

Spring Security

Spring Security & Spring Boot

- Přidejte do pom.xml:

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

Spring Security bez Spring Boot I.

- Jako výchozí projekt vezmeme projekt, který je vytvořen podle artefaktu `org.fluttercode.knappsack:spring-mvc-jpa-archetype`
- Dále je nutné přidat do `pom.xml` závislosti.
- Nejprve přidejte do tagu `<properties>`:

```
<spring.security.version>4.2.2.RELEASE</spring.security.version>
```

Spring Security bez Spring Boot II.

- Dále přidejte do pom.xml do tagu <dependencies>:

```
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-config</artifactId>
    <version>${spring.security.version}</version>
</dependency>
```

```
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-web</artifactId>
    <version>${spring.security.version}</version>
</dependency>
```

Spring Security bez Spring Boot III.

- Přidejte do `web.xml` následující filtr a jeho mapování:

```
<filter>
    <filter-name>springSecurityFilterChain</filter-name>
    <filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>
</filter>

<filter-mapping>
    <filter-name>springSecurityFilterChain</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

Filtr se MUSÍ takto jmenovat!

- Pozor! Pokud máte ve `web.xml` více filtrů, pak záleží na pořadí tagů `<filter-mapping>` (první uvedený se provede nejdříve). Spring Security filter mapping by tedy měl být ideálně jako první.

Konfigurace Spring Security I.

- Vytvořte konfigurační soubor Spring Security ... například `src/main/resources/security.xml` s obsahem uvedeným na další stránce a nainportujte ho do root Spring application context:
 - Pomocí XML – nainportujte `security.xml` uvnitř `applicationContext.xml`:

```
<import resource="classpath:security.xml" />
```

- Pomocí anotací – přidejte níže uvedenou anotaci ke třídě s anotací `@Configuration`:

```
@ImportResource("classpath:security.xml")
```

Konfigurace Spring Security II.

```
<?xml version="1.0" encoding="UTF-8"?>

<beans:beans xmlns:beans="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://www.springframework.org/schema/security"
xsi:schemaLocation="http://www.springframework.org/schema/security
http://www.springframework.org/schema/security/spring-security-4.2.xsd
http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd">


  <http>
    <csrf disabled="true" /> <form-login />
    <intercept-url pattern="/**" access="hasRole('ROLE_USER')" />
  </http>

  <authentication-manager>
    <authentication-provider>
      <user-service>
        <user name="jirka" password="jirka" authorities="ROLE_USER" />
      </user-service>
    </authentication-provider>
  </authentication-manager>
</beans:beans>
```

Od Spring Security 3.2 je CSRF ochrana
ve výchozím nastavení zapnutá


Nejjednodušší způsob vytvoření uživatele s heslem.
Výborné pro testování, ale později se zde změní tato
konfigurace na jinou, která získává data z databáze,
LDAPu, přes CAS (SSO) apod.

Nastavení I.

- Můžete také nastavit logout URL: `<logout logout-url="/logout" />`  Výchozí hodnota
- Pokud použijete přihlašování pomocí formuláře, pak Spring Security automaticky vygeneruje přihlašovací stránku. Toto můžete změnit:

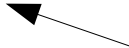
```
<http>
```

```
<form-login login-page="/Login.html"/>
```

 Pro přihlášení se použije stránka /login.html

```
<intercept-url pattern="/Login.html" access="permitAll"/>
```

```
</http>
```

 URL /login.html bude veřejně přístupné bez přihlášení

login.jsp



Nastavení II.

- Vytvořte soubor /login.html s následujícím obsahem:

```
<form action="/login" method="post">
```

```
<h2>Please sign in</h2>
```

Pozor! Nová výchozí hodnota od Spring Security 3.2

```
<div class="input-group">
```

```
<label>Username:</label>
```

```
<input type="text" name="username" placeholder="Username" class="form-control" />
```

```
</div>
```

```
<div class="input-group">
```

```
<label>Password:</label>
```

```
<input type="password" name="password" placeholder="Password" class="form-control" />
```

```
</div>
```

```
<input type="submit" name="submit" class="btn btn-primary btn-lg" />
```

```
</form>
```

Pozor! Nová výchozí hodnota od Spring Security 3.2

- Poznámka: Ve formuláři je použit Twitter Bootstrap

Basic I.

- Pokud se nechcete přihlašovat pomocí formuláře, ale pomocí basic autentizace (to využijete u webových služeb – ať už REST nebo SOAP WS), pak použijete tuto konfiguraci:

```
<http>  
  <http-basic />  
  <intercept-url pattern="/**" access="hasRole('ROLE_USER')" />  
</http>
```

Basic II.

- Form a Basic autentizaci můžete také kombinovat:

`<http pattern="/ws/**">` ← Pozor! Záleží na pořadí http tagů!

`<http-basic/>`

`<intercept-url pattern="/**" access="hasRole('ROLE_ADMIN')"/>`

`</http>`

`<http>`

`<intercept-url pattern="/**" access="hasRole('ROLE_USER')"/>`

`</http>`

- Při tomto zapojení se bude pro přístup k webové službě používat basic autentizace a pro přístup ke zbytku aplikace form autentizace.

Spring Security & Java Config

- Od Spring Security 3.2 je podporována Spring Security konfigurace pomocí Java Config.
- Hello world konfigurace (která zároveň vytvoří filtr):

@Configuration

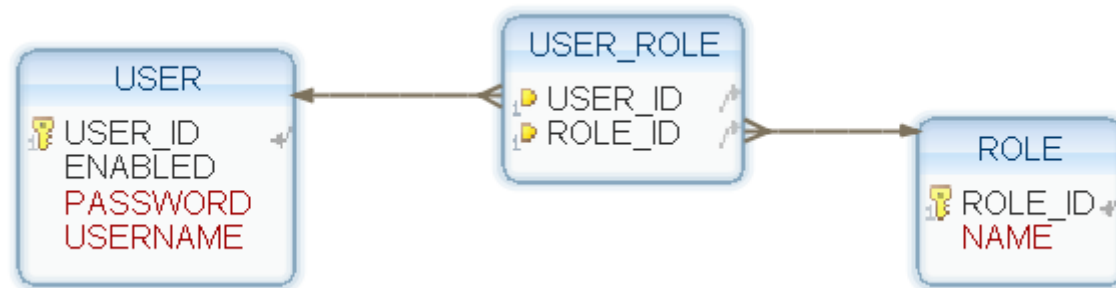
@EnableWebSecurity

```
public class SecurityConfig extends WebSecurityConfigurerAdapter {  
    @Autowired ← Opravdu @Autowired  
    public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {  
        auth.inMemoryAuthentication()  
            .withUser("user").password("password").roles("USER");  
    }  
}
```

- Více v dokumentaci:
 - <http://docs.spring.io/spring-security/site/docs/3.2.x/reference/htmlsingle/#jc>

Authentication Provider: jdbc-user-service I.

- V databázi jsou obvykle role, uživatelé a informace o tom, jaký uživatel má jakou roli v následující podobě:



- Názvy tabulek, sloupců i struktura dat se v každé databázi obvykle liší, ale Spring Security si s tím poradí.

Authentication Provider: jdbc-user-service II.

- Vložte do Spring Security konfiguračního souboru následující kus kódu:

```
<authentication-manager>
```

```
<authentication-provider>
```

```
<password-encoder hash="bcrypt" />
```

```
<jdbc-user-service data-source-ref="dataSource"
```

```
  authorities-by-username-query="select user.username, role.name from user
```

```
    join user_role on user.user_id = user_role.user_id
```

```
    join role on user_role.role_id = role.role_id
```

```
    where user.username = ?"
```

```
  users-by-username-query=
```

```
    "select username,password,enabled from user where username = ?" />
```

```
</authentication-provider>
```

```
</authentication-manager>
```

Způsob zakódování hesla, heslo je také možné osolit pomocí párové varianty tohoto tagu

Reference na beanu typu DataSource

SELECT pro získání informací o uživateli a jeho rolích

SELECT pro získání jména a hesla uživatele a jestli se může přihlásit

HTTPS I.

- Přes HTTP protokol se posílají data nezašifrovaně, což je v řadě situací problematické (například při přihlašování do webové aplikace, kdy se po HTTP protokolu přenáší nezašifrované heslo a kdokoli mezi klientem a serverem ho může zachytit – známé jako man-in-the-middle útok).
- Nejprve musíte vytvořit keystore, ve kterém budou klíče a certifikáty. Je možné k tomu použít konzolové aplikace jako keytool a openssl, ale obvykle je mnohem lepší použít grafickou aplikaci jako je portecle:
 - <http://portecle.sourceforge.net/>

HTTPS II.

- Po vygenerování keystore je nutné v Java EE serveru aktivovat HTTPS port a předat mu vytvořený keystore.
- Keystore uložte ve formátu JKS do souboru [tomcat]/conf/keystore
- V Tomcatu je nutné do server.xml přidat:

```
<Connector port="8443" protocol="HTTP/1.1" SSLEnabled="true"  
    maxThreads="150" scheme="https" secure="true"  
    clientAuth="false" sslProtocol="TLS"  
    keystoreFile="${catalina.home}/conf/keystore" keystorePass="" />
```



Cesta ke keystore

Keystore heslo

- Více informací o Tomcatu a SSL:
 - <http://tomcat.apache.org/tomcat-7.0-doc/ssl-howto.html>

HTTPS III.

- V tagu `intercept-url` můžete stanovit, jestli se pro přenos dat bude používat protokol HTTP, HTTPS nebo jestli na tom nezáleží.
- Je také možné specifikovat porty, které se budou pro HTTP a HTTPS protokoly používat.

```
<http>
```

```
  <intercept-url pattern="/*" access="hasRole('ROLE_USER')" requires-channel="https" />
```

```
  <port-mappings>
```

```
    <port-mapping http="8080" https="8443" />
```

```
  </port-mappings>
```

```
</http>
```

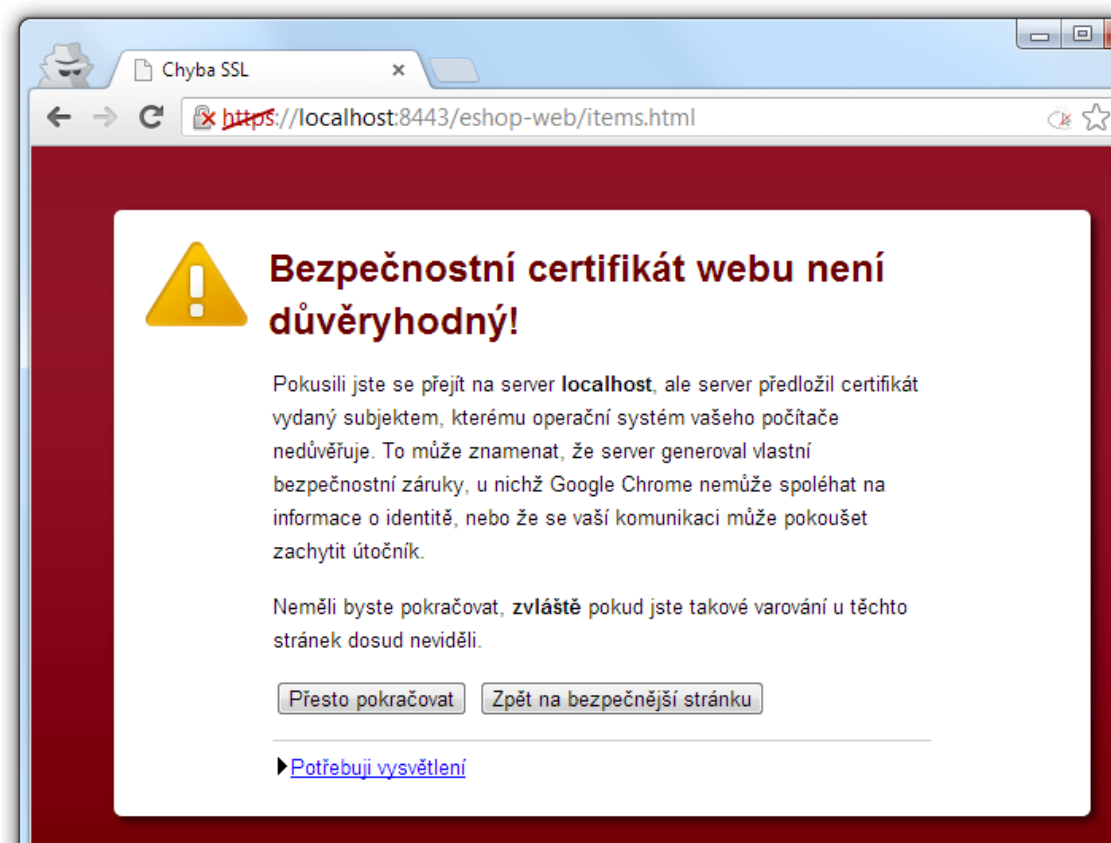
Ve výchozím nastavení
zbytečné, je to ale nutné
u jiných, exotických portů

Vynucení HTTPS
pro celou aplikaci

- Je samozřejmě možné nastavit vynucení HTTPS jenom u určitých částí webové aplikace. Každopádně na HTTPS musí být každá stránka, která spustí proces přihlašování uživatele, v opačném případě se bude heslo posílat přes síť v plaintextu!

Přidání podpory HTTPS: zadání

- Vytvořte keystore (nejjednodušším způsobem pomocí portecle), přidejte ho do Apache Tomcat (v Eclipse je obvykle nutné to udělat v projektu Servers → Tomcat X. → server.xml) a nastavte povinnost HTTPS protokolu na příslušných URL stránkách.
- **Poznámka:**
 - při prvním příchodu na stránku je toto naprosto správné chování:



Přidání podpory HTTPS: řešení

- Konfigurace `server.xml` v Apache Tomcat byla popsána na předcházejících snímcích.
- Konfigurace v `security.xml`:

```
<http>
```

```
<intercept-url pattern="/order**" access="hasRole('ROLE_USER')"  
               requires-channel="https" />
```

```
<intercept-url pattern="/user-orders**" access="hasRole('ROLE_ADMIN')"  
               requires-channel="https" />
```

```
</http>
```

Zabezpečení aplikace na úrovni metod I.

Od Spring Security 3.2 jsou pre-post-annotations ve výchozím nastavení enabled!

- Existují tři způsoby zabezpečení aplikace na úrovni metod:
 - Pomocí anotace @Secured ze Spring Security:
`<global-method-security secured-annotations="enabled" />`
 - Pomocí anotací JSR-250:
`<global-method-security jsr250-annotations="enabled" />`
 - Pomocí Pre/Post anotací:
`<global-method-security pre-post-annotations="enabled" />`
- Zabezpečení metod je standardně vypnuté, pro jejich použití je nutné je nejprve aktivovat výše uvedeným způsobem!
- Tyto způsoby je možné kombinovat, k jejich zapojení použijte pouze jeden tag `global-method-security` s více atributy:

```
<global-method-security secured-annotations="enabled"  
jsr250-annotations="enabled" pre-post-annotations="enabled" />
```

Zabezpečení aplikace na úrovni metod II.

- Kam se všemi jednotlivými typy anotací? Do servisní vrstvy!
- Použití Secured anotací:

```
@Secured("ROLE_ADMIN")
```

```
public Iterable<UserOrder> findAll() { ... }
```

- Anotaci je možné přidat před definicí metody třídy nebo interface.
 - Tato anotace omezuje přístup k metodě pouze pro danou roli nebo role (dovnitř je také možné vložit pole).
- JSR-250 anotace slouží ke stejnému účelu, ale oproti Secured anotacím jsou standardem:

```
@RolesAllowed("ROLE_ADMIN")
```

```
public Iterable<UserOrder> findAll() { ... }
```

Poznámka: Pro JSR-250 anotace je nutné přidat tuto dependency:

```
<dependency>  
  <groupId>javax.annotation</groupId>  
  <artifactId>jsr250-api</artifactId>  
  <version>1.0</version>  
</dependency>
```

Zabezpečení aplikace na úrovni metod III.

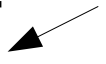
- Mnohem výkonnější než Secured a JSR-250 jsou Pre/Post anotace. Celkem jsou čtyři: `@PreAuthorize`, `@PostAuthorize`, `@PreFilter`, `@PostFilter`.
- Nejčastěji používané jsou `@PreAuthorize` a `@PostFilter`:
 - `@PreAuthorize`: Zjišťuje, jestli se metoda může vykonat.
 - `@PostFilter`: Prochází výstupní kolekci záznamů a odstraňuje záznamy, které nesplní podmínku.
 - Při použití `@PostFilter` je nutné pamatovat na to, že filtrování většího množství záznamů je velice neefektivní, výrazně výkonnější je filtrování na úrovni databáze.

Zabezpečení aplikace na úrovni metod IV.

Standardní výrazy

Výraz	Popis
hasRole([role])	Vrací true, jestli má aktuální principal roli „role“
hasAnyRole([role1,role2])	Vrací true, pokud má aktuální principal jednu z rolí
principal	Přímý přístup k principal objektu
authentication	Přímý přístup k Authentication objektu z SecurityContext
permitAll	Vždy vrátí true
denyAll	Vždy vrátí false
isAnonymous()	Jestli je aktuální principal anonym
isRememberMe()	Jestli je aktuální principal remember-me uživatel
isAuthenticated()	Jestli není uživatel anonym
isFullyAuthenticated()	Jestli není uživatel anonym nebo remember-me uživatel

Zabezpečení aplikace na úrovni metod V.

- Doplnění: výrazy je také možné použít k zabezpečení URL. K dispozici jsou všechny standardní výrazy plus jeden navíc: `hasIpAddress()`.
- Použití je následující:  Od Spring Security 3.2 je ve výchozím nastavení zapnuté

```
<http use-expressions="true">  
  <intercept-url pattern="/admin*" access="hasRole('admin') and hasIpAddress('192.168.1.0/24')" />  
</http>
```

- Pozor! V tagu `intercept-url` se poté musí používat výrazy, nikoli pouhé názvy rolí!
- `hasRole` můžete také tímto způsobem kombinovat:

```
<intercept-url pattern="/fi/referent/**" access="hasRole('referent') and hasRole('fakulta-informatiky')" />
```


Zabezpečení aplikace na úrovni metod VI.

- Příklady použití `@PreAuthorize` a `@PostFilter`:

```
@PreAuthorize("hasRole('ROLE_USER')")
```

```
public void create(Contact contact);
```

```
@PreAuthorize("hasPermission(#contact, 'admin')")
```

```
public void deletePermission(Contact contact, Sid recipient, Permission permission);
```

```
@PreAuthorize("#contact.name == authentication.name")
```

```
public void doSomething(Contact contact);
```

```
@PreAuthorize("hasRole('ROLE_USER')")
```

```
@PostFilter("hasPermission(filterObject, 'read') or hasPermission(filterObject, 'admin')")
```

```
public List<Contact> getAll();
```

↖ Aktuální objekt v kolekci

- Anotace `@PreAuthorize` je velice mocná, protože se dovnitř vkládá Spring EL výraz. Více o jejím použití v dokumentaci:
 - <http://static.springsource.org/spring-security/site/docs/3.2.x/reference/el-access.html>

Principal I.

- V Controlleru můžete lehce zjistit jméno přihlášeného uživatele:

```
@RequestMapping
```

```
public String view(Principal principal) {  
    System.out.println("principal name: " + principal.getName());  
    return "view";  
}
```

- V JSP můžete zjistit jméno přihlášeného uživatele v EL tímto způsobem:

```
${pageContext.request.remoteUser}
```

Principal II.

- Kdekoli můžete zjistit jméno přihlášeného uživatele tímto způsobem:

```
Object principal =  
    SecurityContextHolder.getContext().getAuthentication().getPrincipal();  
  
String username = null;  
  
if (principal instanceof UserDetails) {  
    username = ((UserDetails)principal).getUsername();  
} else {  
    username = principal.toString();  
}
```

isUserInRole

- Tímto způsobem můžete zjistit, jestli má přihlášený uživatel nějakou konkrétní roli:

```
@RequestMapping
public String show(Model model, Principal principal,
                   SecurityContextHolderAwareRequestWrapper request) {
    if (request.isUserInRole("ROLE_ADMIN")) {
        // TODO DODELAT ...
    }
    return "view";
}
```

- Třída SecurityContextHolderAwareRequestWrapper má řadu užitečných metod.
- Jak zjistit seznam rolí:
 - <http://stackoverflow.com/questions/10092882/how-to-get-the-current-user-roles-from-spring-security-3-1>

Spring Security & JSP I.

- Spring Security má vlastní knihovnu tagů, která slouží k aplikování zabezpečení v JSP stránkách.
- Nejprve je nutné přidat tuto závislost:

```
<dependency>  
    <groupId>org.springframework.security</groupId>  
    <artifactId>spring-security-taglibs</artifactId>  
    <version>${spring.security.version}</version>  
</dependency>
```

- Poté přidat do JSP stránky tuto knihovnu tagů:

```
<%@ taglib uri="http://www.springframework.org/security/tags" prefix="security"%>
```

pokračování



Spring Security & JSP II.

- Základní použití je následující:

```
<security:authorize access="isAuthenticated()">
    <a href="<c:url value="/logout" />">logout</a>
</security:authorize>
```

- Výsledek výrazu můžete také uložit do pomocného atributu:

```
<security:authorize access="isAuthenticated()" var="LoggedIn" />
<c:choose>
    <c:when test="${loggedIn}">
        Obsah pro přihlášeného uživatele
    </c:when>
    <c:otherwise>
        Obsah pro nepřihlášeného uživatele
    </c:otherwise>
</c:choose>
```

Spring Security & Thymeleaf

- Nejprve je nutné přidat tuto závislost:

```
<dependency>
  <groupId>org.thymeleaf.extras</groupId>
  <artifactId>thymeleaf-extras-springsecurity4</artifactId>
</dependency>
```

- Dále přidejte k html tagu atribut:

```
xmlns:sec="http://www.thymeleaf.org/thymeleaf-extras-springsecurity4"
```

- Použití:

```
<th:block sec:authorize="${hasRole('ROLE_ADMIN')}">
</th:block>
```

Registrace nového uživatele

- Spring Security se stará pouze o přihlášení uživatele a jestli má práva na čtení příslušného obsahu. Nemá nic pro registraci nového uživatele či editaci stávajícího uživatele.
- Je to z toho důvodu, že v různých firmách je toto řešeno různými způsoby, tudíž si musíme vytvořit vlastní rutiny pro vytvoření a změnu uživatele.
- Spring Security nám ale v něčem může hodně pomoci a to je v enkódování hesla:

```
User user = new User();  
user.setUsername("jirka");  
user.setEnabled(true);  
BCryptPasswordEncoder encoder  
    = new BCryptPasswordEncoder();  
String hashedPass = encoder.encode("jirka");  
user.setPassword(hashedPass);
```

Všechny enkódovací třídy:

- BaseDigestPasswordEncoder
- BasePasswordEncoder
- LdapShaPasswordEncoder
- Md4PasswordEncoder
- Md5PasswordEncoder
- MessageDigestPasswordEncoder
- PlaintextPasswordEncoder
- ShaPasswordEncoder
- BCryptPasswordEncoder

BCryptPasswordEncoder

- Pokud tvoříte zabezpečení na zelené louce a nemáte v databázi žádné uživatele a hesla, použijte BcryptPasswordEncoder. V současnosti se jedná o nejlepší způsob.
- Zakódování hesla:

```
BCryptPasswordEncoder encoder = new BCryptPasswordEncoder();
```

```
String encodedPass = encoder.encode(pass);
```

- Zapojení v konfiguraci:

```
<beans:bean id="encoder"
```

```
    class="org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder" />
```

```
<authentication-manager>
```

```
    <authentication-provider>
```

```
        <password-encoder ref="encoder" />
```

```
    ...
```

Remember Me autentizace

- Spring Security podporuje dvě implementace Remember Me:
 - **Hash token**
 - Uživatelovi se pošle cookie ve formátu: `base64(username + ":" + expirationTime + ":" + md5Hex(username + ":" + expirationTime + ":" + password + ":" + key))`
 - Nebezpečný v tom smyslu, že když útočník odchyťí tento token, tak pokud uživatel nezmění heslo, útočník se může přihlašovat dokud nevyprší platnost tokenu.
 - Velice jednoduché nastavení, stačí do http tagu přidat:
`<remember-me key="myAppKey"/>`
 - **Persistentní**
 - Vyžaduje tabulku v databázi, jinak zapojení je obdobně jednoduché:
`<remember-me data-source-ref="myDataSource"/>`
 - Podrobnější popis:
 - http://jaspan.com/improved_persistent_login_cookie_best_practice

Authentication Success/Failure Handler I.

- Často se dostanete do situace, kdy po úspěšném či neúspěšném přihlášení chcete vykonat nějakou akci. Jak na to?
- Nastavte v tagu `<form-login>` handlers:

```
<form-login login-page="/Login.jsp"
            authentication-success-handler-ref="authSuccessHandler"
            authentication-failure-handler-ref="authFailureHandler" />
```

- Vytvořte Spring beany:

```
@Service("authSuccessHandler")
public class AuthSuccessHandler implements AuthenticationSuccessHandler {
    @Override
    public void onAuthenticationSuccess(HttpServletRequest request, HttpServletResponse response,
                                       Authentication authentication) throws IOException, ServletException {
        System.out.println("uzivatel se prihlasil: " + authentication.getName());
        response.sendRedirect(request.getContextPath());
    }
}
```

Je to Spring bean

pokračování



Authentication Success/Failure Handler II.

- Spring bean pro chybu při přihlašování:

```
@Service("authFailureHandler")

public class AuthFailureHandler implements AuthenticationFailureHandler {

    @Override

    public void onAuthenticationFailure(HttpServletRequest request,

                                         HttpServletResponse response, AuthenticationException exception)

                                         throws IOException, ServletException {

        String name = request.getParameter("username");

        System.out.println("Uzivatel se nepodarilo prihlasit: " + name);

        response.sendRedirect(request.getHeader("referer"));

    }

}
```

- Časté použití tohoto mechanismu je pro uložení informace o tom, kdy se uživatel naposledy úspěšně přihlásil (success handler) a kolik je neúspěšných pokusů o přihlášení (failure handler).