

Progetto con Spring senza XML: parte 2

Progetto con il database

Per far sì che la nostra applicazione possa connettersi ad un database postgresql, sostituiamo il seguente codice XML contenuto in springmvc-config.xml

```
<!-- DATASOURCE: collegamento con il database -->
<bean id="dataSource" class="org.springframework.jdbc.datasource.
                                DriverManagerDataSource">
    <property name="driverClassName" value="org.postgresql.Driver" />
    <property name="url" value="jdbc:postgresql://
                                localhost:5432/agenzia0" />
    <property name="username" value="postgres" />
    <property name="password" value="paperino" />
</bean>
```

con il seguente codice Java da scrivere nella classe WebConfig.java

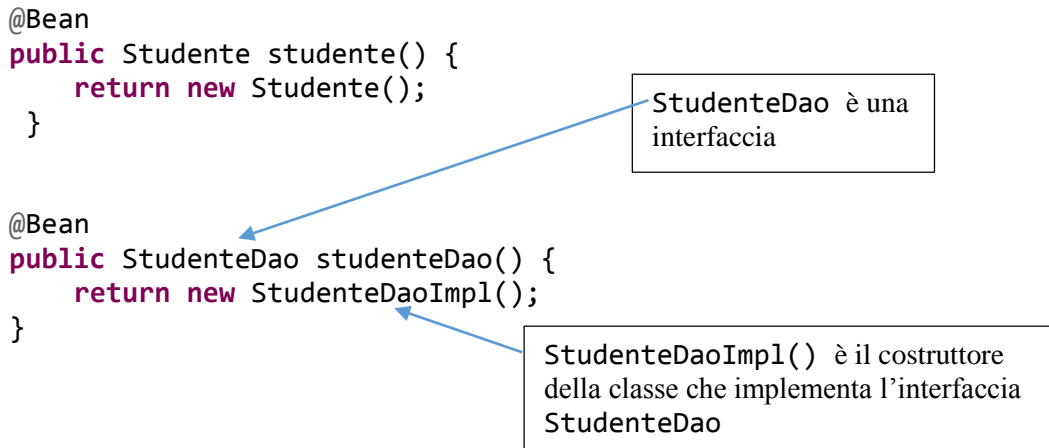
```
@Bean(name = "datasource")
public DriverManagerDataSource driverManagerDataSource() {
    DriverManagerDataSource dataSource = new DriverManagerDataSource();
    dataSource.setDriverClassName("org.postgresql.Driver");
    dataSource.setUrl("jdbc:postgresql://localhost:5432/studente0");
    dataSource.setUsername("postgres");
    dataSource.setPassword("paperino");
    return dataSource;
}
```

Nel progetto p1step2, all'interno del file springmvc-config.xml sono contenute due definizioni di bean necessarie all'inserimento dei dati nel database.

```
<bean id="studente" class="domain.Studente">
</bean>

<bean id="studenteDao" class="persistence.StudenteDaoImpl">
</bean>
```

Queste definizioni di bean in Java diventano:



Esternalizzazione e internalizzazione del progetto

cosa si intende con esternalizzazione

Fino adesso abbiamo scritto tutto il testo delle etichette delle JSP nel codice.

Con la esternalizzazione, noi andiamo a scrivere i testi delle etichette presenti nelle JSP, in un file separato. In questo modo possiamo gestire tali testi da un unico file, centralizzando quindi l'eventuale operazione di modifica.

Come viene effettuata l'esternalizzazione

creazione dei file contenente i vari testi

nella cartella **resources** creiamo la cartella **il8n** e i file `messages.properties` e `messages_it_IT.properties`

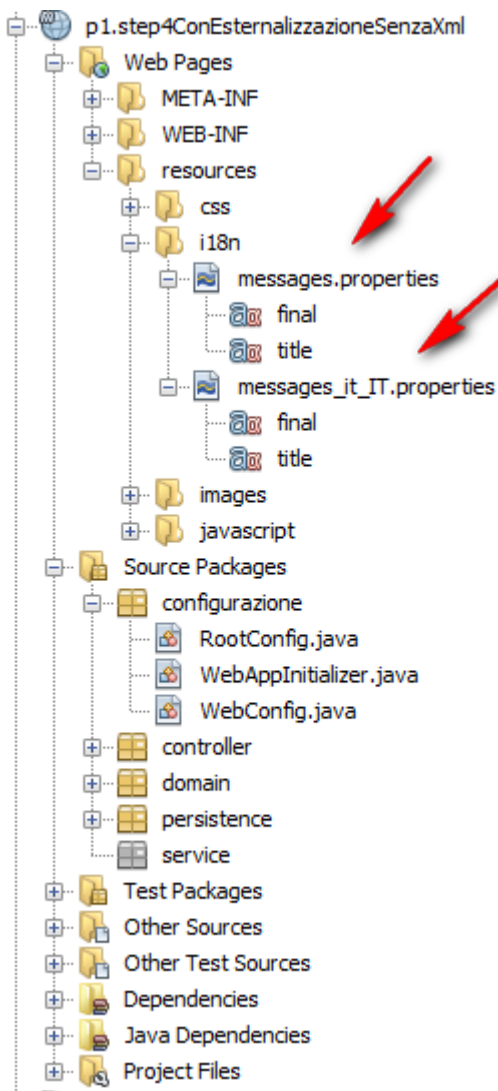
Nel file `messages.properties` scriviamo:

```
title=title
final=final
```

Nel file `messages_it_IT.properties` scriviamo:

```
title=L'esternalizzazione funziona)
final=(FINE PAGINA ottenuto con l'esternalizzazione)
```

Lo schema della applicazione diventa quindi:



modifica del file di configurazione di Spring

Di seguito vediamo la parte di codice del file `springmvc-config.xml` (file di configurazione di Spring) da sostituire

```
<!--Questo bean permette la gestione esternalizzata dei messaggi cioè contenuti
in un unico file esterno alle JSP -->
<bean id="messageSource"
      class="org.springframework.context.support.
              ReloadableResourceBundleMessageSource">
    <property name="basename" value="WEB-INF/i18n/messages" />
</bean>
```

che viene sostituito con il seguente codice Java da scrivere nella classe `WebConfig.java`

```
@Bean
public MessageSource messageSource() {
    ReloadableResourceBundleMessageSource messageSource = new
        ReloadableResourceBundleMessageSource();
    messageSource.setBasename("/resources/i18n/messages");
    return messageSource;
}
```

jsp

nella JSP in cui vogliamo fare l'operazione di esternalizzazione, aggiungiamo il riferimento al seguente taglib:

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

Nella JSP aggiungiamo quindi il seguente codice

```
<h1>MENU' <spring:message code="title" /></h1>
...
<spring:message code="final" />
```

`<spring:message` dice a Spring di andare a cercare i messaggi `title` e `final` in un file esterno (avente estensione `.properties`) alla JSP

Facendo girare il codice otteniamo:



MENU' (L'esternalizzazione funziona)

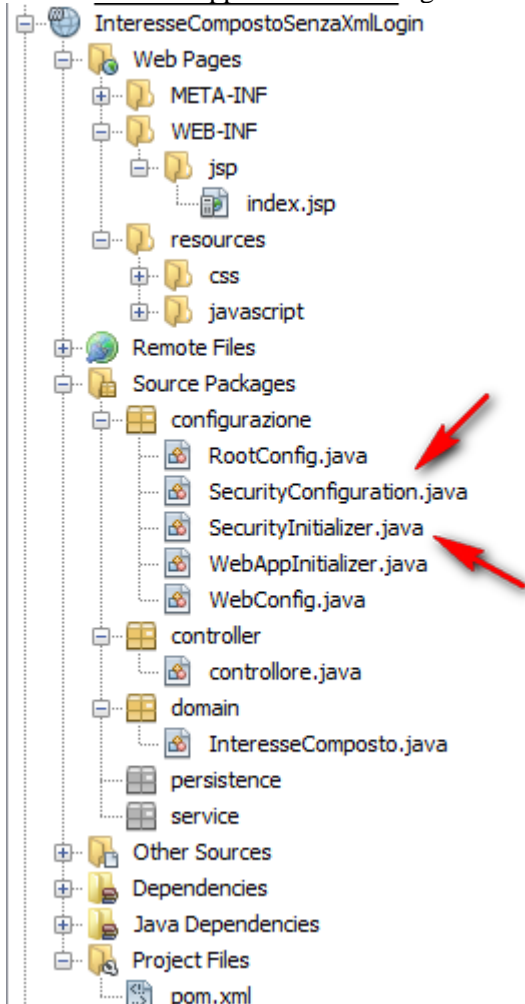
[Prenotazione](#)

[Tabella Studenti](#)

(FINE PAGINA ottenuto con l'esternalizzazione)

Spring security: Login con autenticazione

Lo schema della applicazione è il seguente:



pom.xml

Nel file pom.xml aggiungiamo le dipendenze rispetto a Spring security

```
<!-- Spring Security -->
```

```
<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>
```

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

e aggiungiamo tra `<properties>` e `</properties>`

```
<spring.security.version>3.2.0.RELEASE</spring.security.version>
```

descrittore dell'applicazione web.xml in java

```
package configurazione;

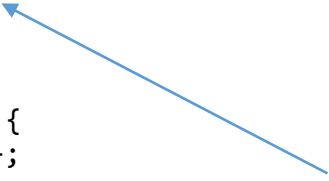
import org.springframework.web.servlet.support.
    AbstractAnnotationConfigDispatcherServletInitializer;
import org.springframework.core.annotation.Order;

public class WebAppInitializer extends
    AbstractAnnotationConfigDispatcherServletInitializer {

    @Override
    protected Class<?>[] getRootConfigClasses() {
        return new Class<?>[] { SecurityConfiguration.class };
    }

    @Override
    protected Class<?>[] getServletConfigClasses() {
        return new Class<?>[] { WebConfig.class };
    }

    @Override
    protected String[] getServletMappings() {
        return new String[] { "/" };
    }
}
```



parte di codice del file web.xml che sostituiamo con la configurazione Java

Di seguito vediamo la parte di codice del file web.xml (file di configurazione di Spring) da sostituire con Java

```
<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>
```

```
<filter-mapping>
```

specifica che il filtro DelegatingFilterProxy è mappato su tutte le richieste Web (HTTP request) in arrivo.

```
<filter-name> assegna il nome al filtro.
```

Questo codice serve a configurare il filtro DelegatingFilterProxy nel file web.xml e viene sostituito con la seguente classe Java:

SecurityInitializer.java

```
package configurazione;

import org.springframework.core.annotation.Order;
import org.springframework.security.web.context.
    AbstractSecurityWebApplicationInitializer;

public class SecurityInitializer extends
    AbstractSecurityWebApplicationInitializer {

}
```

file di configurazione di Spring security che sostituiamo con la configurazione Java

Di seguito vediamo il file xml di configurazione relativo a spring security:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<beans:beans xmlns="http://www.springframework.org/schema/security"
    xmlns:beans="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/security
        http://www.springframework.org/schema/security/spring-security-
        3.2.xsd">

    <http use-expressions="true">
        <intercept-url pattern="/visualizzaTabellaDegliStudenti"
            access="hasAnyRole('ROLE_CUSTOMER', 'ROLE_ADMIN')" />
        <form-login />
        <logout />
        <remember-me />
        <headers>
            <cache-control />
            <xss-protection />
        </headers>
    </http>

    <authentication-manager>
        <authentication-provider>
```

```

        <user-service>
            <user name="admin" password="admin"
                authorities="ROLE_ADMIN" />
        </user-service>
    </authentication-provider>
</authentication-manager>

    <global-method-security secured-annotations="enabled"
        jsr250-annotations="enabled" />
</beans:beans>

```

Ecco la classe Java che sostituisce il file XML di configurazione di Spring Security

SecurityConfiguration.java

```

package configurazione;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.authentication.builders.
    AuthenticationManagerBuilder;
import org.springframework.security.config.annotation.web.builders.
    HttpSecurity;
import org.springframework.security.config.annotation.web.configuration.
    EnableWebSecurity;

import org.springframework.security.config.annotation.web.configuration.
    WebSecurityConfigurerAdapter;

@Configuration
@EnableWebSecurity
public class SecurityConfiguration extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests()
            .antMatchers("/**")
            .authenticated()
            .and()
            .formLogin();
    }

    @Override
    protected void configure(AuthenticationManagerBuilder auth)
        throws Exception {
        auth.inMemoryAuthentication()
            .withUser("user").password("password").roles("USER")
            .and()
            .withUser("admin").password("password").roles("ADMIN");
    }
}

```


Come funziona la configurazione di Spring security in XML: elemento <http>

```
<http use-expressions="true">
  <intercept-url pattern="/visualizzaTabellaDegliStudenti"
    access="hasAnyRole('ROLE_CUSTOMER', 'ROLE_ADMIN')" />
  <form-login />
  <logout />
  <remember-me />
  <headers>
    <cache-control />
    <xss-protection />
  </headers>
</http>
```

Spring recipes 3rd edition pag 331

Spring Security consente di configurare la protezione delle applicazioni web attraverso l'elemento <http> .

- **Servizio di login Form-based:** questo servizio fornisce una pagina di login di default che contiene un modulo di accesso per gli utenti che vogliono accedere alla applicazione.
- **Servizio logout:** questo servizio fornisce un gestore, mappato con un URL, per gli utenti che vogliono uscire dalla applicazione applicazione .
- **Autenticazione HTTP di base:** Questo servizio è in grado di elaborare le credenziali di autenticazione di base contenute nelle intestazioni delle richieste HTTP. Può anche essere utilizzato per l'autenticazione di richieste effettuate con protocolli di comunicazione remota e servizi web.
- **Login Anonimo:** Questo servizio assegna un committente e concede autorità a un utente anonimo in modo che è possibile gestire un utente anonimo come un utente normale.
- **Supporto Remember-me:** Questo servizio può ricordare l'identità di un utente in più sessioni del browser; di solito memorizza un cookie nel browser dell'utente.
- **integrazione delle API Servlet:** questo servizio consente di accedere alle informazioni di sicurezza contenute nella propria applicazione web tramite API Servlet standard, come `HttpServletRequest.isUserInRole ()` e `HttpServletRequest.getUserPrincipal ()`.

getting started with Spring framework cap 14

Il file XML dell' application context della sicurezza, utilizza lo schema `spring-security-3.2.xsd`.

Il filtro `DelegatingFilterProxy`, che noi abbiamo configurato in precedenza nel descrittore della applicazione (file `web.xml`), delega la gestione della richiesta Web al bean `springSecurityFilterChain`.

Il bean di nome `springSecurityFilterChain` è ottenuto dal framework security di Spring, analizzando l'elemento `<http` che, contiene la configurazione della sicurezza dell'applicazione rispetto le richieste Web.

elemento `<intercept-url`

specifica una espressione EL di Spring che restituisce un valore booleano.

Se tale valore è true, gli URL che corrispondono con gli attributi specificati in tale elemento, sono accessibili all'utente.

Il framework Spring fornisce alcune espressioni precostruite.

Nel nostro esempio, l'espressione `hasAnyRole('ROLE_CUSTOMER', 'ROLE_ADMIN')` restituisce true solo se l'utente autenticato ha ruolo `ROLE_CUSTOMER` o `ROLE_ADMIN`.

Poiché l'attributo `pattern` specifica `/**`, che corrisponde a tutti gli URL, l'elemento `<intercept-url` specifica che solo gli utenti con il ruolo `ROLE_CUSTOMER` o `ROLE_ADMIN` possono accedere all'applicazione.

Notare che l'uso della espressione EL è possibile solo se l'attributo `use-expressions` si imposta a true.

elemento `<form-login />`

configura una pagina di login che viene utilizzata per autenticare gli utenti.

È possibile utilizzare vari attributi dell'elemento `<form-login />` come `login-page`, `default-target-url` eccetera.

L'attributo `login-page` permette di specificare l'URL della pagina di login.

Se tale attributo non è specificato, la pagina di login che viene mostrata è quella avente URL: `/spring_security_login`.

Elemento `<logout />`

permette di specificare l'URL che effettua il processo di logout.

Elemento `<remember-me />`

configura l'autenticazione `remember-me` in cui, l'applicazione Web, ricorda l'identità dell'utente autenticato tra due sessioni.

Quando l'utente viene autenticato, il framework Spring security genera un token univoco che può essere memorizzato in memoria permanente oppure inviato all'utente con un cookie.

Quando l'utente ritorna all'applicazione Web, il token viene recuperato dal cookie e l'utente viene automaticamente autenticato.

Nel nostro esempio, il token è memorizzato in un cookie.

Elemento `<headers>`

specifica le impostazioni di sicurezza che vengono aggiunte, dal framework Spring security, alla risposta http inviata al cliente.

Come funziona la configurazione di Spring security in XML : Elemento `<authentication-manager>`

```

<authentication-manager>
  <authentication-provider>
    <user-service>
      <user name="admin" password="admin"
        authorities="ROLE_ADMIN" />
    </user-service>
  </authentication-provider>
</authentication-manager>

```

Elemento `<authentication-manager>`: configura una istanza di `AuthenticationManager`.

Elemento `<authentication-provider>`

configura una istanza di `AuthenticationProvider`.

Di default, questo elemento configura un `DaoAuthenticationProvider` (una implementazione di `AuthenticationProvider`) che utilizza `UserDetailsService` di Spring come DAO (Data Object Access) per caricare i dettagli degli utenti abilitati.

`DaoAuthenticationProvider` esegue l'autenticazione dell'utente confrontando le credenziali di accesso fornite dall'utente stesso, con i dettagli dell'utente caricati a partire da `UserDetailsService` (`UserDetailsService` contiene i dati associati ai vari nomi utente). Notiamo che `UserDetailsService` può caricare i dati dell'utente da un database, da un generico file, o da un qualsiasi altro deposito contenente i dati degli utenti autenticati.

Elemento `<user-service>`

configura un `UserDetailsService` formato dagli utenti autenticati definiti negli elementi `<user>`.

L'attributo `name` specifica il nome dell'utente autenticato,
 l'attributo `password` specifica la password dell'utente autenticato,
 l'attributo `authorities` specifica il ruolo dell'utente autenticato.

Nel nostro esempio, l'applicazione ha tre utenti autenticati, un `admin` con `authorities` (ruolo) `ROLE_ADMIN` e due clienti che hanno ruolo `ROLE_CUSTOMER`.

Di seguito vediamo il codice Java che sostituisce il file XML di configurazione di Spring Security


Come funziona il codice

quando un utente accede (tramite il browser) alla applicazione,
 l'URL `/` è inviato alla applicazione.

Nel controllore che gestisce la nostra applicazione (`controllore.java`)
 l'URL `/` attiva il metodo `inizio(...)` che lancia la JSP `index.jsp`.

Prima però che l'URL `/` possa attivare il metodo `inizio (...)`, tale URL viene intercettato dal metodo `configure(HttpSecurity http)` grazie al metodo `.antMatchers("/**")`

Questo fa sì che Spring security lanci la JSP di login di default (che è fornita da Spring security stesso) e che è mostrata qua sotto:



http://localhost:8080/interesseComposto/spring_security_

Login with Username and Password

User:

Password:

☐ Remember me on this computer.

Login

Se inseriamo username e password corretta, (che nel nostro caso sono admin, password) Spring security ci manda alla JSP `index.jsp`

Se inseriamo username e password sbagliati, Spring security ci tiene sulla pagina di login visualizzando un messaggio di errore

Nota: CLASSPATH

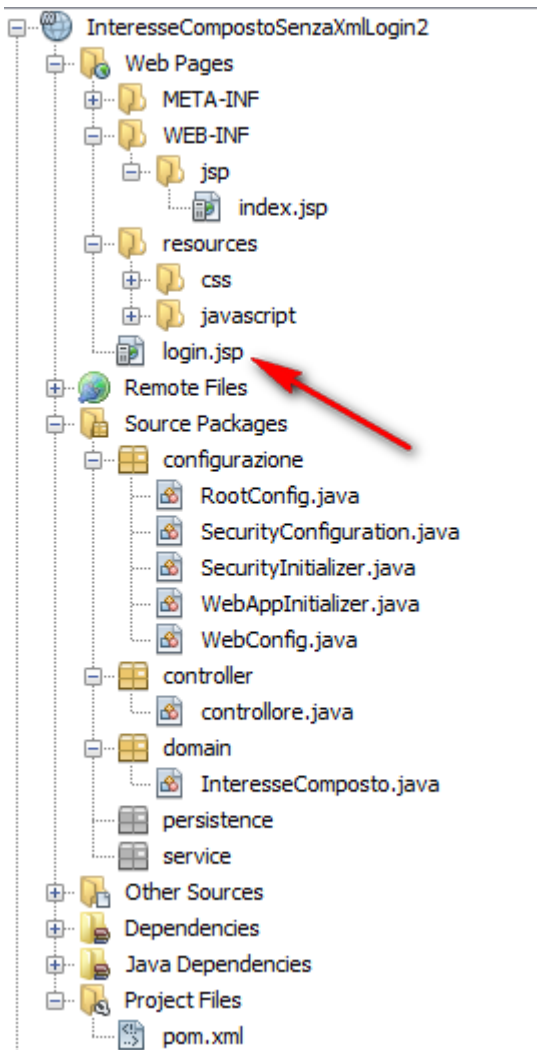
<http://www.html.it/faq/cose-il-classpath/>

Il CLASSPATH è una variabile d'ambiente che descrive una lista di directory in cui il compilatore Java può andare a ricercare eventuali classi (.class) o librerie referenziate all'interno di un'applicazione.

Spring security: pagina di login non generata da Spring 1

schema del codice

In questo caso aggiungiamo una pagina di login, creata da noi, in sostituzione della pagina di login generata automaticamente da Spring.



È preferibile mettere la pagina di login all'esterno della cartella WEB-INF in modo che sia visualizzabile dall'utente.

Modifiche al metodo `configure()` della classe `SecurityConfiguration.java`

Vediamo quindi come cambiano le configurazioni della classe `SecurityConfiguration.java`:
il metodo `configure(HttpSecurity http)`

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests()
        .antMatchers("/**")
        .authenticated()
        .and()
        .formLogin();
}
```

diventa

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests()
        // considera tutti gli url della applicazione
        .antMatchers("/**").authenticated().anyRequest()
```

```

        .authenticated()

        .and()
        // pagina di login diversa da quella generata da spring
        .formLogin().loginPage("/login.jsp").permitAll()

        .and()
        // disabilita controllo csrf
        .csrf().disable();
    }
}

```

con `login.jsp` che rappresenta la jsp da cui viene effettuato il login e, che è creata da noi.

La pagina di login

Di seguito, vediamo la pagina di login utilizzata nel nostro software che, è stata presa dalla documentazione di Spring security:

<http://docs.spring.io/spring-security/site/docs/3.2.6.RELEASE/reference/htmlsingle/#jc-form>

3.3. Java Configuration and Form Login

login.jsp

```

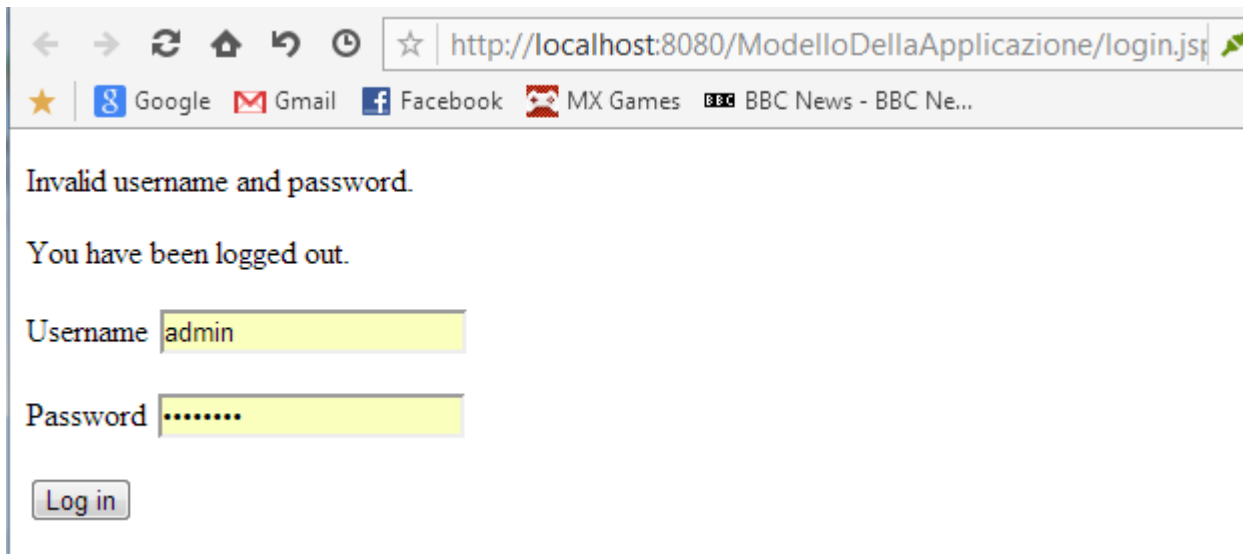
<!DOCTYPE html>
<c:url value="/Login" var="loginUrl" />
<form action="${loginUrl}" method="post">
    <c:if test="${param.error != null}">
        <p>Invalid username and password.</p>
    </c:if>
    <c:if test="${param.logout != null}">
        <p>You have been logged out.</p>
    </c:if>
    <p>
        <label for="username">Username</label> <input type="text"
            id="username" name="username" />

        <p>
            <label for="password">Password</label> <input type="password"
                id="password" name="password" />
        </p>
        <input type="hidden" name="${_csrf.parameterName}"
            value="${_csrf.token}" />
        <button type="submit" class="btn">Log in</button>
    </form>

```

Cosa accade quando eseguo il codice

quando mando in esecuzione il codice, viene visualizzata la seguente pagina:



Invalid username and password.

You have been logged out.

Username

Password

Se inserisco come username “admin” e come password “password”, l’applicazione accede alla parte operativa del programma.

Come funziona il codice

quando un utente accede (tramite il browser) alla applicazione, l’URL `/` è inviato alla applicazione.

Nel controllore che gestisce la nostra applicazione (`controllore.java`) l’URL `/` attiva il metodo `inizio(...)` che lancia la JSP `index.jsp`.

Prima però che l’URL `/` possa attivare il metodo `inizio (...)`, tale URL viene intercettato dal metodo `configure(HttpSecurity http)` grazie al metodo `.antMatchers("/")`.

Il metodo `loginPage("/login.jsp")` di `configure(HttpSecurity http)` lancia la pagina `login.jsp`.

La pagina di login ha un form con la seguente inizializzazione:

```
<c:url value="/login" var="loginUrl" />
<form action="${loginUrl}" method="post">
```

osserviamo, come visto in precedenza, che

```
<form action="${loginUrl}" method="post">
```

può essere sostituito con

```
<form action="${pageContext.request.contextPath}/Login.jsp" method="post">
```

il risultato finale è che gli elementi di input del form username e password, inviano il proprio contenuto a spring che confronta questo valore con i valori contenuti nel seguente metodo:

```

@Override
protected void configure(AuthenticationManagerBuilder auth)
    throws Exception {
    auth.inMemoryAuthentication()
        .withUser("user").password("password").roles("USER")
        .and()
        .withUser("admin").password("password").roles("ADMIN");
}

```

Nota: come funziona il tag <c:url> di JSTL core

<http://beginnersbook.com/2013/11/jstl-curl-core-tag/>

sintassi

Il tag <c:url> è usato per la codifica degli URL.

Questo tag converte un URL relativo in un URL dell'application context (contesto dell'applicazione).

Attributo obbligatorio:

value => rappresenta l'URL di partenza

Attributi opzionali:

var => nome di variabile in cui viene memorizzato l'URL risultante.

context => è usato per specificare il nome del progetto o della applicazione.

scope => scope della variabile in cui è memorizzato l'URL risultante

Esempio 1: uso dell'attributo value

considero la seguente JSP che fa parte di un progetto di nome BeginnersBook:

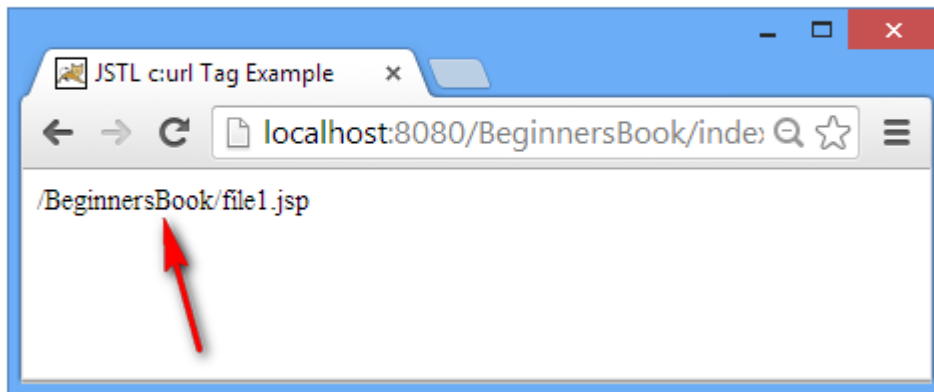
```

<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<html>
<head>
<title>JSTL c:url Tag Example</title>
</head>
<body>
    <c:url value="/file1.jsp" />
</body>
</html>

```

Il risultato che ottengo quando viene eseguita questa JSP è il seguente:

/BeginnersBook/file1.jsp



Esempio 2: uso dell'attributo var

sostituiamo

```
<body>
  <c:url value="/file1.jsp" />
</body>
```

con

```
<body>
  <c:url var="myurl" value="/file1.jsp"/>
  ${myurl}
</body>
```

Il risultato che ottengo quando viene eseguita la JSP è il seguente:

/BeginnersBook/file1.jsp

Esempio 3: uso dell'attributo context

sostituiamo

```
<body>
  <c:url value="/file1.jsp" />
</body>
```

con

```
<body>
  <c:url var="myurl" value="/file1.jsp" context="/MyJSPProject"/>
  ${myurl}
</body>
```

Il risultato che ottengo quando viene eseguita la JSP è il seguente:

/MyJSPProject/file1.jsp

Esempio 4: uso dell'attributo scope

sostituiamo

```
<body>
  <c:url value="/file1.jsp" />
</body>
```

con

```
<body>
  <c:url var="myurl" value="/file1.jsp" context="/MyJSPProject"
        scope="session"/>

  ${sessionScope.myurl}
</body>
```

Il risultato che ottengo quando viene eseguita la JSP è il seguente:

```
/MyJSPProject/file1.jsp
```

Spring security: pagina di login non generata da Spring 2

problema: non riesco utilizzare la dichiarazione della libreria core di JSTL

La pagina di login visto in precedenza:

```
<!DOCTYPE html>
<c:url value="/login" var="loginUrl" />
<form action="${loginUrl}" method="post">
  <c:if test="${param.error != null}">
    <p>Invalid username and password.</p>
  </c:if>
  <c:if test="${param.logout != null}">
    <p>You have been logged out.</p>
  </c:if>
  <p>
    <label for="username">Username</label> <input type="text"
      id="username" name="username" />
  </p>
  <p>
    <label for="password">Password</label> <input type="password"
      id="password" name="password" />
  </p>
</form>
```

```

</p>
<input type="hidden" name="${_csrf.parameterName}"
      value="${_csrf.token}" />
<button type="submit" class="btn">Log in</button>
</form>

```

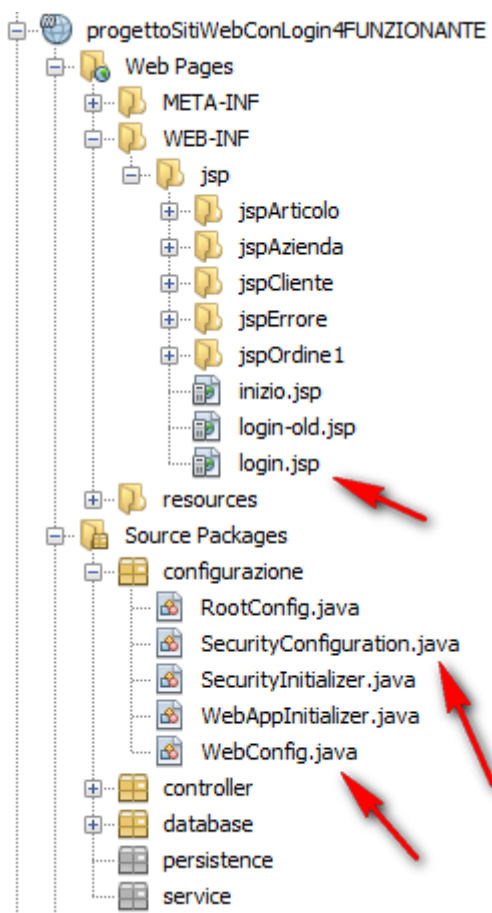
non funziona (cioè non viene effettuato il login) se nella JSP utilizzo la dichiarazione della libreria core:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
```

Soluzione

Poiché nella applicazione **progettoSitiWebConLogin4FUNZIONANTE** mi serve di avere una pagina di login con la libreria core, devo modificare la pagina di login e i file di configurazione di Spring security.

Il seguente schema della applicazione ci dice quali file dovranno essere modificati



jsp login contenente la libreria core di JSTL

login.jsp

```

<!DOCTYPE html >
<%@taglib prefix="form" uri="http://www.springframework.org/tags/form"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<html>
<head>
<style>
h1 {
    font: 2.2em Arial, Helvetica, sans-serif;
}

```

```

input.txt {
    color: #00008B;
    background-color: #e5e5e5;
    border-style: 1px #BLACK;
    width: 200px;
}

/*....altre configurazioni dei CSS ....*/

.divInternoCella { /*uso un div per dare una dimensione MAX a td e th*/
    overflow: hidden;
}
</style>
</head>
<body>
    <div id="banner"></div>
    <div id="content">
        <fieldset>
            <legend>Modulo di Login</legend>
            <div class="etich">Inserire Nome Account & Password:</div>

            <form id="form" action="<c:url value='/login.do'/" method="POST">
                <TABLE WIDTH="80%">
                    <TR>
                        <TD WIDTH="50%" align="right">Username:</TD>
                        <TD WIDTH="50%" align="left"><input type="text" id="username"
                            name="username" /></TD>
                    </TR>
                    <TR>
                        <TD align="right">Password:</TD>
                        <TD align="left"><input type="password" id="password"
                            name="password" /></TD>
                    <TR>
                        <TD></TD>
                        <TD align="left"><INPUT id="b_submit" name="cmdSubmit"
                            TYPE="submit" VALUE="Accedi"> <INPUT id="b_reset"
                            name="cmdReset" TYPE="reset" VALUE="Cancella">
                    </TR>
                </TABLE>

                </FORM>

            </fieldset>

        </div>
        <div id="navigation">
            <li class="titolo">CLIENTE</li>
            <li><a href="inserimento_dati_cliente">Inserimento dati
                cliente</a></li>
            <li><a href="cancella_modifica_cliente_tag">cancella/modifica
                dati cliente</a></li>
            <li><a href="lettura_dati_cliente">Lettura dati clienti </a></li>
            <li class="titolo">ARTICOLO</li>

```

```

<li><a href="inserimento_dati_articolo">Inserimento dati
    articolo</a></li>
<li><a href="cancella_modifica_articolo_tag_1">cancella/modifica
    dati articolo</a></li>
<li class="titolo">AZIENDA</li>
<li><a href="inserimento_dati_azienza">Inserimento dati
    azienda</a></li>
<li><a href="cancella_modifica_azienza_tag">cancella/modifica
    dati azienda</a></li>
<li class="titolo">ORDINE</li>
<li><a href="crea_ordine">Effettua ordine</a></li>
<li><a href="visualizza_ordini">Visualizza ordini</a></li>
</div>
</body>
</html>

```

È importante notare come l'attributo action del form faccia riferimento all'indirizzo *login.do*

```
<form id="form" action="<c:url value='/login.do'/" method="POST">
```

E che i CSS siano stati inseriti direttamente nella pagina di login tramite il tag

```
<style>
</style>
```

classe WebConfig.java: configurazione di un controllore per la pagina di login

nella classe **WebConfig.java** aggiungiamo il seguente codice che permette di puntare alla nostra pagina di login quando l'applicazione viene lanciata:

```

@Override
public void addViewControllers(ViewControllerRegistry registry) {
    registry.addViewController("/login").setViewName("login");
    registry.setOrder(0);
}

```

classe SecurityConfiguration: come modifichiamo il metodo configure(HttpSecurity http) per poter operare con la nuova pagina di login

Per poter utilizzare la pagina di login vista sopra, è necessario modificare il metodo `configure(HttpSecurity http)` nella classe **SecurityConfiguration.java** nel seguente modo:

```

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

        // considera tutti gli url della applicazione
        .antMatchers("/**").authenticated()
        .anyRequest().authenticated()
        .and()

        // pagina di login diversa da quella generata da spring
        .formLogin().loginPage("/login")
        .loginProcessingUrl("/login.do")
}

```

```

        .defaultSuccessUrl("/")
        .failureUrl("/login?err=1")
        .usernameParameter("username")
        .passwordParameter("password").permitAll()
        .and()

        // disabilita controllo csrf
        .csrf().disable();
    }

```

`loginPage("/login")`

ci dice che la pagina di login ha nome logico `/login` (quindi viene mappata con l'indirizzo effettivo `WEB-INF/jsp/login.jsp`)

`loginProcessingUrl("/login.do")`

ci dice che l'URL che porta i dati del login è `login.do` (che è lo stesso URL contenuto nell'attributo `action` del form della JSP che effettua il login).

`defaultSuccessUrl("/")`

/ indirizzo che viene richiamato se il login ha successo.

`failureUrl("/login?err=1")`

`/login?err=1` indirizzo che viene richiamato se il login fallisce; nella mia applicazione non lo utilizzo.

`usernameParameter("username")`

`passwordParameter("password")`

nomi dei parametri utilizzati come username e password.

Perché i CSS sono contenuti nella pagina di login

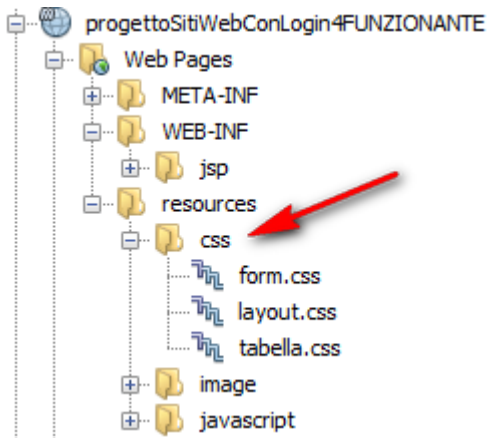
Le pagine JSP della applicazione **progettoSitiWebConLogin4FUNZIONANTE**, fanno riferimento a file CSS esterni ad esse con il seguente codice:

```

<!-- collegamento con i file dei CSS-->
<link href="<c:url value="/resources/css/form.css" />" rel="stylesheet">
<link href="<c:url value="/resources/css/layout.css" />" rel="stylesheet">
<link href="<c:url value="/resources/css/tabella.css" />" rel="stylesheet">

```

I file CSS sono contenuti nella cartella `resources`



In quest'applicazione, spring security, blocca tutti gli indirizzi interni della applicazione finché non viene effettuato un login valido.

In questo modo spring security blocca anche gli indirizzi con cui le JSP fanno riferimento ai file CSS quindi la pagina di login (la cui visualizzazione è permessa da spring security essendo essa necessaria per effettuare il login) viene mostrata senza CSS.

Per mostrare i CSS con la pagina di login è stato quindi necessario inserire il contenuto dei file CSS all'interno della pagina di login.

Nota: il loop di reindirizzamento

in alcuni browser, la seguente configurazione del metodo `configure(HttpSecurity http)` di Spring security ha come effetto di generare un loop infinito, come mostrato nella immagine dopo.

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

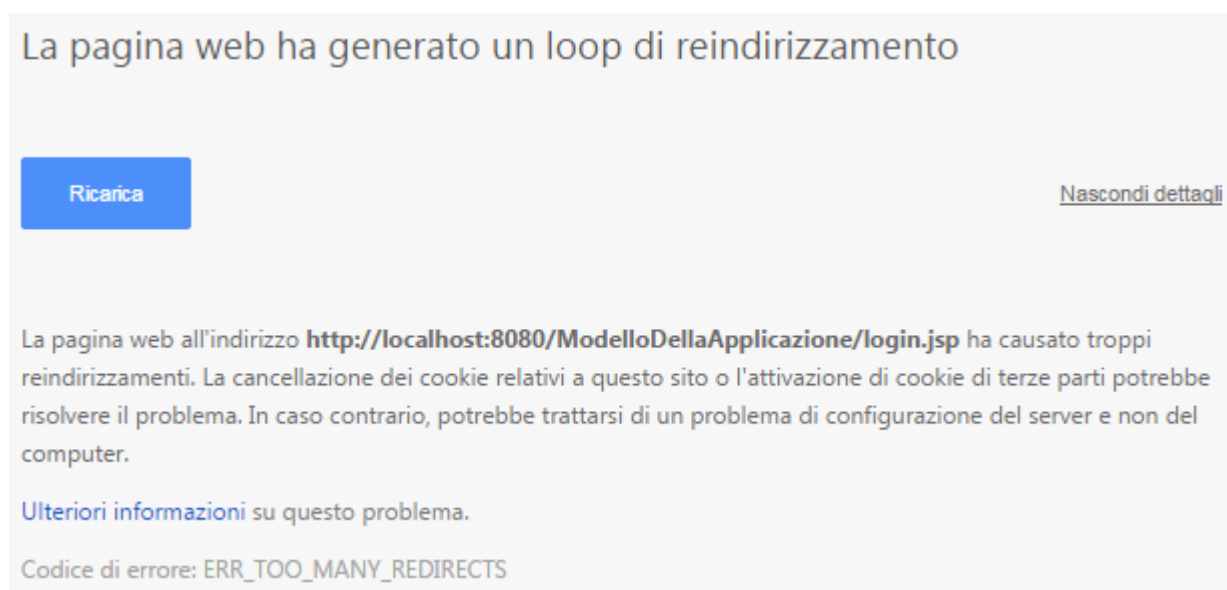
`loginPage("/login")`

questo metodo, quando è richiesta una autenticazione, dice a spring security di re-indirizzare il browser a `/login`

Il problema del loop di re indirizzamento su alcuni browser è causato dalle seguenti operazioni:

- noi facciamo una richiesta alla nostra applicazione Web
- spring security vede che non siamo autenticati quindi ci re-indirizza a `/login` (ovvero i dati inviati dalla nostra applicazione al nostro browser sono quelli di `/login`)
- il browser a seguito del re-indirizzamento ha `/login` nella request
- spring security vede che il browser non è autenticato quindi lo re indirizza a `/login`
- e così via di seguito.

Il risultato è che sul browser compare una pagina come la seguente:



Per risolvere questo problema si aggiunge al metodo `loginPage("/login")` il metodo `permitAll()` ottenendo quindi il seguente codice:


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

il metodo `permitAll()` permette a qualsiasi URL (anche se non autenticato) di accedere alla pagina contenuta nel metodo `loginPage("/login")` e questo evita il loop con il browser.

Ovviamente il metodo `permitAll()` permette a qualsiasi URL di accedere alla pagina `/login` ma solamente ad essa, non al resto delle pagine che formano l'applicazione.

Progetto con Spring senza XML: parte 2

Progetto con il database	1
Esternalizzazione e internalizzazione del progetto	2
cosa si intende con esternalizzazione	2
Come viene effettuata l'esternalizzazione	2
creazione dei file contenente i vari testi	2
modifica del file di configurazione di Spring	3
jsp	4
Spring security: Login con autenticazione	5
pom.xml	5
descrittore dell'applicazione web.xml in java	6
parte di codice del file web.xml che sostituiamo con la configurazione Java	6
SecurityInitializer.java	7
file di configurazione di Spring security che sostituiamo con la configurazione Java	7
SecurityConfiguration.java	8
Come funziona la configurazione di Spring security in XML: elemento <http>	9
Spring recipes 3rd edition pag 331	9
getting started with Spring framework cap 14	9
Come funziona la configurazione di Spring security in XML : Elemento	10
<authentication-manager>	10
Come funziona il codice	11
Nota: CLASSPATH	12
Spring security: pagina di login non generata da Spring 1	12
schema del codice	12
Modifiche al metodo configure() della classe SecurityConfiguration.java	13
La pagina di login	14
login.jsp	14
Cosa accade quando eseguo il codice	15
Come funziona il codice	15
Nota: come funziona il tag <c:url> di JSTL core	16
sintassi	16
Esempio 1: uso dell'attributo value	16
Esempio 2: uso dell'attributo var	17
Esempio 3: uso dell'attributo context	17
Esempio 4: uso dell'attributo scope	18
Spring security: pagina di login non generata da Spring 2	18
problema: non riesco utilizzare la dichiarazione della libreria core di JSTL	18
Soluzione	19
jsp login contenente la libreria core di JSTL	19
login.jsp	19
classe WebConfig.java: configurazione di un controllore per la pagina di login	21

classe SecurityConfiguration: come modifichiamo il metodo configure(HttpSecurity http) per poter operare con la nuova pagina di login _____ 21

Perché i CSS sono contenuti nella pagina di login _____ **22**

Nota: il loop di reindirizzamento _____ **24**