Summary Till Now

👉 Spring Security with Database Authentication: Complete Summary

➢ The Security Configuration

The heart of our Spring Security setup is the SecurityConfig class:

Example:

```java
package com.telusko.springsecdemo.config;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.authentication.AuthenticationProvider;
import org.springframework.security.authentication.dao.DaoAuthenticationProvider;
import org.springframework.security.config.Customizer;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.config.http.SessionCreationPolicy;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.crypto.password.NoOpPasswordEncoder;
import org.springframework.security.web.SecurityFilterChain;

@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Autowired
    private UserDetailsService userDetailsService;

    @Bean
    public AuthenticationProvider authProvider() {
        DaoAuthenticationProvider provider = new DaoAuthenticationProvider();
        provider.setUserDetailsService(userDetailsService);
        provider.setPasswordEncoder(NoOpPasswordEncoder.getInstance());
        return provider;
    }

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        http.csrf(customizer -> customizer.disable())
            .authorizeHttpRequests(request -> request.anyRequest().authenticated())
            .httpBasic(Customizer.withDefaults())
            .sessionManagement(session -> session.sessionCreationPolicy(SessionCreationPolicy.STATELESS));

        return http.build();
    }
}
```

- We use @Configuration and @EnableWebSecurity to mark this as a Spring Security configuration
- We create a custom AuthenticationProvider bean that uses database authentication
- We configure security settings like CSRF protection, authentication requirements, and session management

➢ **The Custom UserDetailsService**

To connect with our database, we implement the UserDetailsService interface:

Example:

```java
package com.telusko.springsecdemo.service;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.core.userdetails.UsernameNotFoundException;
import org.springframework.stereotype.Service;

import com.telusko.springsecdemo.dao.UserRepo;
import com.telusko.springsecdemo.model.User;
import com.telusko.springsecdemo.model.UserPrincipal;

@Service
public class MyUserDetailsService implements UserDetailsService {

    @Autowired
    private UserRepo repo;

    @Override
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
        User user = repo.findByUsername(username);

        if (user == null) {
            System.out.println("User 404");
            throw new UsernameNotFoundException("User 404");
        }
        return new UserPrincipal(user);
    }
}
```

- The @Service annotation makes this a Spring-managed bean that can be autowired
- We inject the UserRepo to access our database
- The loadUserByUsername() method fetches a user by username from the database
- If the user exists, we wrap it in a UserPrincipal object and return it
- If the user doesn't exist, we throw a UsernameNotFoundException

➢ **The Repository Layer**

We create a UserRepo interface to interact with our database:

Example:

```java
package com.telusko.springsecdemo.dao;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

import com.telusko.springsecdemo.model.User;

@Repository
public interface UserRepo extends JpaRepository<User, Integer> {
    User findByUsername(String username);
}
```

- It extends JpaRepository to get basic CRUD operations for free
- We add a custom findByUsername() method to search by username
- Spring Data JPA automatically implements this method based on the name

➢ **The User Entity**

Our User class maps to the database table:

Example:

```java
package com.telusko.springsecdemo.model;

import jakarta.persistence.Entity;
import jakarta.persistence.Id;
import jakarta.persistence.Table;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

@Data
@Table(name = "users")
@Entity
public class User {

    @Id
    private int id;
    private String username;
    private String password;
}
```

- @Entity and @Table annotations mark this as a JPA entity mapping to the "users" table
- It has fields corresponding to the database columns: id, username, and password
- @Data from Lombok generates getters, setters, equals, hashCode, and toString methods

➢ **The UserPrincipal Class**

This class adapts our database User to Spring Security's UserDetails interface:

Example:

```java
package com.telusko.springsecdemo.model;

import java.util.Collection;
import java.util.Collections;

import org.springframework.security.core.GrantedAuthority;
import org.springframework.security.core.authority.SimpleGrantedAuthority;
import org.springframework.security.core.userdetails.UserDetails;

public class UserPrincipal implements UserDetails {

    private static final long serialVersionUID = 1L;

    private User user;

    public UserPrincipal(User user) {
        this.user = user;
    }

    @Override
    public Collection<? extends GrantedAuthority> getAuthorities() {
        return Collections.singleton(new SimpleGrantedAuthority("USER"));
    }

    @Override
    public String getPassword() {
        return user.getPassword();
    }

    @Override
    public String getUsername() {
        return user.getUsername();
    }

    @Override
    public boolean isAccountNonExpired() {
        return true;
    }

    @Override
    public boolean isAccountNonLocked() {
        return true;
    }

    @Override
    public boolean isCredentialsNonExpired() {
        return true;
    }

    @Override
    public boolean isEnabled() {
        return true;
    }
}
```

- It implements the UserDetails interface required by Spring Security
- It wraps our database User object to access username and password
- The getAuthorities() method returns the user's roles/permissions (here, a simple "USER" role)
- The account status methods all return true, indicating the account is valid

➢ **Database Configuration**

In application.properties, we configure the database connection:

Example:

```
# PostgreSQL Database Configuration
spring.datasource.url=jdbc:postgresql://localhost:5432/telusko
spring.datasource.username=postgres
spring.datasource.password=0000
spring.datasource.driver-class-name=org.postgresql.Driver

server.servlet.session.cookie.same-site=strict
```

- These properties tell Spring how to connect to our PostgreSQL database
- The cookie setting adds extra security for the session cookie

👉 **Explanation**

➢ When a user tries to access a protected resource, Spring Security intercepts the request

➢ Spring Security calls our custom AuthenticationProvider

➢ The provider uses MyUserDetailsService to load the user from the database

➢ MyUserDetailsService uses UserRepo to fetch the user and wraps it in a UserPrincipal

➢ The provider compares the submitted password with the one from the database

➢ If they match, authentication succeeds and the user can access the resource

➢ If they don't match, or the user doesn't exist, authentication fails