SPRING SECURITY & SPRING BOOT WEB APPLICATION

Instructor:

Learning Goals

- **After the session, attendees will be able to:**

Understand Spring Security and implementing with Spring boot

Agenda

1. Introduction: Spring Framework vs. Spring Boot vs. Spring Security
2. Spring Security Fundamentals I
3. Spring Security Configuration
4. Practice: Impl Security
5. Password Handling with Spring Security

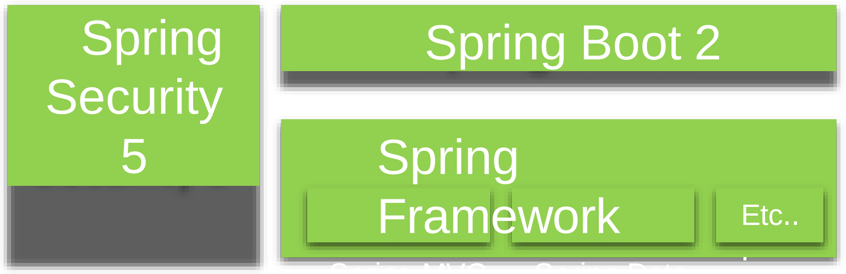
Section 1

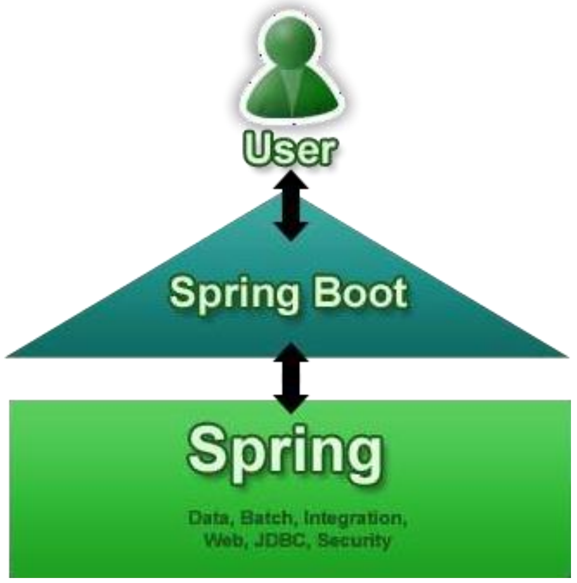
INTRODUCTION

Introduction

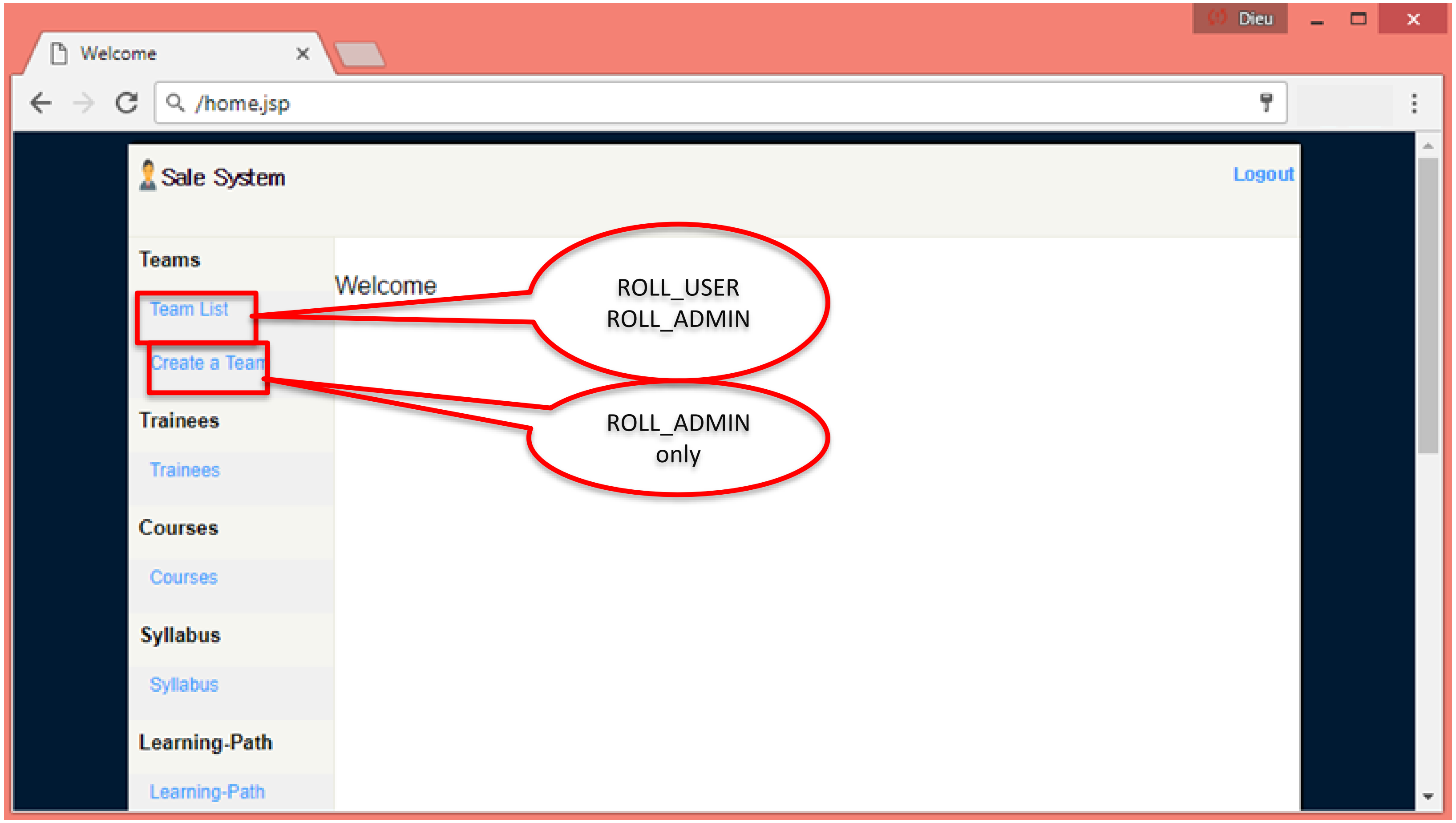
❖ *Spring Framework vs. Spring Boot vs. Spring Security*

- ✓ **Spring Framework** is a Java platform that provides comprehensive infrastructure support for developing Java applications.
- ✓ **Spring Boot** is based on the **Spring Framework**, providing auto-configuration features to your Spring applications and is designed to get you up and running as quickly as possible.
- ✓ **Spring Security** provides comprehensive security services for Java EE- based software applications. There is a particular emphasis on supporting projects built using the **Spring Framework**.





Introduction



Introduction

- **Spring security** is another major module in spring distribution and is supported only for applications developed using JDK 1.5 or higher.
- **Spring Security** is a framework that focuses on providing both **authentication** and **authorization** to Java EE-based enterprise software applications.
- **Spring security** has been divided into **multiple jars** and you should include them as your application need. Only the core module available in **spring-security-core.jar** is mandatory.

Introduction

- **spring-security-core**

  It contains core authentication and access-contol classes and interfaces

- **spring-security-web**

  It contains filters and related web-security infrastructure code. It also enable URL based security which we are going to use in this demo.

- **spring-security-config**

  It contains the security namespace parsing code. You need it if you are using the Spring Security XML file for configuration.

- **spring-security-taglibs**

  It provides basic support for accessing security information and applying security constraints in JSPs.

Authentication and Authorization

- It also provides authentication at **view level** and **method level**.
- It can also provide you with a login page!
- Here are some things that it provides:
  - ✓ Provide capabilities for **login** and **logout**
  - ✓ Control access to a **link based on the role of the user**.
  - ✓ Provide the ability to **hide certain portion of a page** if a user does not have appropriate privileges.
  - ✓ Link to a database for authentication

What is Spring Security?

- **Spring Security** is a **framework** that focuses on providing both **authentication** and **authorization** (or “access-control”) to Java web application and SOAP/RESTful web services.
  - ✓ **Authentication** is the process of **knowing** and **identifying** the user that wants to access.
  - ✓ **Authorization** is the process to **allow authority to perform actions** in the application.
- **Spring Security Features**
  - ✓ LDAP (Lightweight Directory Access Protocol)
  - ✓ Single sign-on
  - ✓ JAAS (Java Authentication and Authorization Service) LoginModule
  - ✓ Basic Access Authentication
  - ✓ Digest Access Authentication
  - ✓ Remember-me
  - ✓ Web Form Authentication
  - ✓ Authorization
  - ✓ Software Localization
  - ✓ HTTP Authorization

Spring Security Fundamentals I

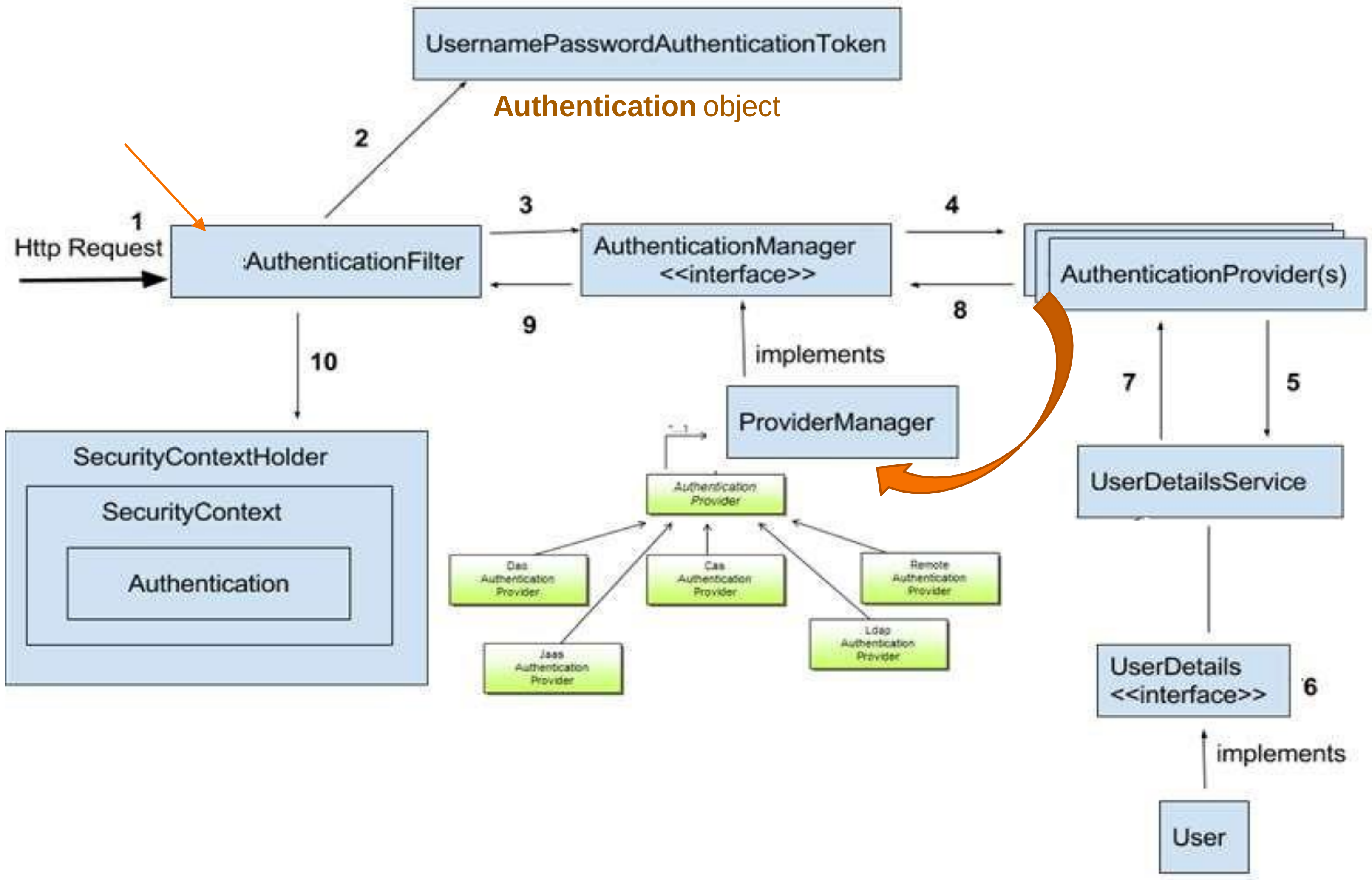
- **Principal**
  - ✓ User that performs the action
- **Authentication**
  - ✓ Confirming truth of credentials
- **Authorization**
  - ✓ Define access policy for principal
- **GrantedAuthority**
  - ✓ Application permission granted to a principal
- **SecurityContext**
  - ✓ Hold the authentication and other security information
- **SecurityContextHolder**
  - ✓ Provides access to SecurityContext

Spring Security Fundamentals I

- **AuthenticationManager**
  - Controller in the authentication process
- **AuthenticationProvider**
  - Interface that maps to a data store which stores your user data.
- **Authentication** Object
  - Object is created upon authentication, which holds the login credentials.
- **UserDetails**
  - Data object which contains the user credentials, but also the Roles of the user.
- **UserDetailsService**
  - ✓ Collects the user credentials, authorities(roles) and build an UserDetails object.

Spring Security Architecture

- Spring security has a series/chain of filters (HTTP Basic, OAuth2, JWT)


UsernamePasswordAuthenticationToken
Authentication object
2
1
Http Request
AuthenticationFilter
3
AuthenticationManager <<interface>>
4
AuthenticationProvider(s)
9
8
10
implements
ProviderManager
7
5
SecurityContextHolder
SecurityContext
Authentication
Authentication Provider
Dao Authentication Provider
Cas Authentication Provider
Remote Authentication Provider
Jaas Authentication Provider
Ldap Authentication Provider
UserDetailsService
UserDetails <<interface>>
6
implements
User

SecurityContext and SecurityContextHolder

- The **SecurityContext** and **SecurityContextHolder** are two fundamental classes of Spring Security.
  - ✓ The **SecurityContext** is used to store the details of the currently authenticated user, also known as a **principle**.
  - ✓ The **SecurityContextHolder** is a helper class, which provide access to the security context.
- **How to get the current logged-in Username in Spring Security:**

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

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

Current Logged-in

- If you ever need to know current logged-in user details e.g. in Spring MVC controller.
- I suggest you declare a dependency and let the Spring provide you the **Principal** object:

```
import java.security.Principal;

@Controller
public class MVCController {

    @RequestMapping(value = "/username",
                                      method = RequestMethod.GET)
    @ResponseBody
    public String currentUserName(Principal principal) {
        return principal.getName();
    }
}
```

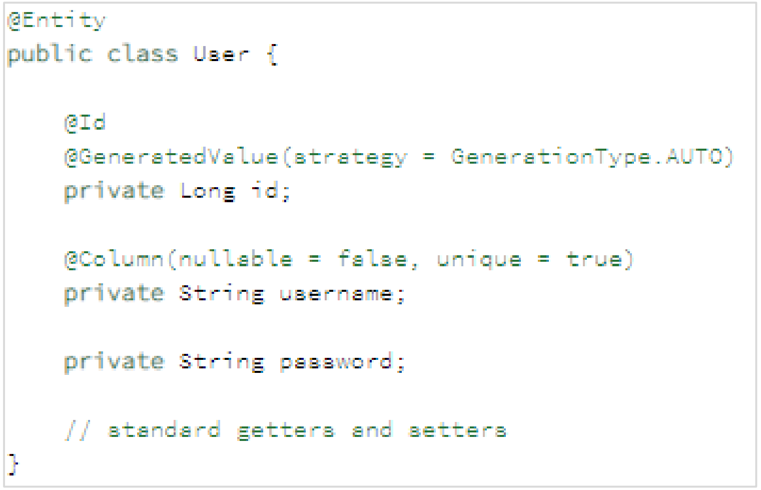
Current Logged-in

- Alternatively, you can also ask for **Authentication** object instead of a **Principal** object as shown below:

```
import org.springframework.security.core.Authentication;

@Controller
public class SpringMVCController {

    @RequestMapping(value = "/username",
                                       method = RequestMethod.GET)
    @ResponseBody
    public String currentUserName(Authentication authentication) {
         return authentication.getName();
    }
}
```

UserDetailsService interface

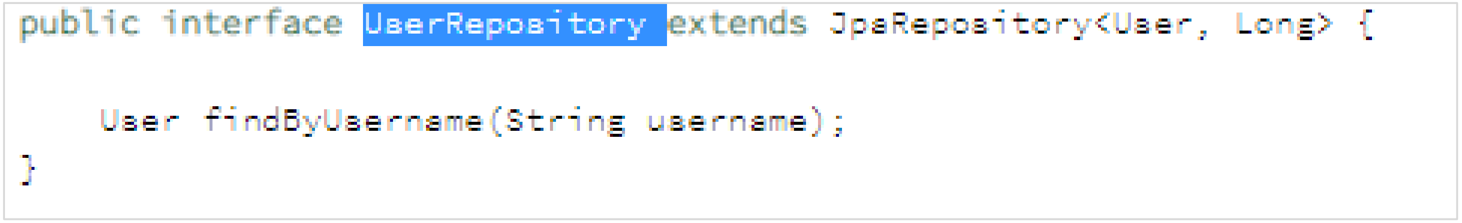
- The **UserDetailsService** means a central interface in Spring Security.
  - ✓ It is used to retrieve user-related data.
  - ✓ It is a service to search "*User account and such user's roles*".
  - ✓ It is used by the Spring Security everytime when users log in the system.
- It has one method named ***loadUserByUsername()*** which can be overridden to customize the process of finding the user.
- **Example**:
  - ✓ Create a *User* entity that is mapped to a database table, with the following attributes:



```
@Entity
public class User {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    @Column(nullable = false, unique = true)
    private String username;

    private String password;

    // standard getters and setters
}
```

UserDetailsService interface

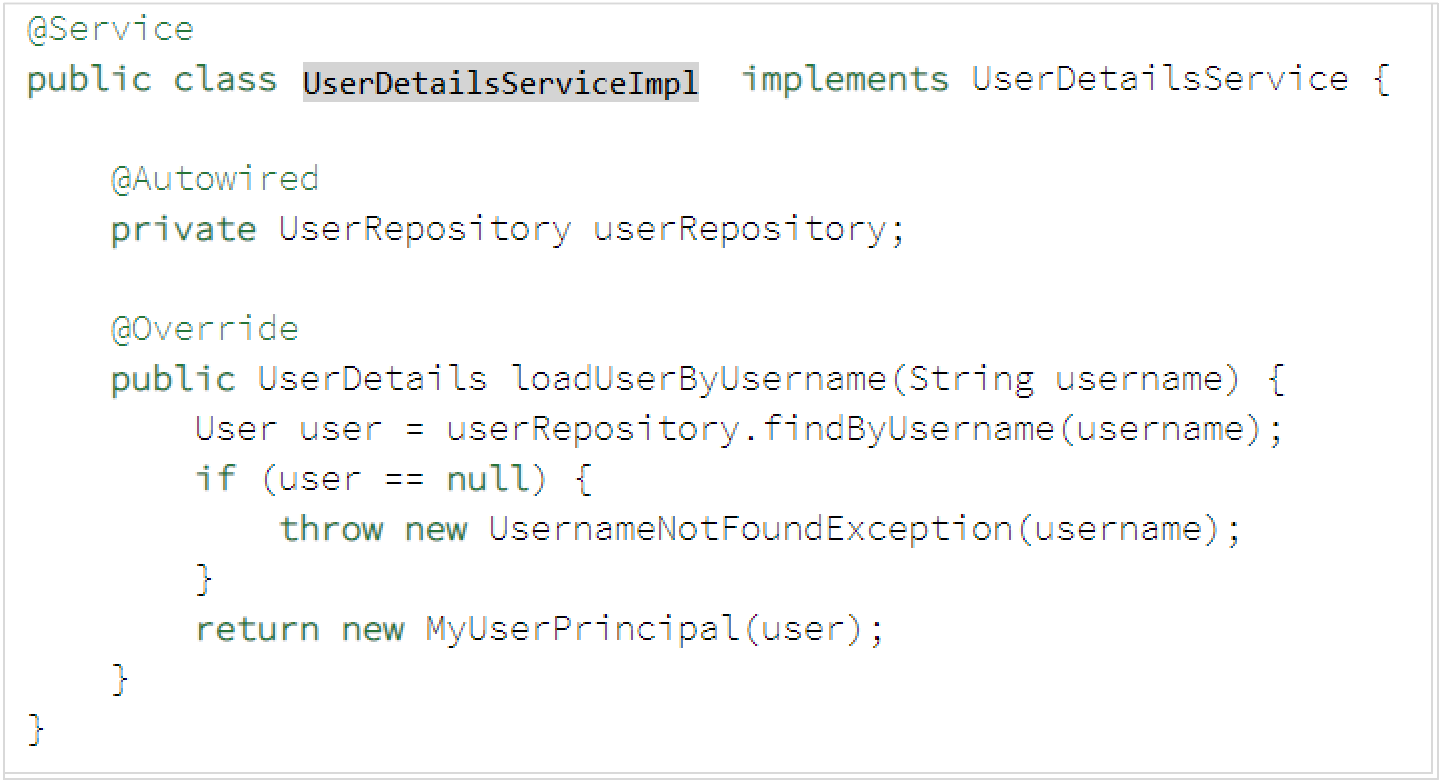
- **Retrieving a User**
  - ✓ Create a *UserRepository* interface using *Spring Data* by extending the *JpaRepository* interface:



```
public interface UserRepository extends JpaRepository<User, Long> {

    User findByUsername(String username);
}
```

UserDetailsService interface

- You will need to implement the *UserDetailsService* interface.
- We'll create a class called *UserDetailsServiceImpl* that overrides the method *loadUserByUsername()* of the interface.
  - ✓ **We retrieve the *User* object using the *UserReposiroty:***



```
@Service
public class UserDetailsServiceImpl  implements UserDetailsService {

    @Autowired
    private UserRepository userRepository;

    @Override
    public UserDetails loadUserByUsername(String username) {
        User user = userRepository.findByUsername(username);
        if (user == null) {
            throw new UsernameNotFoundException(username);
        }
        return new MyUserPrincipal(user);
    }
}
```

UserDetails Interface

- **UserDetails** là một interface cốt lõi của Spring Security.
- **UserDetails** đại diện cho một principal nhưng theo một cách mở rộng và cụ thể hơn. Vậy UserDetails cung cấp cho ta những thông tin gì?
- UserDetails methods:
  - ✓ **getAuthorities**(): trả về danh sách các quyền của người dùng
  - ✓ **getPassword**(): trả về password đã dùng trong qúa trình xác thực
  - ✓ **getUsername**(): trả về username đã dùng trong qúa trình xác thực
  - ✓ **isAccountNonExpired**(): trả về true nếu tài khoản của người dùng chưa hết hạn
  - ✓ **isAccountNonLocked**(): trả về true nếu người dùng chưa bị khóa
  - ✓ **isCredentialsNonExpired**(): trả về true nếu chứng thực (mật khẩu) của người dùng chưa hết hạn
  - ✓ **isEnabled**(): trả về true nếu người dùng đã được kích hoạt

UserDetails Interface

- The **UserDetails** interface only provides methods to access the user's basic information.
- To extend more information, we will create a class **CustomUserDetails** implements **UserDetails**:

```java
public class CustomUserDetails implements UserDetails {

    private static final long serialVersionUID = 1L;
    private String userName;
    private String password;
    private List<GrantedAuthority> authorities;

    public CustomUserDetails(String userName, String password,
            List<GrantedAuthority> authorities) {
        this.userName = userName;
        this.password = password;
        this.authorities = authorities;
    }

    @Override
    public Collection<? extends GrantedAuthority> getAuthorities() {
        return authorities;
    }
```

UserDetails Interface

```
    @Override
    public String getPassword() {
        return password;
    }

    @Override
    public String getUsername() {
        return userName;
    }

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

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

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

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

UserDetails Interface

- Update **UserDetailsServiceImpl** class:

```
public class UserDetailsServiceImpl implements UserDetailsService {
    @Autowired
    private UserRepository userRepository;

    @Override
    public UserDetails loadUserByUsername(String userName)
            throws UsernameNotFoundException {

        User user = userRepository.findByUsername(userName);

        if (user == null)
            throw new UsernameNotFoundException("User name not found");
        List<GrantedAuthority> authorities = new ArrayList<GrantedAuthority>();

        SimpleGrantedAuthority authority = new SimpleGrantedAuthority(
                "ROLE_ADMIN");

        authorities.add(authority);
        CustomUserDetails userDetail = new CustomUserDetails(userName,
              user.getPassword(), authorities);
        return userDetail;
    }
}
```

GrantedAuthority class

- A **GrantedAuthority** is an authority granted to the principal.
- The permissions are prefixed with **ROLE_**.
- For example **ROLE_ADMIN**, **ROLE_MEMBER** ...

```
List<GrantedAuthority> authorities = new ArrayList<GrantedAuthority>();

SimpleGrantedAuthority authority = new SimpleGrantedAuthority(
                "ROLE_ADMIN");

authorities.add(authority);
```

Section 2

CREATE A LOGIN APPLICATION WITH SPRING BOOT, SPRING SECURITY, JPA

Overview

- **This document is based on:**
  - ✓ Spring Boot 2.x
  - ✓ Spring Security
  - ✓ Spring Data JPA
  - ✓ JSP
  - ✓ Database: SQL Server

  To create a Login Application with Spring Boot, Spring Security, JPA

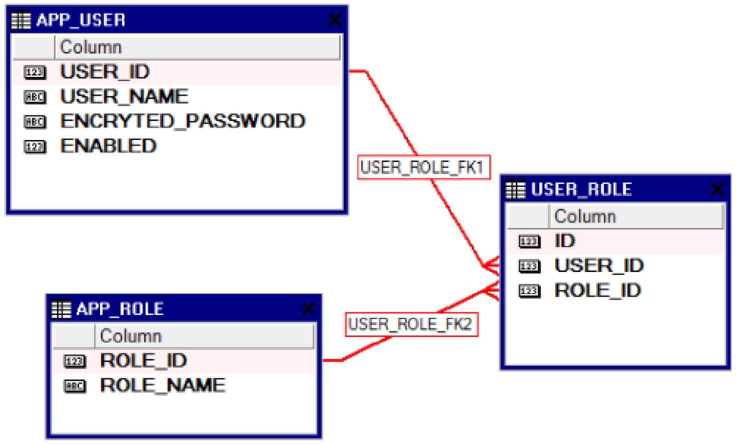
(1) Add dependency

- In the Maven we only need the **spring-boot-starter-security** dependency.

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

(2) Create database tables

- Create the 3 tables: **APP_USER, APP_ROLE,** and **USER_ROLE:**



- **Data Test:**

USER_NAME	PASSWORD	ENCRYPED_PASSWORD	ROLES
dbuser1	123	$2a$10$PrI5Gk9L.tSZiW9FXhTS8O8Mz9E97k2FZbFvGFFaSsiTUIl.TCrFu	ROLE_USER
dbadmin1	123	$2a$10$PrI5Gk9L.tSZiW9FXhTS8O8Mz9E97k2FZbFvGFFaSsiTUIl.TCrFu	ROLE_USER, ROLE_ADMIN

(3) Entity classes

❖ **AppRole** class:

```
@Entity
@Table(name = "APP_ROLE", schema = "training",
        uniqueConstraints = {
                @UniqueConstraint(name = "APP_ROLE_UK",
                 columnNames = "ROLE_NAME") })
public class AppRole {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "ROLE_ID", nullable = false)
    private Long roleId;

    @Column(name = "ROLE_NAME", length = 30, nullable = false)
    private String roleName;

    // getter and getter methods

}
```

(3) Entity classes

- **AppUser** class:

```
@Entity
@Table(name = "APP_USER", schema = "training", uniqueConstraints = {
        @UniqueConstraint(columnNames = "USER_NAME",
        name = "APP_USER_UK") })
public class AppUser {
    @Id
    @Column(name = "USER_ID")
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long userId;

    @Column(name = "USER_NAME", length = 36)
    private String userName;

    @Column(name = "ENCRYTED_PASSWORD", length = 128)
    private String encryptedPassword;

    @Column(name = "ENABLED", columnDefinition = "BIT")
    private int enabled;

     // getter and getter methods
}
```

(3) Entity classes

- **AppUserRole** class:

```
@Entity
@Table(name = "APP_USER_ROLE", schema = "training",
        uniqueConstraints = {
                @UniqueConstraint(name = "USER_ROLE_UK",
                  columnNames = { "USER_ID", "ROLE_ID" }) })
public class AppUserRole {

    @Id
    @GeneratedValue
    @Column(name = "ID", nullable = false)
    private Long id;

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "USER_ID", nullable = false)
    private AppUser appUser;

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "ROLE_ID", nullable = false)
    private AppRole appRole;
     // getter and getter methods
}
```

(4) Repositories

- **AppRoleRepository** interface:

```
public interface AppRoleRepository extends JpaRepository<AppRole, Long> {
    @Query()
    List<String> findByRoleName(Long userId);
}
```

- **AppUserRepository** interface:

```
public interface AppUserRepository extends JpaRepository<AppUser, Long> {

    AppUser findByUserName(String userName);

}
```

(5) UserDetailsService

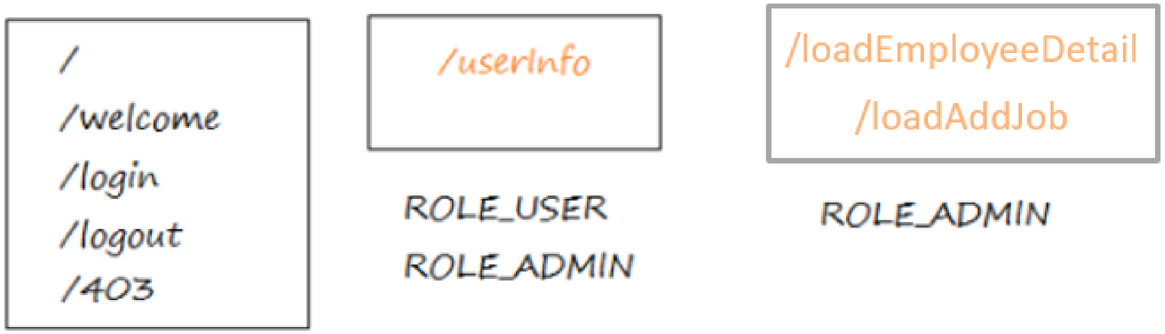
```
import org.springframework.security.core.GrantedAuthority;
import org.springframework.security.core.authority.SimpleGrantedAuthority;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.core.userdetails.UsernameNotFoundException;
@Service
public class UserDetailsServiceImpl implements UserDetailsService {
    @Autowired
    private AppUserRepository appUserRepository;
    @Autowired
    private AppRoleRepository appRoleRepository;

    @Override
    public UserDetails loadUserByUsername(String userName)
            throws UsernameNotFoundException {

        AppUser appUser = appUserRepository.findByUserName(userName);

        if (appUser == null) {
            LogUtils.getLogger().error("User not found! " + userName);

            throw new UsernameNotFoundException(
                    "User " + userName + " was not found in the database");
        }
```

(5) UserDetailsService

```
        LogUtils.getLogger().info("Found: " + appUser);

        // [ROLE_USER, ROLE_ADMIN,..]
        List<String> roleNames =
                            appRoleRepository.findByRoleNames(appUser.getUserId());

        List<GrantedAuthority> grantList = new ArrayList<GrantedAuthority>();

        if (roleNames != null) {
            for (String role : roleNames) {
                // ROLE_USER, ROLE_ADMIN,..
                GrantedAuthority authority = new SimpleGrantedAuthority(role);
                grantList.add(authority);
            }
        }

        UserDetails userDetails = (UserDetails) new
              CustomUserDetails(appUser.getUserName(), appUser.getEncryptedPassword(),
              grantList);

        return userDetails;
    }
}
```

(6) Security configuration & Remember Me

Roles mapping:



- Create a **WebSecurityConfig** class is used to configure security for the application.
- It is annotated by **@Configuration.**

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

}
```

(6) Security configuration & Remember Me

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

    @Autowired
    private UserDetailsServiceImpl userDetailsService;

    @Autowired
    private DataSource dataSource;

    @Bean
    public BCryptPasswordEncoder passwordEncoder() {
        BCryptPasswordEncoder bCryptPasswordEncoder = new BCryptPasswordEncoder();
        return bCryptPasswordEncoder;
    }

    @Autowired
    public void configureGlobal(AuthenticationManagerBuilder auth)
            throws Exception {

        // Setting Service to find User in the database.
        // And Setting PassswordEncoder
        auth.userDetailsService(userDetailsService)
                                                    .passwordEncoder(passwordEncoder());
    }
```

(6) Security configuration & Remember Me

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

        http.csrf().disable();

        // The pages does not require login
        http.authorizeRequests().antMatchers("/", "/login", "/logout").permitAll();

        // /userInfo page requires login as ROLE_USER or ROLE_ADMIN.
        // If no login, it will redirect to /login page.
        http.authorizeRequests().antMatchers("/userInfo")
                .access("hasAnyRole('ROLE_USER', 'ROLE_ADMIN')");

        // For ADMIN only.
        http.authorizeRequests().antMatchers("/loadAddJob", "/loadEmployeeDetail")
                .access("hasRole('ROLE_ADMIN')");

        // When the user has logged in as XX.
        // But access a page that requires role YY,
        // AccessDeniedException will be thrown.
        http.authorizeRequests().and().exceptionHandling().accessDeniedPage("/403");
```

(6) Security configuration & Remember Me

```
// Config for Login Form
http.authorizeRequests().and().formLogin()//
                // Submit URL of login page.
                .loginProcessingUrl("/j_spring_security_check")//Submit URL/action form
                .loginPage("/login")//
                .defaultSuccessUrl("/index")//
                .failureUrl("/login?error=true")//
                .usernameParameter("username")//
                .passwordParameter("password")
                // Config for Logout Page
                .and().logout().logoutUrl("/logout")
                .logoutSuccessUrl("/login");

// Config Remember Me.
http.authorizeRequests().and() //
                .rememberMe().tokenRepository(this.persistentTokenRepository()) //
                .tokenValiditySeconds(1 * 24 * 60 * 60); // 24h
}
@Bean
public PersistentTokenRepository persistentTokenRepository() {
        JdbcTokenRepositoryImpl db = new JdbcTokenRepositoryImpl();
        db.setDataSource(dataSource);
        return db;
}
}
```

The same name with the input tag in the form.

(7) Controller

- **/login**: open login form

```
@GetMapping(value = { "/", "/login" })
public String init(Model model) {
        LogUtils.getLogger().info("Loading login form...");
        model.addAttribute("user", new User());

        return "login";
}
```

- **/index**: login success, open index page

```
@GetMapping("/index")
public String initIndex(Principal principal, Model model) {
        LogUtils.getLogger().info(principal.getName());

        UserDetails loginedUser = (UserDetails) ((Authentication) principal)
                .getPrincipal();

         LogUtils.getLogger().info(loginedUser);

         model.addAttribute("userName", loginedUser.getUsername());

        return "index";
}
```

(7) Controller

/403: accessDeniedPage

```
@RequestMapping(value = "/403", method = RequestMethod.GET)
public String accessDenied(Model model, Principal principal) {

        if (principal != null) {
            UserDetails loginedUser = (UserDetails) ((Authentication) principal)
                                                                 .getPrincipal();

            LogUtils.getLogger().info(loginedUser);

            model.addAttribute("userInfo", loginedUser.getUsername());

            String message = "Hi " + principal.getName() //
                    + "<br> You do not have permission to access this page!";
            model.addAttribute("message", message);

        }

        return "403";
    }
```

(7) Controller

- **/userInfo**: view detail user info

```
@RequestMapping(value = "/userInfo", method = RequestMethod.GET)
public String userInfo(Model model, Principal principal) {

        // After user login successfully.
        String userName = principal.getName();

        LogUtils.getLogger().info("User Name: " + userName);

        UserDetails loginedUser = (UserDetails)
                            ((Authentication) principal).getPrincipal();

        model.addAttribute("userInfo", loginedUser.getUsername());

        return "userInfoPage";
    }
```

(8) View

- **/views/login.jsp:**

```
<form:form action="${pageContext.request.contextPath}/j_spring_security_check"
                                                       method="post" modelAttribute="user">
<h2 class="text-center">Log in</h2>
<label style="color: red">${message}</label>
<!-- JSP Expression -->
<div class="form-group">
     <form:input type="text" path="username" class="form-control" placeholder="Username" />
     <form:errors path="username" cssClass="error" />
</div>
<div class="form-group">
     <form:input type="password" path="password" class="form-control" placeholder="Password" />
     <form:errors path="password" cssClass="error" />
</div>
<div class="form-group">
     <button type="submit" class="btn btn-primary btn-block">Login</button>
</div>
<div class="clearfix">
     <label class="float-left form-check-label">
          <input type="checkbox"> Remember me
     </label>
     <a href="#" class="float-right">Forgot Password?</a>
</div>
</form:form>
```

(8) View

- **views/403.jsp**:

```
<body>
    <h2 style="color: red">${message} !!!</h2>
    <h3>Please click
        <a href="${pageContext.request.contextPath}/login">here
        </a> to login with another account!
    </h3>
</body>
```

Section 3

PASSWORD HANDLING WITH SPRING SECURITY

Password Encoders

- How Spring Security supports these algorithms and how we can handle passwords with them?
- All password encoders implement the interface **PasswordEncoder**.
  - ✓ **encode**(): to convert the plain password into the encoded form;
  - ✓ **matches**(): to compare a plain password with the encoded password.
- **Every encoder** has a default constructor that creates an instance with the default work factor:
  - ✓ **BCryptPasswordEncoder**
  - ✓ **Pbkdf2PasswordEncoder**
  - ✓ **SCryptPasswordEncoder**
  - ✓ **Argon2PasswordEncoder**

BCryptPasswordEncoder

- **BCryptPasswordEncoder** has the parameter strength.
- The default value in Spring Security is 10.
- It’s recommended to use a **SecureRandom** as salt generator, because it provides a cryptographically strong random number.

```
int strength = 10; // work factor of bcrypt
BCryptPasswordEncoder bCryptPasswordEncoder =
         new BCryptPasswordEncoder(strength, new SecureRandom());
String encodedPassword =
    bCryptPasswordEncoder.encode(plainPassword);
```

- **The output looks like this:**

```
$2a$10$EzbrJCN8wj8M8B5aQiRmiuWqVvnxna73Ccvm38aoneiJb88kkwlH2
```

Pbkdf2PasswordEncoder

- The **PBKDF2** algorithm was not designed for password encoding but for key derivation from a password.
- **Pbkdf2PasswordEncoder** runs the hash algorithm over the plain password many times.

```
String pepper = "pepper"; // secret key used by password encoding
int iterations = 200000;  // number of hash iteration
int hashWidth = 256;      // hash width in bits

Pbkdf2PasswordEncoder pbkdf2PasswordEncoder =
    new Pbkdf2PasswordEncoder(pepper, iterations, hashWidth);
pbkdf2PasswordEncoder.setEncodeHashAsBase64(true);
String encodedPassword =
                     pbkdf2PasswordEncoder.encode(plainPassword);
```

- **The output looks like this:**

```
lLDINGz0YLUUFQuuj5ChAsq0GNM9yHeUAJiL2Be7WUh43Xo3gmXNaw==
```

SCryptPasswordEncoder

- The **scrypt** algorithm can not only configure the CPU cost but also memory cost.
- This **encoder** puts the parameter for work factor and salt in the result string, so there is no additional information to save.

```
int cpuCost = (int) Math.pow(2, 14); // factor to increase CPU costs
        int memoryCost = 8;      // increases memory usage
        int parallelization = 1; // currently not supported by Spring Security
        int keyLength = 32;      // key length in bytes
        int saltLength = 64;     // salt length in bytes

SCryptPasswordEncoder sCryptPasswordEncoder = new SCryptPasswordEncoder(
          cpuCost,
          memoryCost,
          parallelization,
          keyLength,
          saltLength);
String encodedPassword = sCryptPasswordEncoder.encode(plainPassword);
```

- $e0801$jRlFuIUd6eAZcuM1wKrzswD8TeKPed9wuWf3lwsWkStxHs0DvdpOZQB32cQJnf0lq/dxL+QsbDpSyyc9Pnet1A==$P3imAo3G8k27RccgP5iR/uoP8FgWGSS920YnHj+CRVA=

Argon2PasswordEncoder

- **Argon2** is the winner of Password Hashing Competition in 2015.
- This algorithm, too, allows us to tune CPU and memory costs.
- The Argon2 encoder saves all the parameters in the result string. If we want to use this password encoder, we'll have to import the BouncyCastle crypto library.

```
int saltLength = 16; // salt length in bytes
        int hashLength = 32; // hash length in bytes
        int parallelism = 1; // currently not supported by Spring Security
        int memory = 4096;   // memory costs
        int iterations = 3;

Argon2PasswordEncoder argon2PasswordEncoder =
                                              new Argon2PasswordEncoder(
          saltLength,
          hashLength,
          parallelism,
          memory,
          iterations);
        String encodePassword = argon2PasswordEncoder.encode(plainPassword);
```

SUMMAY

1. Introduction: Spring Framework vs. Spring Boot vs. Spring Security
2. Spring Security Fundamentals I
3. Spring Security Configuration
4. Practice: Impl Security
5. Password Handling with Spring Security

Thank you