.
    - Der View wird implizit durch die RequestToViewNameTranslator Klasse bestimmt
  - Void — Die Methode ist für den View selber verantwortlich
  - @ResponseBody — Der Returnwert ist direkt die HTML Body Information

@RequestMapping

- Mehrere Maps

```
@RequestMapping(value={"/method1","/method1/second"})
@ResponseBody
public String method1(){
  return "method1";
}
```

- Definierte HTTP-Verben

```
@RequestMapping(value="/show", method={RequestMethod.POST,RequestMethod.GET})
@ResponseBody public String method3() {
  return "method3";
}
```

Controller Methoden

- `@PathVariable` für ein URL-Pfad Fragment (`/{id}`)
- `@RequestParam` Query Parameters zu extrahieren (`/customers?addressId=14534`)

```
@RequestMapping(path="/customers", method=RequestMethod.GET)
public String show(HttpServletRequest request, Model model) { … }

@GetMapping("/customers/{id}/address/{addressId}")
public String show(@PathVariable("id") Long id,
                   @PathVariable int addressId, Locale locale, Model model,
                   @RequestHeader("user-agent") String agent) { … }

@GetMapping("/customers")
public String show(@RequestParam Long id,
                   @RequestParam("addressId") int addressId, Principal user,
                   Map<String, Object> model, HttpSession session,
                   @CookieValue("acceptedDate") String acceptedDate) { … }
```

Http Status Code

- Status Codes signalisieren das Ergebnis des Servers den Request auszuführen
- Aufgeteilt in Kategorien
  - 1XX: Informational
  - 2XX: Success
  - 3XX: Redirection
  - 4XX: Client Error
  - 5XX: Server Error

Media Types

- Accept & Content-Type HTTP Headers
- Client und Server beschreiben den Inhalt (Content)

@ResponseBody

- Konverter für die Antwort (Response) durch `@ResponseBody`
- Konverter behandelt das Rendering, ohne entsprechenden *View*
- Nimmt den *Accept-Header* für die *Content Negotiation*

```
@Controller
public class CustomerController {

    @GetMapping(path="/customers/{id}")
    public @ResponseBody Customer showCustomer(
                @PathVariable("id") long id) { … }
}
```

```
GET /employees/42
Host: www.jobs.com
Accept: application/json
```

```
HTTP/1.1 200 OK
Date: ...
Content-Length: 723
Content-Type:
application/json
{
"employee": {
"id": 123,
"address": [ … ], … }
}
```

@RestController Vereinfachung

- Mit `@RestController` wird `@ResponseBody` nicht benötigt

```
@Controller
public class CustomerController {

    @GetMapping(path="/customers/{id}")
    public @ResponseBody Customer showCustomer(
                @PathVariable("id") long id) { … }
}
```

```
@RestController
public class CustomerController {

    @GetMapping(path="/customers/{id}")
    public Customer showCustomer(
                @PathVariable("id") long id) { … }
}
```

@ResponseStatus

- Der Controller gibt *per Default* den Status *200 OK* zurück
- `@ResponseStatus` um den Status Code zu überschreiben
- `@ResponseStatus` kann auch bei void-Methoden verwendet werden

```
@RestController
public class CustomerController {

    @PutMapping(path="/customers/{id}")
    @ResponseStatus(HttpStatus.NO_CONTENT)
    public void updateCustomer(
            @PathVariable("id") long id, Customer customer) {
                // Update employee data
    }
}
```

@ErrorHandler & @ControllerAdvice

- `@ErrorHandler` Methode in der Controller Klasse behandelt Exception
- Unterstützt `@ResponseStatus` und CustomError Messages
- `@ControllerAdvice` sorgt für die Verfügbarkeit für alle Controllers

```
@Slf4j
@ControllerAdvice
public class RestCustomerControllerAdvice {

   @ExceptionHandler(CustomerNotFoundException.class)
   @ResponseStatus(HttpStatus.NOT_FOUND)
   public void notFound(CustomerNotFoundException e){
      log.error("Error occured: {}", e);
   }
}
```

Rest Client mittels RestTemplate


```
RestTemplate restTemplate = new RestTemplate();
```

HTTP	RESTTEMPLATE
DELETE	delete(String, String...)
GET	getForObject(String, Class, String...)
HEAD	headForHeaders(String, String...)
OPTIONS	optionsForAllow(String, String...)
POST	postForLocation(String, Object, String...)
PUT	put(String, Object, String...)

```
private static void getCustomers()  {
   final String uri = "http://localhost:8080/customers";

   RestTemplate restTemplate = new RestTemplate();

   String result = restTemplate.getForObject(uri, String.class);
   System.out.println(result);
}
```

Hands-on

- Neuer Branch erstellen
- Package Controller erstellen
- RestController für `Customer` und `Checkout` einfügen
- Exception Klassen `CustomerNotFoundException` und `CheckoutNotFoundException` einfügen
- UnitTest `CustomerApiRestControllerTest` einfügen
- In `CheckoutModel` Klasse `@OneToOne(cascade=CascadeType.ALL)`
- `Setter und Getter für die Id Modelklassen Customer und Checkout`

  https://github.zhaw.ch/bacn/ase2-spring-boot-hellorest/blob/rest/readme/rest.md

Zusammenfassung

- Architektur von Spring MVC mit Dispatcher Servlet und Application Context
- Controller inkl. Annotationen `@RequestMapping`, `@RequestParam` oder `@PathVariable`.
  - ResponseStatus, ResponseBody
  - ErrorHandler und Controller Advice
- Rest Client mit RestTemplate

Einführung Spring Security

- Spring Security
  - Was ist Spring Security
  - Setup
- Architektur Spring Security
- Authentisierung
- Autorisierung
- Rest Security mit JWT

Lernziele LE 04-2 – Spring Security

- Die Studierenden…
  - Verstehen die Architektur von Spring Security
  - Können die Benutzer Authentisierung in Spring Security anwenden
  - Verstehen wie die Autorisierung mit Spring umgesetzt werden kann
  - Können die Authentisierung mit JWT für ein zustandsloses Backend umsetzen.

Was ist Spring Security?

- Ein leistungsfähiges und anpassbares Authentisierungs- und Autorisierungs-Framework
- Setzt auf Spring Framework auf
- De-facto Standard für die Absicherung von Spring-basierten Anwendungen

https://docs.spring.io/spring-security/site/docs/current/reference/html5/

Setup für Spring Security (1/2)

- Für Spring Security muss die `spring-boot-starter-security` hinzugefügt werden.

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

Setup für Spring Security (2/2)

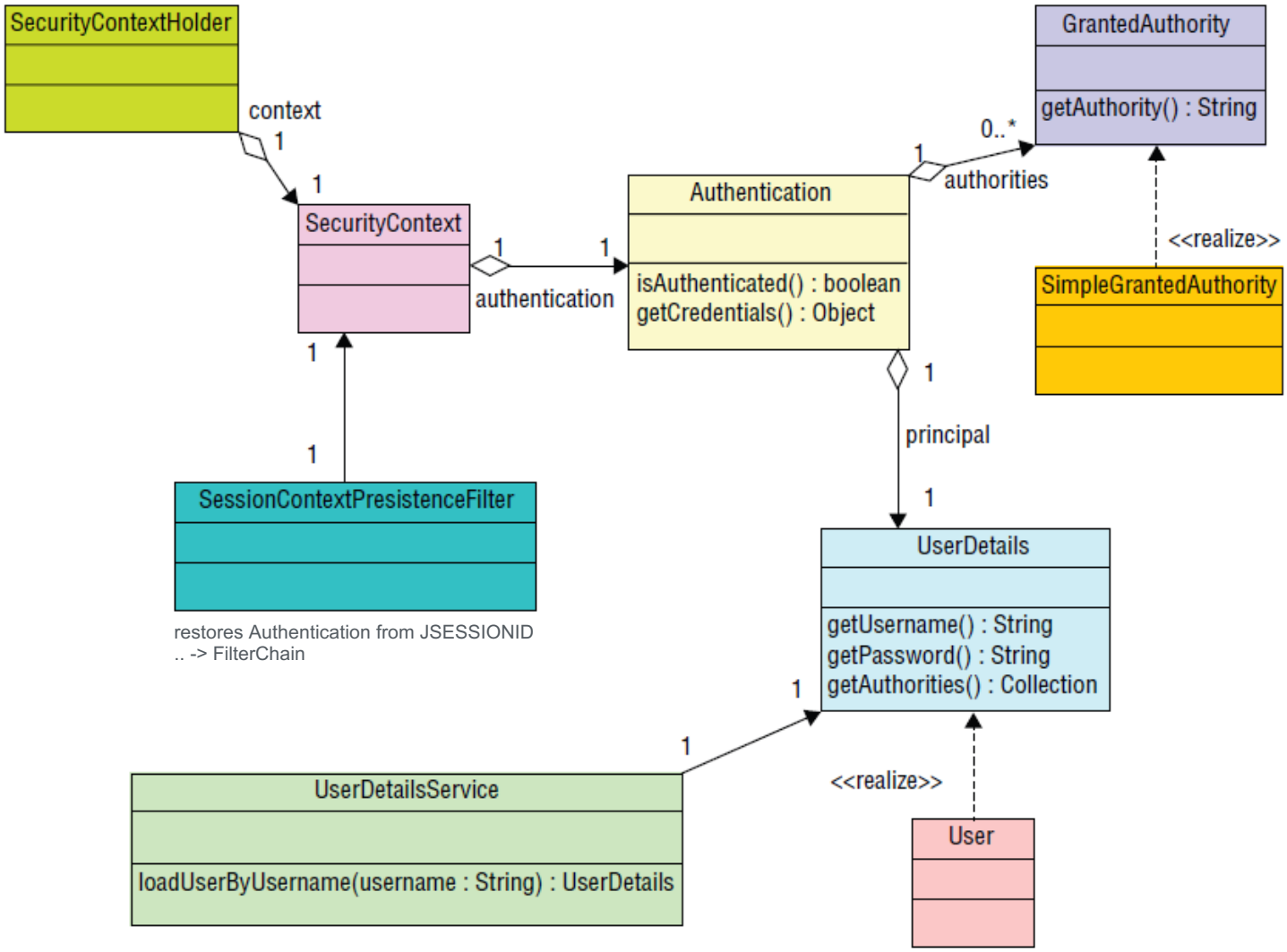
- Starter konfiguriert die Infrastruktur-Spring-Beans automatisch
  - SpringSecurityFilterChain
  - Globale Methoden Sicherheit
  - BasicAuth
- Das zufällige Password wird in der Konsole ausgegeben

```
<!- console log of the password from a default user -->
Using generated security password: 78fa095d-3f5c-88b4-r8ee-e24c31d5cf35
```

Die Architektur von Spring Security

- Authentisierung (who are you?)
  - Unterstützt werden eine grosse Anzahl von Authentisierungs-protokollen von BASIC über LDAP bis hin zu OAuth2
- Autorisierung (what are you allowed to do?)
- Unterstützt werden
  - URL-basierter Zugriffsschutz (für Webanwendungen)
  - Methoden-basierter Zugriffsschutz
  - Access Control Listen für feingranularen Zugriffsschutz auf Stufe von Ressourcen und/oder Objekten
- Definiert kann die Security mit Hilfe von
  - Annotationen und Konfigurationsklassen (Spring Boot)
  - XML Konfiguration (Spring)

Die Architektur von Spring Security



Die Architektur von Spring Security
Begriffe

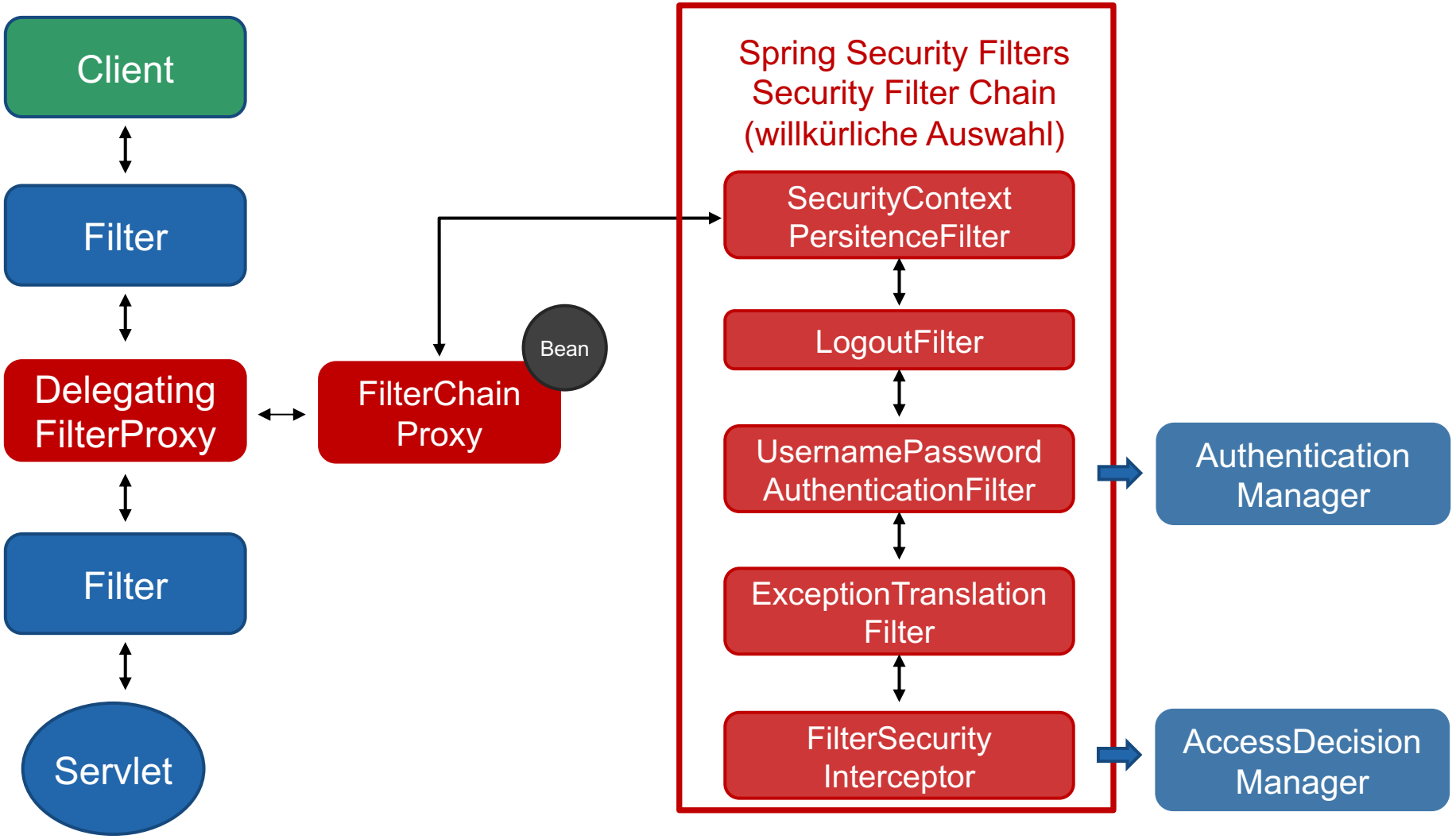
- SecurityContextHolder
  - ermöglicht den Zugriff auf den SecurityContext.
- SecurityContext
  - enthält die Authentication und ev. request-spezifischen Sicherheitsinformationen.
- Authentication
  - Bestätigt die Wahrheit der Credentials
  - repräsentiert den Principal in einer Spring-Security-spezifischen Art.
- Principal
  - Benutzer oder System
- GrantedAuthority
  - READ_AUTHORITY, WRITE_AUTHORITY, UPDATE_AUTHORITY, DELETE_AUTHORITY
  - reflektiert der anwendungsweiten Zugriff (Permission) für einen Principal. Werden normalerweise vom UserDetailsService geladen.

Die Architektur von Spring Security
Begriffe

- UserDetailsService
  - erzeugt ein UserDetails-Objekt aufgrund einer ID, einer Email-Adresse oder eines Zertifikats.
- UserDetails
  - stellt die notwendige Information bereit für ein Authentication Objekt von der Anwendungs-DAO oder einer anderen Quelle.
- Authorization
  - Legt die Zugriffsregeln für einen Principal fest
- Authority
  - Rolle oder Erlaubnis
- Secured Item
  - Resource mit URL oder einer methode

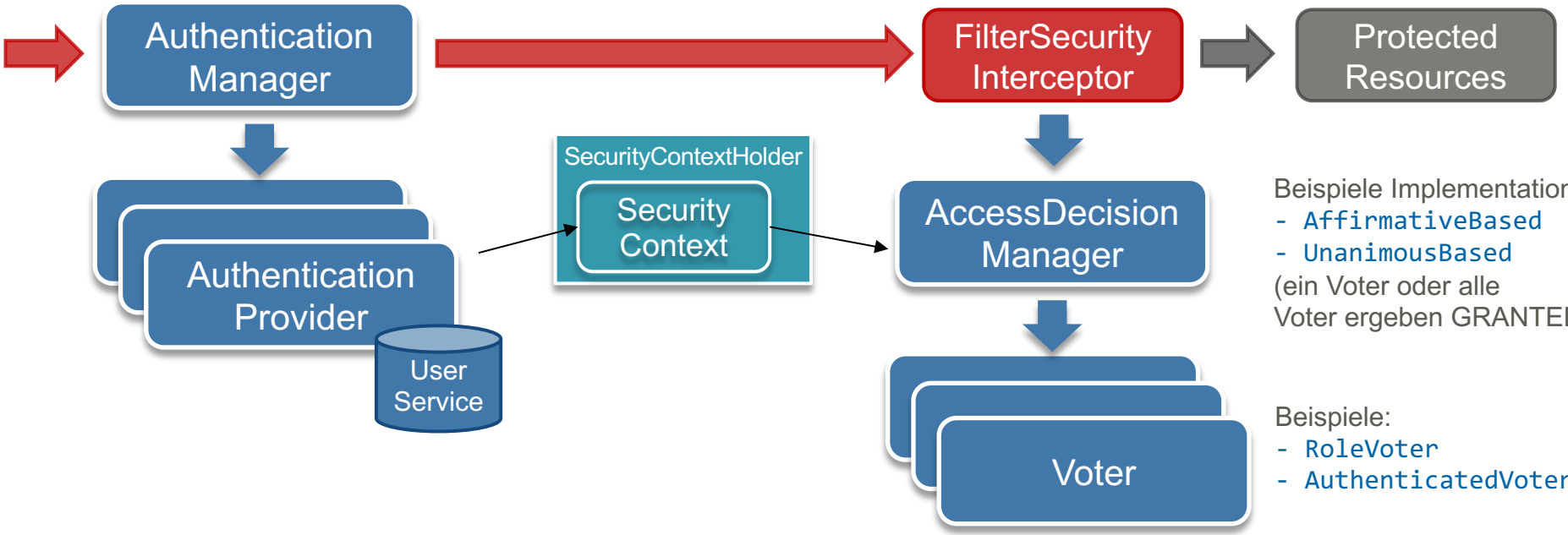
Die Architektur von Spring Security

Spring Security Filter Chain



Die Architektur von Spring Security

Authentisierung, Autorisierung und Security Context



Beispiele Implementation:
- `AffirmativeBased`
- `UnanimousBased`

(ein Voter oder alle Voter ergeben GRANTED)

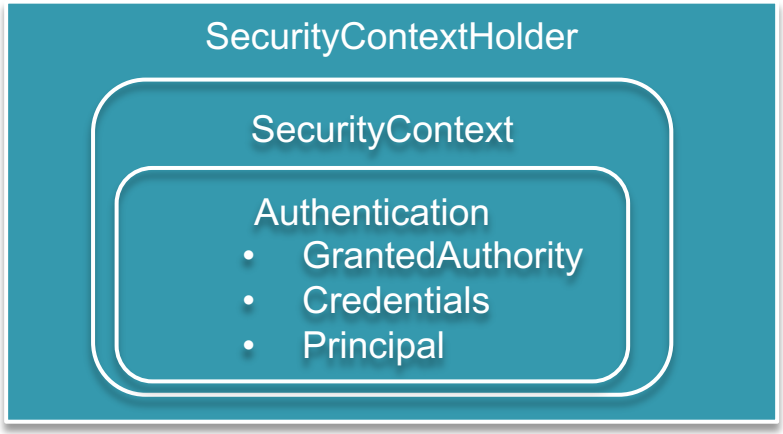
Beispiele:
- `RoleVoter`
- `AuthenticatedVoter`

AuthenticationProvider for
DAO (Jdbc), OpenID, RememberMe, JASIG CAS
Ldap, inMemory, UserDetailsServices

Rückgabewerte eines `AccessDecisionVoters`:
ACCESS_GRANTED bzw. ACCESS_DENIED

Die Architektur von Spring Security

SecurityContextHolder und SecurityContext



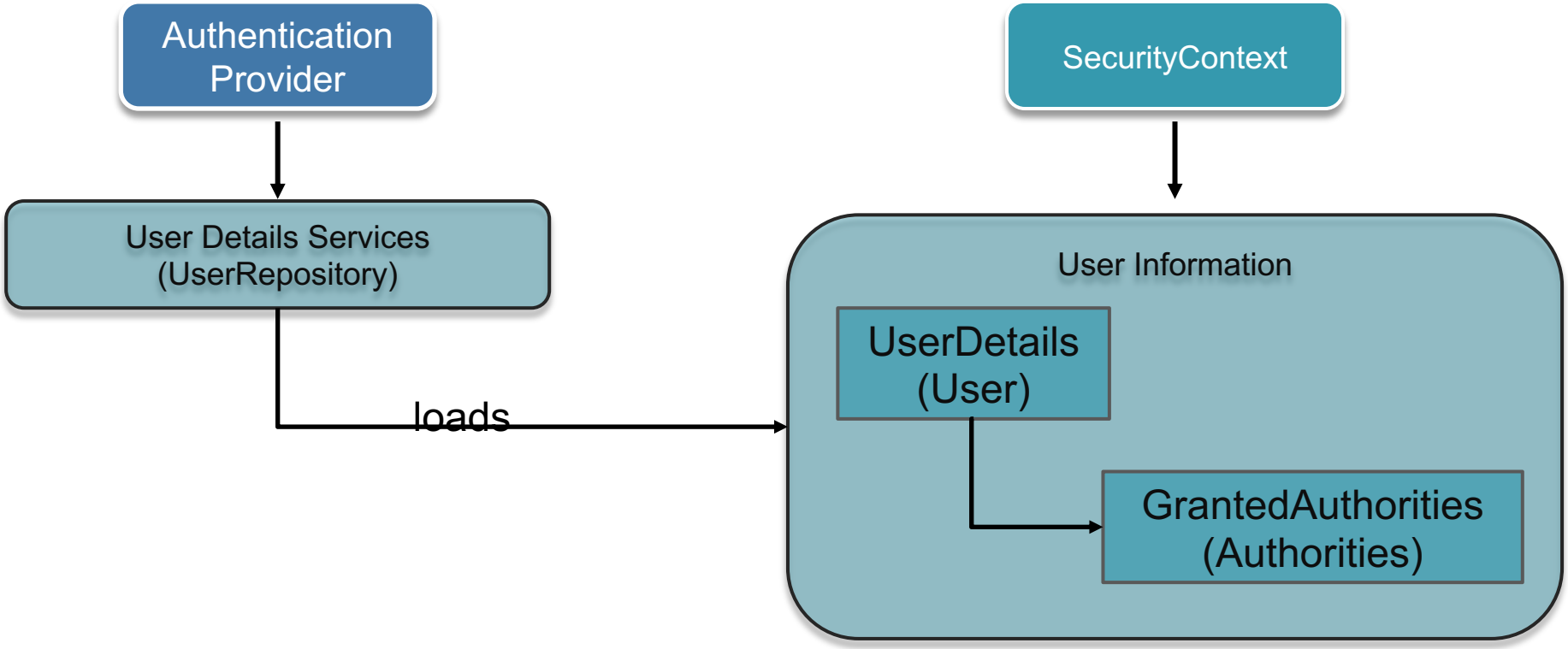
- Zugriff auf den aktuellen Benutzernamen

```
Object principal =
     SecurityContextHolder.getContext().getAuthentication().getPrincipal();

if (principal instanceof UserDetails) {
   String username = ((UserDetails)principal).getUsername();
} else {
   String username = principal.toString();
}
```

Die Architektur von Spring Security

Das UserDetails Objekt

- Die Informationen des aktuellen Benutzers können über den UserDetailsService und über die UserRepository von der DB gelesen werden. Der UserDetails Objekt enthält die GrantedAuthorities.



Spring Konfiguration

- Spring Security kann via XML und via Java konfiguriert werden. Typischerweise in Java mit Spring Boot
  - Mit `@EnableWebSecurity()` innerhalb einer Konfigurationsklasse (@Configuration) für die **Authentisierung** und **Autorisierung** auf *Stufe URL*
  - Mit `@EnableGlobalMethodSecurity()` innerhalb einer Konfigurationsklasse (@Configuration) für die **Autorisierung** auf *Stufe Methode*

```
@EnableGlobalMethodSecurity(securedEnabled=true, prePostEnabled=true)
```

Authentisierung
Die Benutzerverwaltung

- Soll ein Softwaresystem von mehreren Benutzern benutzt werden, dann muss es über eine interne oder externe Benutzerverwaltung (Identity-Store) verfügen
- In der Benutzerverwaltung müssen die Benutzer-Attribute und allenfalls die Credentials abgelegt werden.
- Identity Store können sein:
  - LDAP / ActiveDirectory
  - Externe Identity Provider (z.B. Google, GitHub, usw.),
  - Benutzer-Datenbank, Filesystem
  - usw.
- Die Authentisierung kann auch via Third-Party-Service erfolgen:
  - RSA-Server, KeyCloak
  - CAS (Central Authentication Service), usw.

Authentisierung

Die Benutzerverwaltung: Passwörter

- Spring bietet Mechanismen um Passwörter (mit hoher Güte) sicher zu speichern
  - Hash-Algorithmen, die einen Salt verwenden
  - Hashfunktionen, die sich parametrisieren lassen (Zukunft)
  - Hashfunktionen, welche gewollt langsam sind (min. 1s/Hash)
- Spring verwendet standardmässig den DelegationPasswortEncrypter
  - Ermöglicht verschiedene Hashverfahren (auch künftige)
  - Verwendet das Format {hashID}PasswortHash
  - Fehlt die {hashID} verwendet Spring den Default-PasswortEncrypter (BCrypt)

Authentisierung

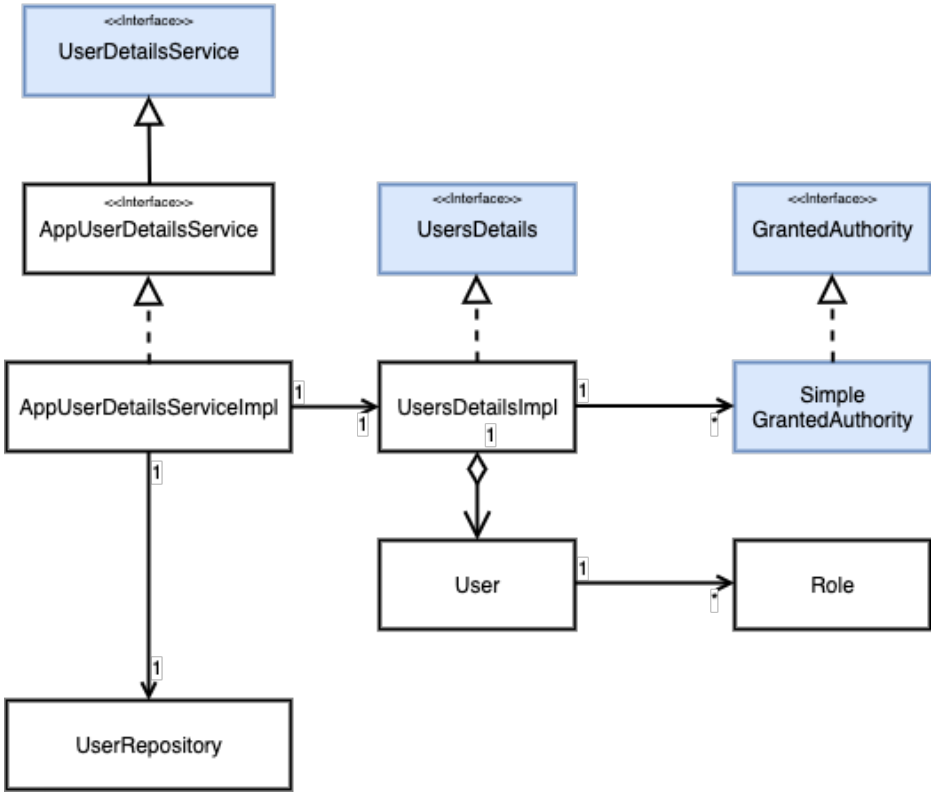
Die Benutzerverwaltung: In-Memory (Test only)


```
@Configuration
@EnableWebSecurity
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception {
        auth.inMemoryAuthentication()
                //configure user with role and password
                .withUser("user1").roles("USER").password("{noop}12345678")
                .and()
                .withUser("user2").roles("USER").password("{noop}12345678")
                .and()
                .withUser("admin").roles("ADMIN").password("{bcrypt}$2a$12$S……C9.872")
                .and()
                //with custom password encoder
                .passwordEncoder(passwordEncoder());

    }
}
```

Authentisierung

Die Benutzerverwaltung: Hello-Rest Tutorial

- Benutzer und Rollen werden über User und Role Model Klassen zur Verfügung gestellt.
- Benutzer werden über den UserDetailsService via Repository geladen
- UserDetails wird mit UserDetailsImpl umgesetzt

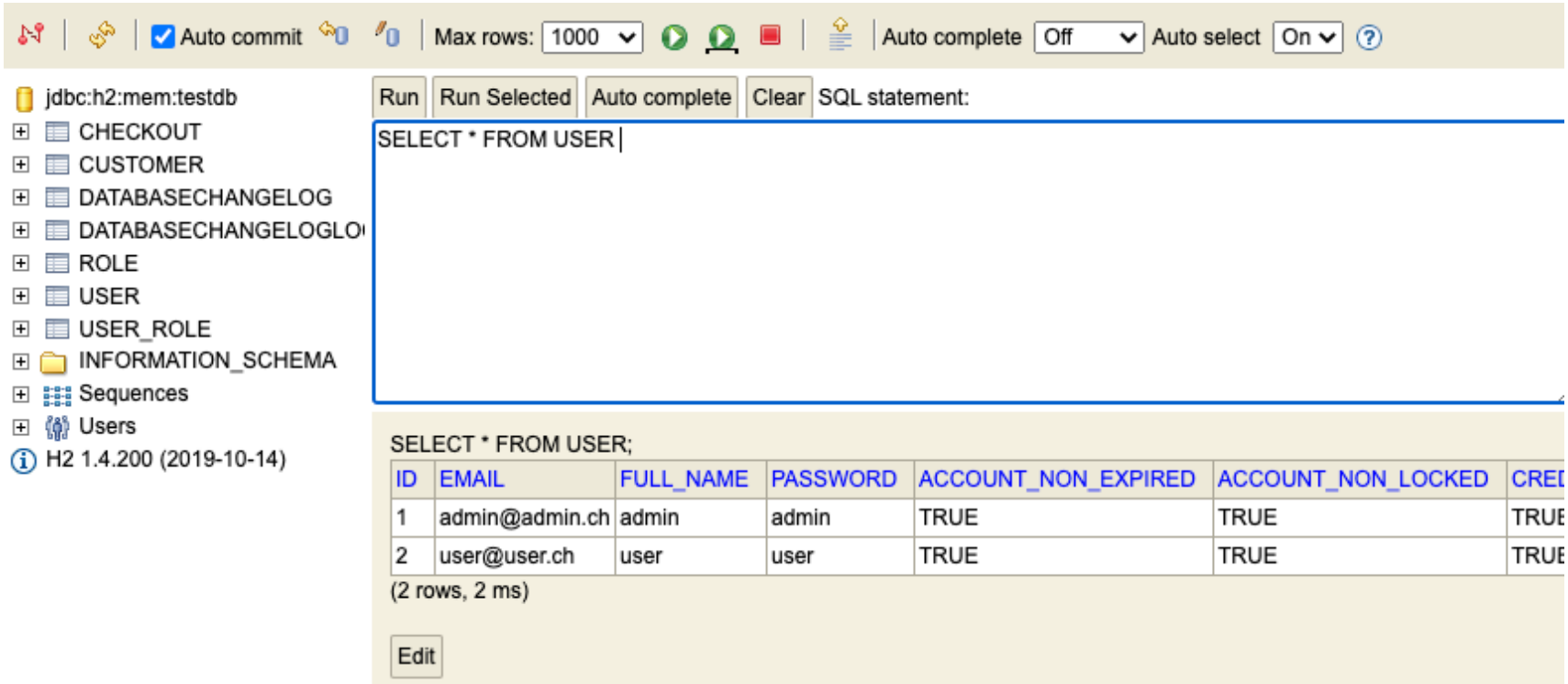


https://github.zhaw.ch/bacn/ase2-spring-boot-hellorest/blob/master/readme/security-step-1.md

Authentisierung

Die Benutzerverwaltung: Hello-Rest Tutorial

- Die Datenbank hat für die Benutzerverwaltung 3 Tabellen:
  - USER, ROLE und USER_ROLE (2 x 1:N von USER zu ROLE)



Authentisierung
Security Session-Management

- ALWAYS
  - Erstellt bei jedem Zugriff eine neue Session
- IF_REQUIRED
  - Erstellt eine neue Session, wenn diese aufgrund einer Konfiguration benötigt wird. Das ist die Default-Einstellung von Spring Security
- NEVER
  - Es wird nie eine neue Session erstellt, aber eine vorhandene Session kann gegebenenfalls genutzt werden
- STATELES
  - Es wird keine Session erstellt
- Code-Beispiel

```
http.headers().frameOptions().disable()
        .and().sessionManagement()
            .sessionCreationPolicy(SessionCreationPolicy.STATELESS);
```

Authentisierung
Zusammenfassung

- Unterscheidung in Bezug auf Backend-Varianten
  - Session-basierte Sicherheit (für Standard MVC Anwendung)
  - Sicherheitsmodell für ein zustandsloses Backend ohne Session
- Authentication Optionen
  - Basic Authentication
  - Digest Authentication
  - Zertifikats-basierte Authentication
  - Token-basierte Authentication (JWT – Json Web Token)
  - OAuth-basierte Authentication
- Integration der Autentisierung in das Spring Framework mit UserDetails, GrantedAuthorities und ggf. UserDetailsService

Autorisierung
Grundlagen

- Neben der Authentisierung muss ein Benutzer, ein System für den Zugriff auf geschützte Ressourcen autorisiert werden.
- Autorisierungen könnten auf Stufe URL oder auf Stufe Java-Methode erfolgen:
  - Bei URL wird die Autorisierung mit Hilfe von Servlet-Filtern gemacht
  - Bei Java-Methoden wird die Autorisierung mit Hilfe dynamischer Proxies(AOP) und/oder mit ACL-Listen gemacht
- Zugriffsregeln können mit verschiedenen Methoden definiert werden:
  - Durch einfache Rollendeklaration
  - Durch Spring Expression Language (SpEL) Scripts (mächtig)

Autorisierung

Grundlagen

- Die Berechtigungen können in verschiedenen Formen vergeben werden:
  - Rollen [*ROLE_*]
  - Berechtigungen [ohne Präfix]
  - Access Control List (ACL)
    [*READ, WRITE, CREATE, DELETE, ADMINISTRATION*]

Autorisierung
URL-basierte Zugriffskontrolle: HttpSecurity

- URL-basierte Zugriffsverwaltung kann mit dem Fluent-API der Klasse `WebSecurityConfigAdapter` realisiert werden.
- Dabei muss die Methode `configure(HttpSecurity http)`, welche das `HTTPSecurity` Objekt erhält, überschrieben werden:
  - Mittels `authorizeRequests()` können Zugriffsregeln erstellt werden
  - Mittels `mvcMatchers()` oder `antMatchers()` kann definiert werden, für welche **URLs** die Konfiguration gilt
  - Optional kann der Zugriff eine HTTP-Methode (GET, POST usw.) und auf Rollen eingeschränkt werden
- Findet Spring Security einen Treffer, wird nicht weiter gesucht, also muss die Konfiguration von fein nach grob erfolgen

Autorisierung
URL-basierte Zugriffskontrolle: HttpSecurity

- Die Default Konfiguration von Spring Boot kann überschrieben werden

```
@Configuration
@EnableWebSecurity
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
                .authorizeRequests()
                   .mvcMatchers("/","/home").permitAll()
                   .anyRequest().authenticated()
                   .and()
                .formLogin()
                   .loginPage("/login")
                   .permitAll()
                   .and()
                .logout()
                   .permitAll();
    }
}
```

Autorisierung
URL-basierte Zugriffskontrolle: HttpSecurity

- Mehrere Bedingungen verknüpfen
- Überprüfung in der programmierten Reihenfolge
- Spezifische Überprüfungen müssen zuerst aufgeführt werden

```
@Override
protected void configure(HttpSecurity http)throws Exception{
    http
            .authorizeRequests()
            .mvcMatchers("/signup","/about").permitAll()
            .mvcMatchers("/employees/edit*").hasRole("ADMIN")
            .mvcMatchers("/employees/**").hasAnyRole("USER","ADMIN")
            .anyRequest().authenticated();
}
```

Autorisierung
Security auslassen: WebSecurity

- Statische Resourcen müssen nicht abgesichert werden
- Bei Verwendung von `HttpSecurity` und `PermitAll()` wird die gesamte SecurityFilterChain durchlaufen (Performance)
- Security Überprüfungen werden nicht beachtet

```
@Override
public void configure(WebSecurity web) throws Exception {
    web.ignoring()
            .antMatchers(HttpMethod.OPTIONS, "/**")
            .antMatchers("/h2-console/**")
            .antMatchers("/swagger-ui.html")
            .antMatchers("/swagger-ui/**")
            .antMatchers("/v3/**");
}
```

Autorisierung
Method based Access-Control

- Mit **`@EnableGlobalMethodSecurity`** auf einer Konfigurationsklasse kann die Autorisierung auf Stufe Methode aktiviert werden
- Es werden folgende Annotationen unterstützt:
  - JSR250-Annotationen (siehe Servlet Security)
  - @Secured (ursprüngliche Spring-Annotation)
  - @PreAutorize/@Postautorized (neue Spring-Annotationen)
- Aktiviert werden die Annotationen mittels entsprechender Attribute:

```
@EnableGlobalMethodSecurity(jsr250Enabled=true,
securedEnabled=true, prePostEnabled=true)
```

Autorisierung

Method based Access-Control: bisher

- Methodensicherheit kann in einem **@Controller** oder einem **@Service** verwendet werden

```
public interface MyService {
    @Secured("IS_AUTHENTICATED_ANONYMOUSLY")
    public Post readPost(Long id);

    @Secured("ROLE_POST_USER")
    public void doPost(Post post);

    @Secured({"ROLE_OPERATOR", "ROLE_USER"})
    public void doAdmin(Post post);
}
```

Autorisierung

Method based Access-Control: neu

```
public interface MyService {

    @PreAuthorize("isAnonymous()")
    public Post readPost(Long id);

    @PreAuthorize("hasAuthority('POST_USER')")
    public void doPost(Post post);

    @PreAuthorize("hasROLE('ROLE_POST_USER')")
    public void doPost(Post post);

    @PreAuthorize(“#post.name == authentication.name“)
    public String updatePost(Post post);

}
```

Autorisierung

Method based Access-Control: Spring Data Rest

- Zugriffkontrolle auf Stufe Methode am Beispiel einer Repository

```
@PreAuthorize("hasRole('USER')")
public interface CustomerRepository extends CrudRepository<Customer, Long> {

    @PreAuthorize("hasRole('ADMIN')")
    @Override
    Customer save(Customer s);

    @PreAuthorize("hasRole('ADMIN')")
    @Override
    void delete(Long id);
}
```

REST Security mit JWT Token

X-Auth oder Authorization: Bearer

- X-auth
  - Benutzername und Passwort werden zum Server übertragen
  - Der Server erzeugt ein Token und gibt es dem Client zurück
  - Die nachfolgenden Aufrufe werden mit dem Token durchgeführt

- Varianten:
  - Im Authorization-Header als Bearer-Token:
    ```
    Authorization: Bearer  eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...
    ```
  - Im Cookie-Header:
    ```
    Cookie: token=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...
    ```

REST Security mit JWT Token

JWT - Json Web Token - Einführung

- JWT – Json Web Token (https://jwt.io/ https://tools.ietf.org/html/rfc7519)
- Offener Standard für die Übertragung von Sicherheitsinformationen zwischen zwei Parteien
  - JSON Objekt
  - Beinhaltet alle notwendigen Informationen
- Hauptsächlich verwendet in Web-Anwendungen
  - Kann als Teil der URL (Query String), Form-Body-Parameter, Cookie oder HTTP Header (x-access-token) verwendet werden
  - Gut geeignet für SSO (Single Sign On), gleiches Token für mehrere Web-Sites

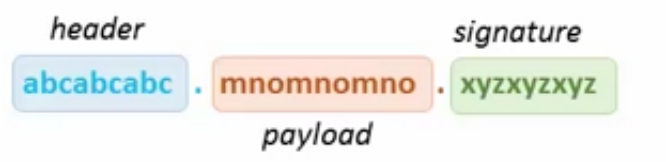
REST Security mit JWT Token

JWT Struktur

Zürcher Hochschule für Angewandte Wissenschaften

School of Engineering

- Drei Bereiche, abgetrennt mit Punkten
  - Header, Payload und Signature
  - Alle sind base64 encoded (nicht encrypted)
- Header – enthält normalerweise 2 Teile
  - Typ: sollte JWT sein
  - Alg: Hashing Algorithmus wie HS256, RS512, etc.
- Payload
  - die Information die übertragen werden soll
  - Bezieht sich auf das Token selber
  - Die Information ist JSON (key:value)
- Signatur
  - Hash von Header und Payload mittels Secret
- JWT = Header + Payload + Signature

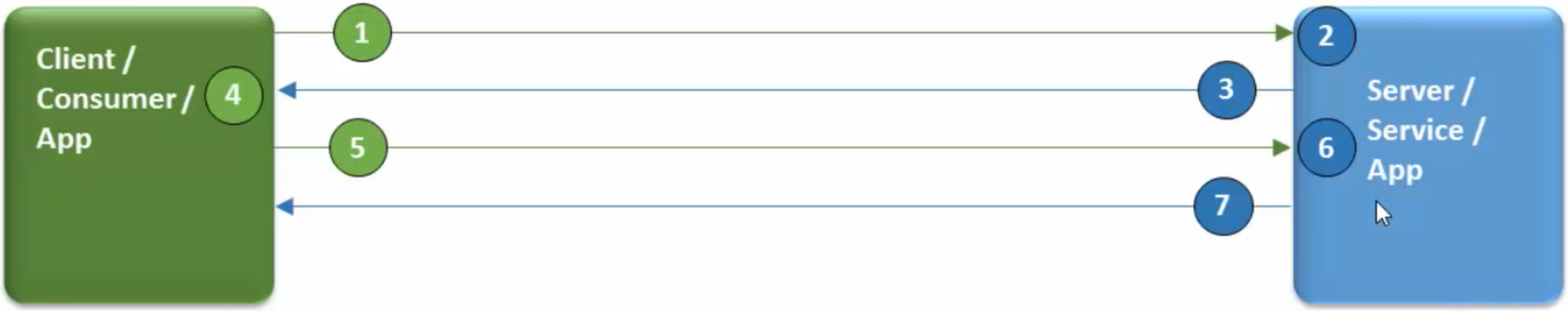



```
var s = base64Encode(header)
        + "."
        + base64Encode(payload);
var signature = hashAlgHs256(s, 'secret');
```

```
var jwt = s + "." + base64Encode(signature)
```

REST Security mit JWT Token

JWT – Client - Server



1. Request token (by providing auth. Credentials)

2. - Check credentials,
   - Create token (JWT)

3. Send JWT to client (or error if not auth.)

4. - Verify token (optional)
   - Extract information for app usage (optional)
   - Persist token (at client-side)

5. Use same (persisted) token for subsequent calls to server

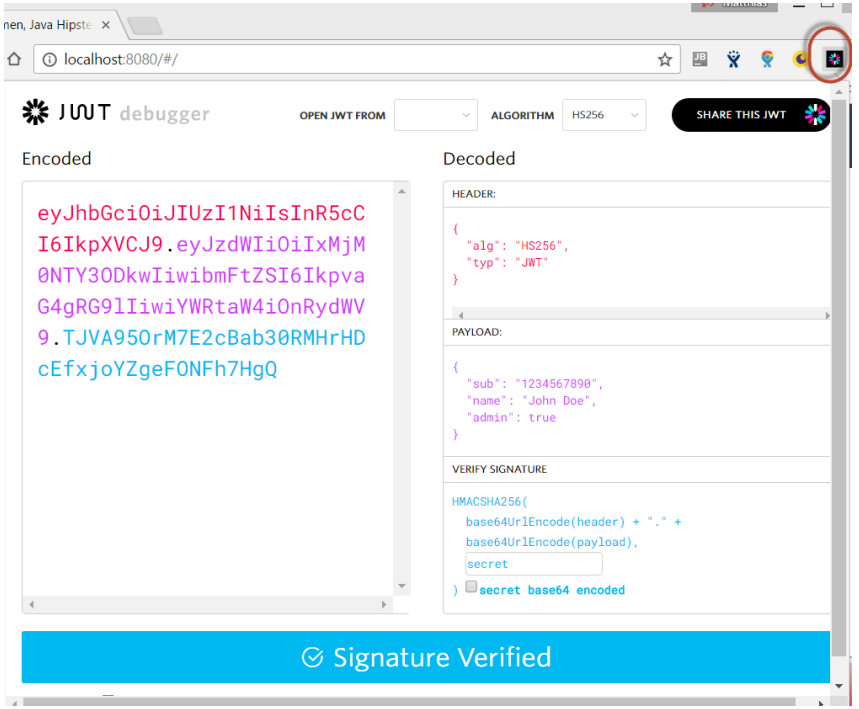
6. Verify token for all subsequent requests

7. If valid token, respond with result else, return error

REST Security mit JWT Token

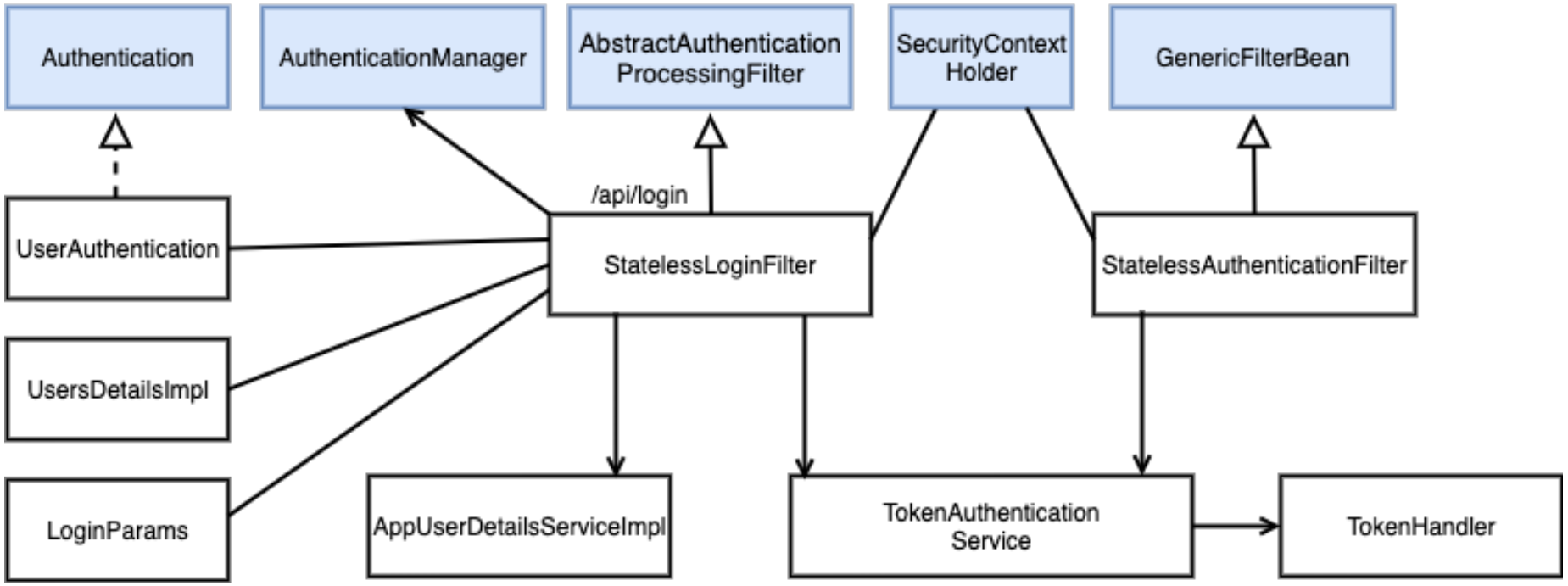
JWT – Online Tools

- Chrome – JWT Erweiterung
- Erstellen von JWT
  - http://jwtbuilder.jamiekurtz.com/
  - http://kjur.github.io/jsjws/tool_jwt.html
- Verifikation von JWT
  - https://jwt.io/
- Base64 Encode / Decode
  - https://www.base64encode.org/
  - https://www.base64decode.org/



Tutorial HelloRest
SecurityConfiguration

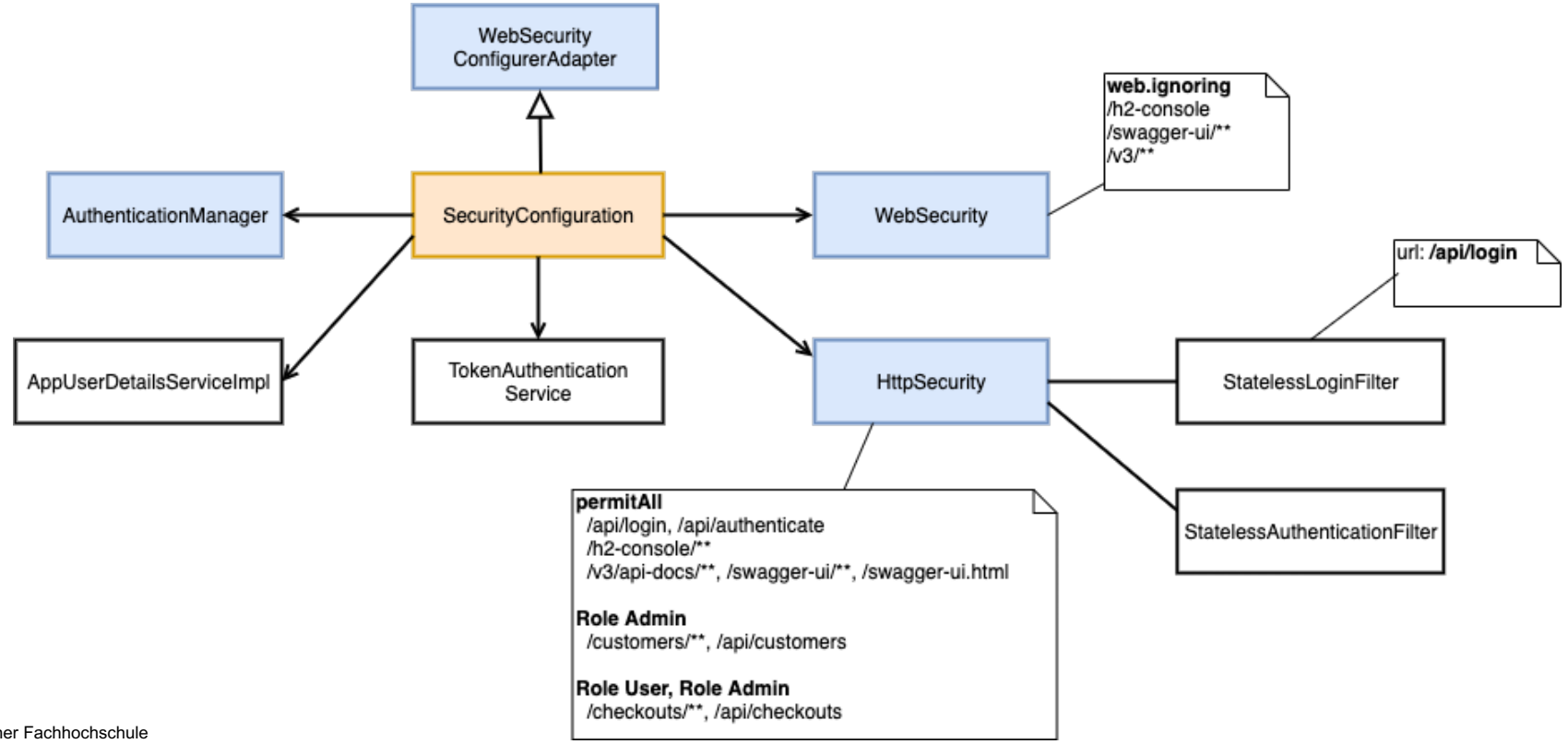
- StatelessLoginFilter: Überprüft eMail und Passwort
  - Erstellen ein Token mittels TokenAuthenticationService und TokenHandler
- StatelessAuthenticationFilter: prüft Signatur des Tokens und Gültigkeit



Tutorial HelloRest

SecurityConfiguration

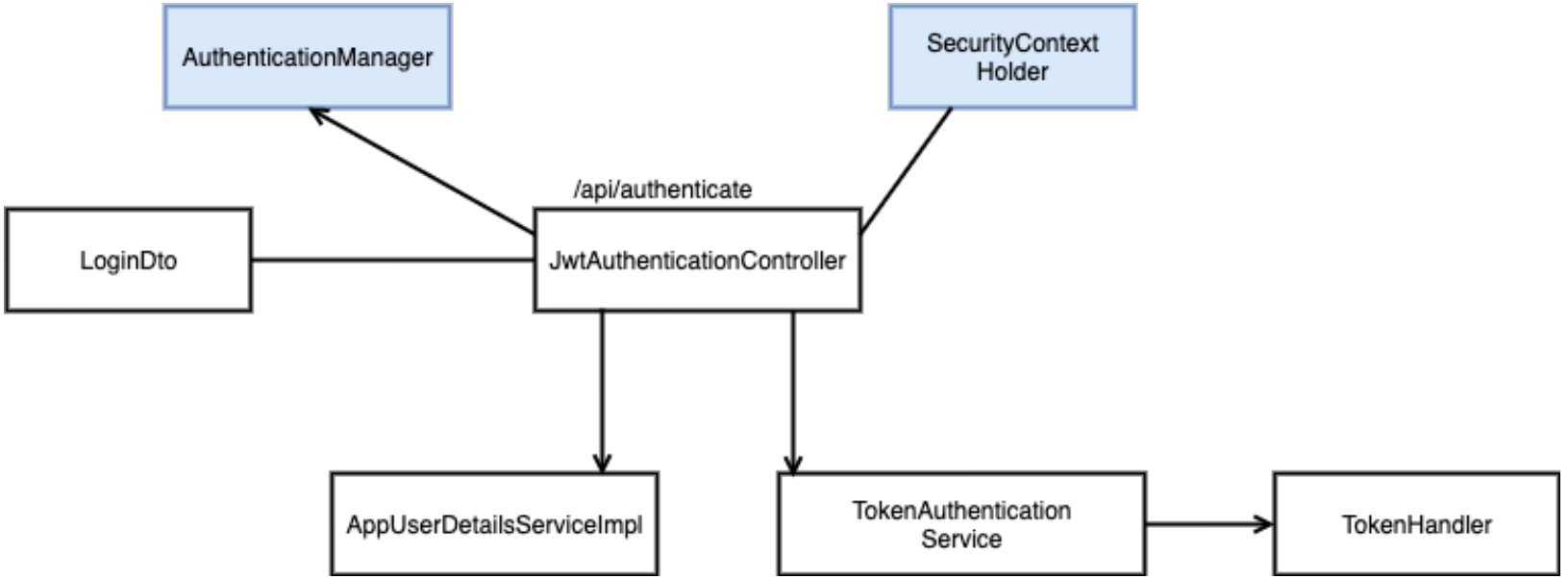
- Konfiguriert WebSecurity und HttpSecurity
- Instanziert StatelessLoginFilter und StatelessAuthenticationFilter
- Beinhaltet CORS und PasswordEncoder Bean



Tutorial HelloRest

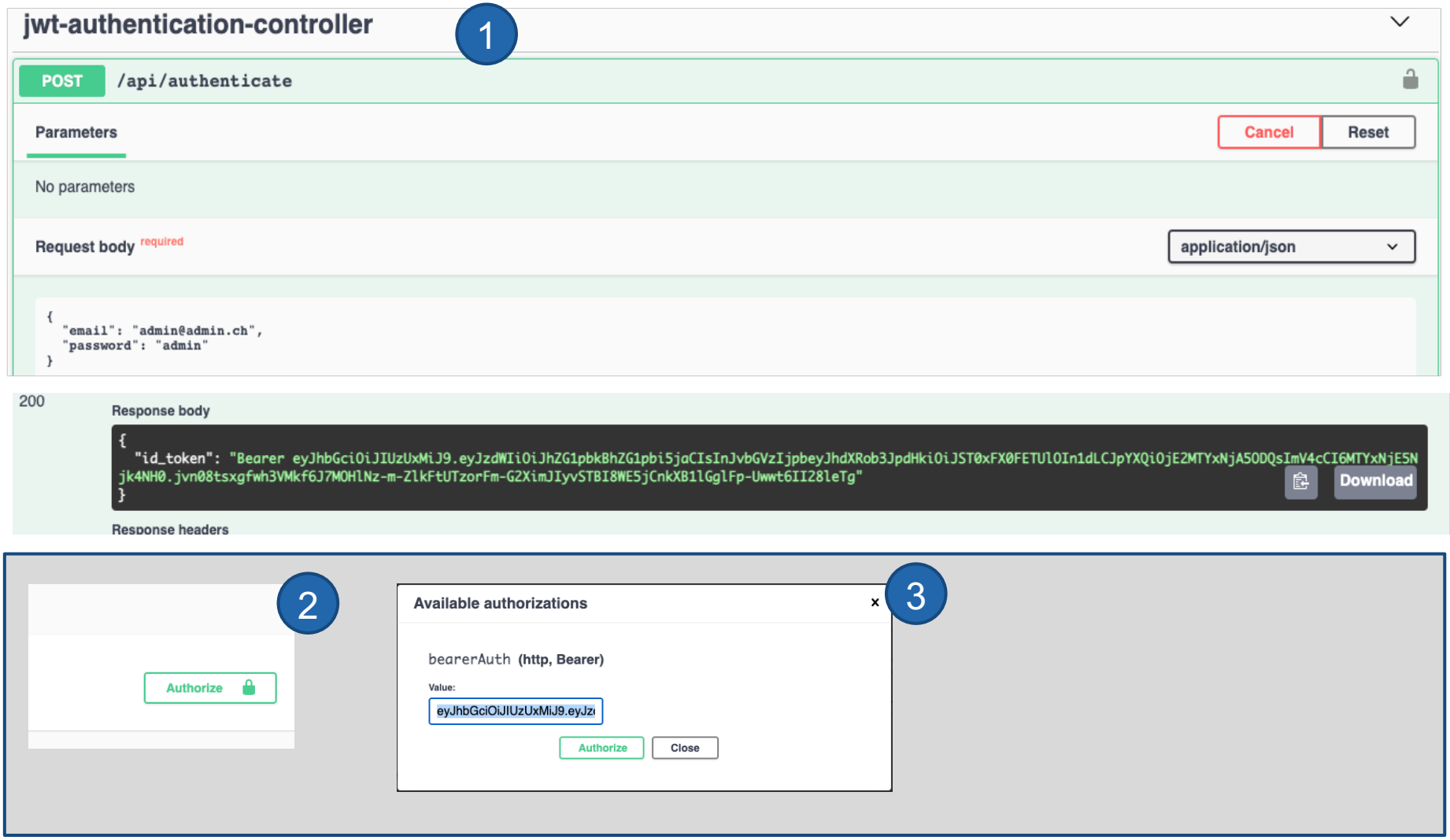
SecurityConfiguration

- Zusätzliche Authentication mittels RestController
- Wird benötigt für OpenApi Login (Token wird im Body ausgeliefert)



Tutorial HelloRest

Testen mit CURL oder OpenApi



Tutorial HelloRest
Testen mit CURL oder OpenApi

- Login

```
curl -X 'POST' \
 'http://localhost:8080/api/authenticate' \
 -H 'accept: application/hal+json' \
 -H 'Content-Type: application/json' \
 -d '{ "email": "admin@admin.ch", "password": "admin" }'
```

- Use the Token

```
curl -X 'GET' \
 'http://localhost:8080/checkouts' \
 -H 'accept: application/hal+json' \
 -H 'Authorization: Bearer <this token is not valid
anymore>eyJhbGciOiJIUzUxMiJ9.eyJzdWIiOiJhZG1pbkBhZG1pbi5jaCIsInJvbGVzIj
pbeyJhdXRob3JpdHkiOiJST0xFX0FETUlOIn1dLCJpYXQiOjE2MTYxNjA5ODQsImV4cCI6M
TYxNjE5Njk4NH0.jvn08tsxgfwh3VMkf6J7MOHlNz-m-ZlkFtUTzorFm-
G2XimJIyvSTBI8WE5jCnkXB1lGglFp-Uwwt6II28leTg'
```

Zusammenfassung

- Architektur Spring Security
- Authentisierung
- Autorisierung
- Rest Security mit JWT