

Modul 223

Multiuser - Apps



Unterrichtsziel heute

- ~~einrichten Backend-Projektumgebung~~
- nutzen und konfigurieren Spring-Security
- implementieren FilterChain
- implementieren RBAC – Datenmodell mit JPA
- erklären Unterschiede und Abläufe bei sessionbasierter und Token-basierter Auth²

Wie war das nochmal...

- Was war nochmal das Besondere an Multiuser-Apps?
- Worauf muss besonders geachtet werden?
- Was hat es mit RBAC auf sich?
- Wie werden HTTP-Requests in einer Springboot-Webapp verarbeitet?
- Wie funktioniert die persistente Datenspeicherung mit der Java Persistence API (JPA)?

Backend-Projekt einrichten

- Template mit Spring-Initializr (<https://start.spring.io/>) generieren



Project

☐ Gradle - Groovy
 ☐ Gradle - Kotlin
 ☒ **Java**
☐ Kotlin
 ☐ Groovy

☒ **Maven**

Spring Boot

☐ 3.4.0 (SNAPSHOT)
 ☐ 3.4.0 (RC1)
 ☐ 3.3.6 (SNAPSHOT)
 ☒ **3.3.5**

☐ 3.2.12 (SNAPSHOT)
 ☐ 3.2.11

Project Metadata

Group

Artifact

Name

Description

Package name

Packaging ☒ **Jar** ☐ War

Java ☐ 23 ☒ **21** ☐ 17

Dependencies

ADD DEPENDENCIES... CTRL + B

Spring Web **WEB**

Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

Rest Repositories **WEB**

Exposing Spring Data repositories over REST via Spring Data REST.

Spring Data JPA **SQL**

Persist data in SQL stores with Java Persistence API using Spring Data and Hibernate.

MySQL Driver **SQL**

MySQL JDBC driver.

Spring Security **SECURITY**

Highly customizable authentication and access-control framework for Spring applications.

Lombok **DEVELOPER TOOLS**

Java annotation library which helps to reduce boilerplate code.

Validation **I/O**

Bean Validation with Hibernate validator.

Backend-Projekt einrichten

- Template entpacken
- in pom.xml: JPA Abschnitt vorerst auskommentieren
- in IDE des Vertrauens ausführen

Spring Security Intro

Please sign in

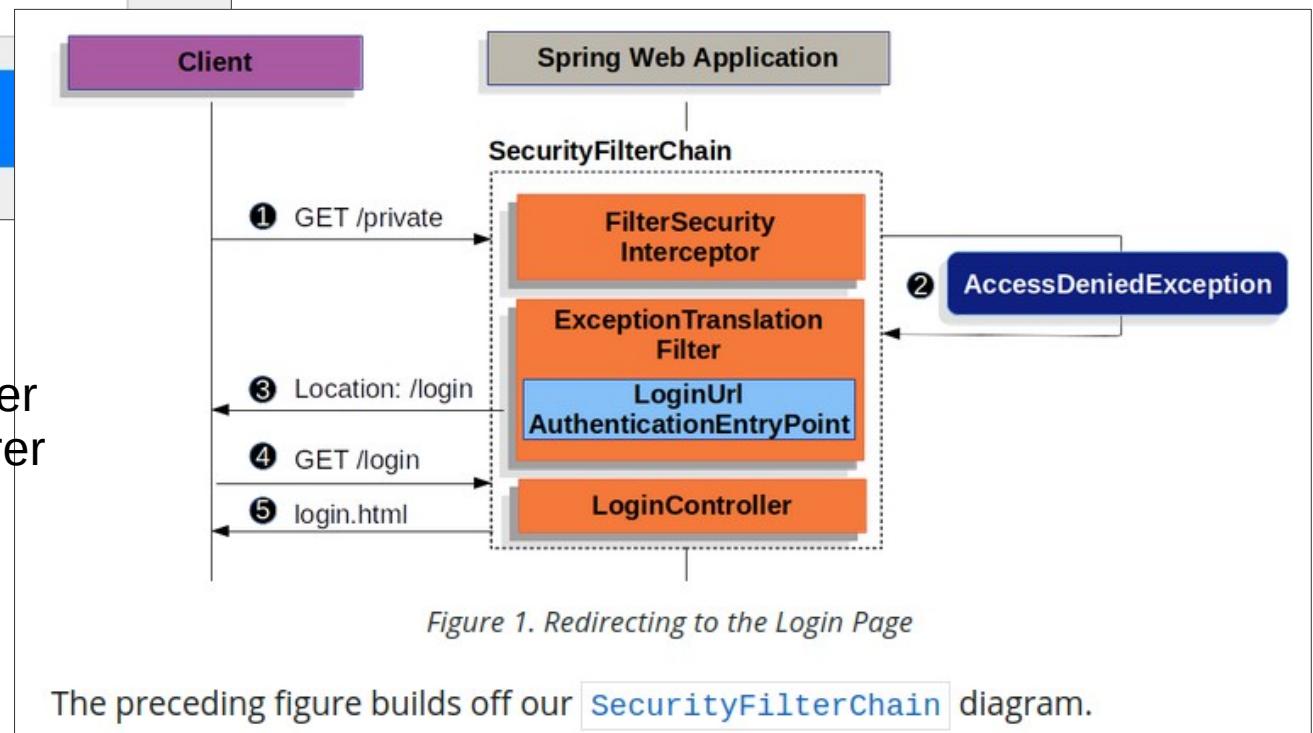
Username

Password

Sign in

Spring Security ist eine **zusätzlich Schicht zwischen** Client und Spring App, die **jeden Request** anhand der **Security Regeln prüft**. Security Regeln werden als **Filter** definiert.

Die Filter bilden eine Kette (**Filter Chain**). Jeder Request durchläuft die Filter der Kette entsprechend ihrer Reihenfolge. Dabei **prüft jeder Filter**, ob dem Request **Zugriff** auf die Anwendung gewährt wird oder nicht.





Spring Security Basiskonfiguration

- alles ist geschützt
- jeder Zugriff mit Benutzer+PWD im Auth-Header
- Standard-"user" mit generiertem Passwort (siehe StartLog), wenn nichts anderes definiert ist
- z.B. eigener Standard-"user" konfigurieren in **application.properties**:
`spring.security.user.name=admin`
`spring.security.user.password=geheim`
oder in **SecurityConfig** (nächste Folie) oder mit **eigenen Klassen** (später)
- Erklärungen siehe

<https://agile-coding.blogspot.com/2020/12/spring-security-starter.html>

Benutzer mit Rolle erstellen

in SecurityConfiguration.java

```
@Configuration
public class SecurityConfiguration {
    @Bean
    public UserDetailsService users(@Autowired PasswordEncoder pwEnc){
        UserDetails user = User.builder()
            .username("user")
            .password(pwEnc.encode("top"))
            .roles("USER")
            .build();
        UserDetails admin = User.builder()
            .username("admin")
            .password(pwEnc.encode("secret"))
            .roles("USER", "ADMIN")
            .build();
        return new InMemoryUserDetailsManager(user, admin);
    }

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


Beispiel SecurityFilterChain

mit öffentlichen und geschützten Bereich(en)

<https://docs.spring.io/spring-security/reference/servlet/authorization/authorize-http-requests.html>

```
@Configuration
@EnableWebSecurity
public class SecurityConfiguration {
    /* ... */
    private static final String[] EVERYONE = {"/public"};

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http)
        throws Exception {

        http.csrf(csrf -> csrf.disable()) //disable Cross-Site Request Forgery (CSRF) prevention
        .cors(Customizer.withDefaults()) //configure CORS: Cross Origin Request Sharing
        .authorizeHttpRequests(auth -> {
            auth.requestMatchers(HttpMethod.POST, "/items").hasRole("ADMIN");
            auth.requestMatchers(EVERYONE).permitAll()
                .anyRequest().authenticated(); } )

        .formLogin(Customizer.withDefaults()) //für Login-Form im Browser
        .httpBasic(Customizer.withDefaults()); // für CURL, Postman, Insomnia
        return http.build();
    }
}
```

- der hier gezeigte Code entspricht dem Spring Security 7.0 Coding Standard
- Erklärungen siehe <https://agile-coding.blogspot.com/2022/09/spring-security-roles.html>
- CSRF: <https://www.baeldung.com/spring-security-csrf>
- CORS: <https://www.baeldung.com/spring-cors>

Test mit Insomnia oder Postman

The image displays two screenshots of API testing tools. The top screenshot shows the Insomnia interface with a GET request to `http://localhost:8080/private` configured with Basic authentication (username: user, password: top). The bottom screenshot shows the Postman interface with a POST request to `http://localhost:8080/items` configured with Basic authentication (username: admin, password: secret). Both interfaces show the 'Auth' tab selected, and the bottom screenshot also shows the 'Body' tab with the response `1 Admin Area`.

Insomnia Interface (Top):

- Environment: M223-Demo
- Request: GET `http://localhost:8080/private`
- Status: 200 OK, 77 ms
- Auth: Basic (Enabled)
- Username: user, Password: top
- Preview: 1 Geheim

Postman Interface (Bottom):

- Collection: POST ADMIN
- Request: POST `http://localhost:8080/items`
- Status: 200 OK, 85 ms, 433 B
- Auth: Basic Auth (Enabled)
- Username: admin, Password: secret
- Body: 1 Admin Area

Alles klar?



Persistente RBAC + Datenmodel mit JPA



- Annotationen
 - @Entity
 - @Id
 - @ManyToOne
 - @ManyToMany
- JPA-Dependency aktivieren
- MySQL-Zugangsdaten in application.properties konfigurieren:

```

spring.jpa.hibernate.ddl-auto=update
spring.datasource.url=jdbc:mysql://127.0.0.1:3306/wiss_quiz
spring.datasource.username=root
spring.datasource.password=example
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
  
```

```

@Entity
@Setter @Getter @NoArgsConstructor
public class User {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    @NotBlank
    private String username;
    @NotBlank
    private String email;
    @NotBlank
    private String password;

    public User(String name, String email, String password) {
        this.username = name;
        this.email = email;
        this.password = password;
    }

    @ManyToMany(fetch = FetchType.LAZY) // N:M Mapping
    private Set<Role> roles = new HashSet<>();
}
  
```

Implementierung: ERole und Role

```
public enum ERole {  
    ROLE_USER,  
    ROLE_MODERATOR,  
    ROLE_ADMIN  
}
```

```
@Entity @Getter @Setter @NoArgsConstructor  
public class Role {  
    @Id  
    @GeneratedValue(strategy = GenerationType.IDENTITY)  
    private Integer id;  
  
    @Enumerated(EnumType.STRING)  
    @Column(length = 20)  
    private ERole name;  
  
    public Role(ERole name) {  
        this.name = name;  
    }  
    public String toString(){  
        return name.toString();  
    }  
}
```

Session-basiert vs. Token-basierte AUTH²

- <https://youtu.be/UBUNrFtufWo>
- Erarbeitet Euch anhand dieser Quelle:
<https://www.geeksforgeeks.org/session-vs-token-based-authentication/>
ein grundlegendes Verständnis für die Abläufe, sowie die Vor- und Nachteile der beiden Verfahren
- siehe Sidequest 3B
- Zeit: 30 Minuten