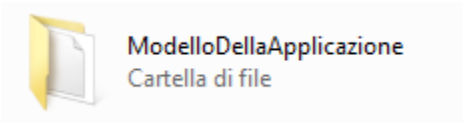
Progetto con Spring senza XML

Progetto

Creare il progetto

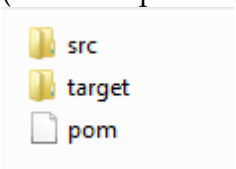
cartella del progetto

Creiamo una nuova cartella di nome `ModelloDellaApplicazione`



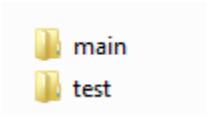
cartelle src target ed il file pom

All'interno della cartella `ModelloDellaApplicazione` creiamo le cartelle src, target ed il file pom (che serve per realizzare un progetto con Maven).



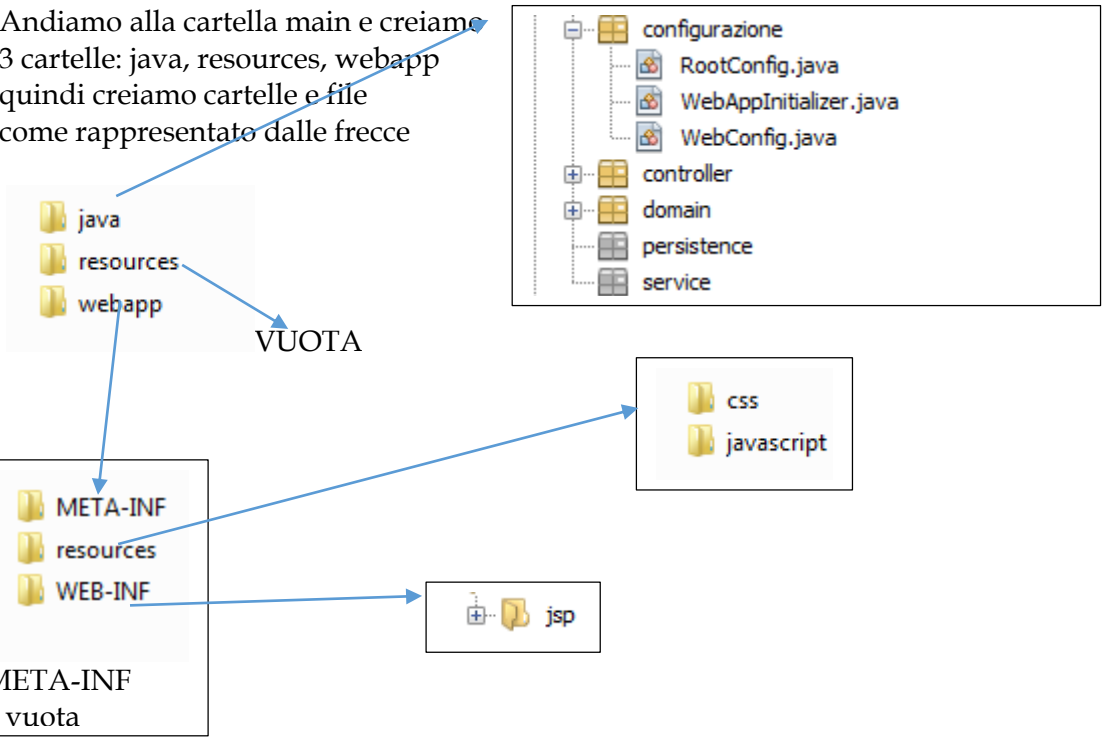
cartelle main e test

Nella cartella src creiamo le cartelle main e test:



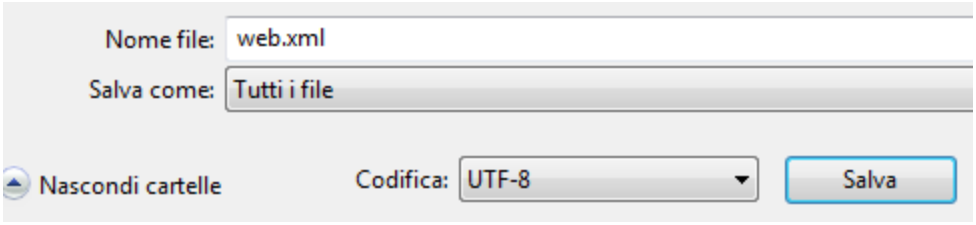
cartelle java, resources, webapp e tutte le altre

Andiamo alla cartella main e creiamo 3 cartelle: java, resources, webapp quindi creiamo cartelle e file come rappresentato dalle frecce



ATTENZIONE:codifica UTF 8 nei file xml

I file xml sono stati creati usando blocco note di windows 7.
Per evitare problemi con Netbeans, è necessario creare i file con codifica UTF 8 invece che con ANSI che è la codifica di default.



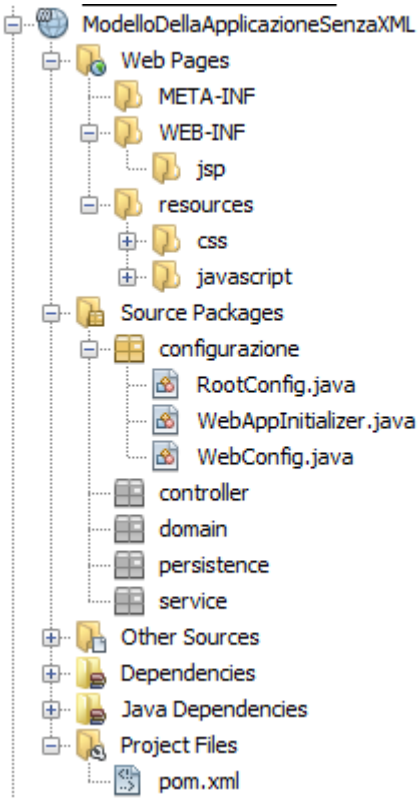
utilizzare il progetto

Per utilizzare il progetto, è sufficiente cambiare `ModelloDellaApplicazione` con il nome del nuovo progetto (ad esempio: `progetto1`) .
Questa sostituzione deve essere fatta:

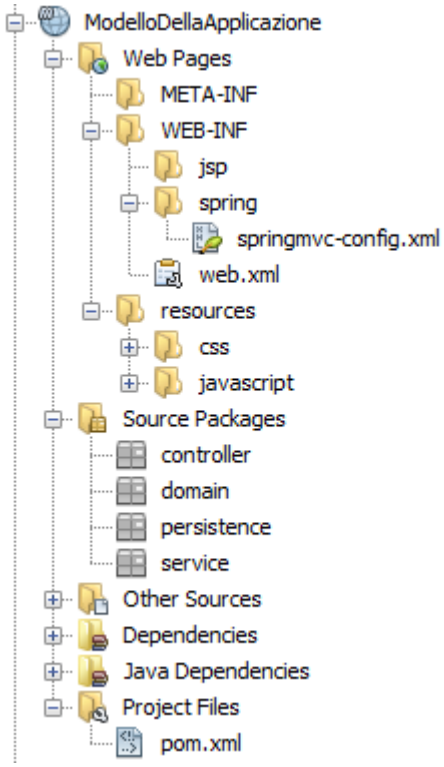
- nella cartella che rappresenta il progetto,
- nel file pom.xml di Maven.

Confronto tra progetto senza XML e quello con XML

progetto senza XML



progetto con XML



File descrittore dell'applicazione web.xml sostituito con una classe Java

Dalla versione 3 di Servlet e 3.1 di Spring, il file descrittore della applicazione (file web.xml contenuto nella directory WEB-INF) può essere sostituito con una classe Java.
È con questa classe Java (che può essere posizionata ovunque nella nostra applicazione) che configuriamo la DispatcherServlet.

Codice da aggiungere a Maven per farlo lavorare senza web.xml

quando usiamo Maven e sostituiamo il file web.xml con una classe Java, è necessario modificare il file pom.xml.
Infatti per default, il plugin war di Maven fallisce se non trova il file web.xml.
Per evitare ciò, bisogna inserire nel file pom.xml di Maven (subito dopo la configurazione iniziale) la seguente parte di codice:

```
<build>
  <plugins>
     <plugin>
         <groupId>org.apache.maven.plugins</groupId>
```

```
            <artifactId>maven-war-plugin</artifactId>
            <configuration>
               <failOnMissingWebXml>false</failOnMissingWebXml>
            </configuration>
         </plugin>
      </plugins>
   </build>
```

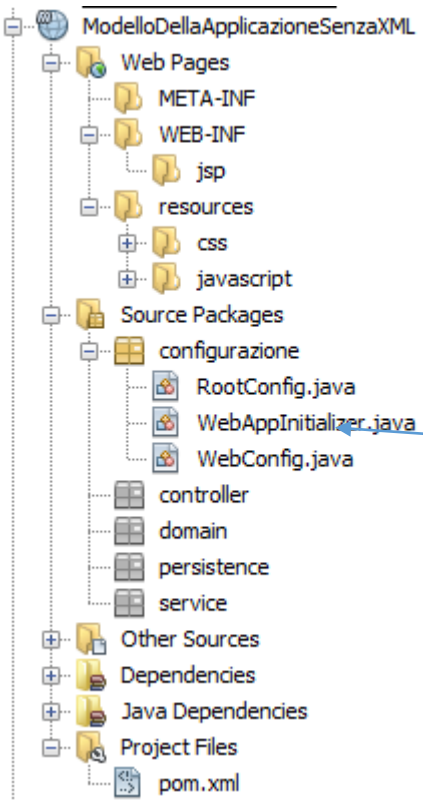
Con questa parte di codice viene posto a `false` l'attributo `<failOnMissingWebXml>` cioè, il plugin war non si blocca se Maven non trova il file web.xml.

Un altro pezzo di codice da aggiungere al file pom è la dipendenza rispetto servlet 3.0. Dobbiamo quindi aggiungere:

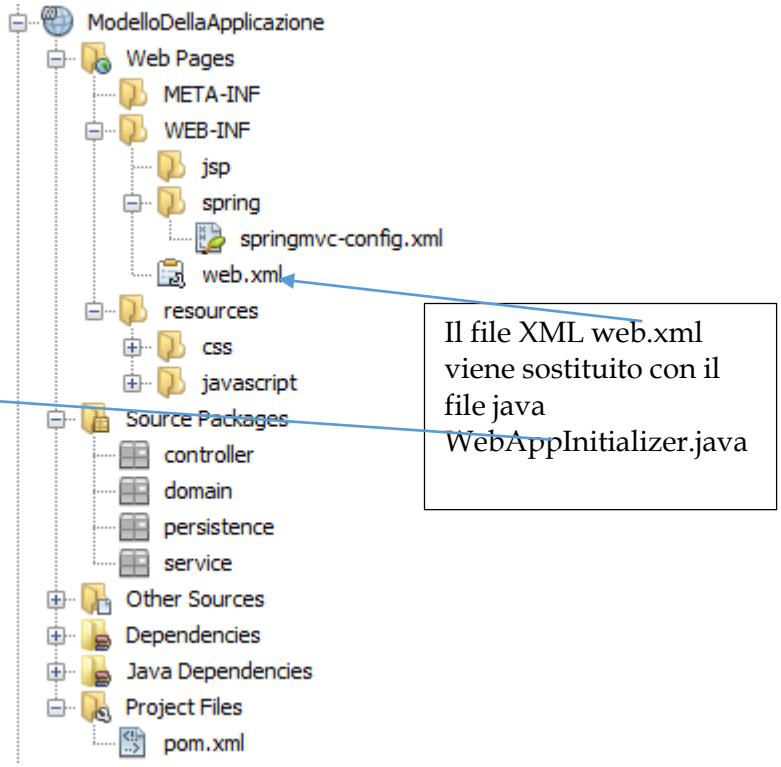
```
<dependency>
   <groupId>javax.servlet</groupId>
   <artifactId>javax.servlet-api</artifactId>
   <version>3.0.1</version>
</dependency>
```

classe WebAppInitializer.java che sostituisce web.xml

progetto senza XML



progetto con XML



Di seguito vediamo il codice del file web.xml che sostituiamo con una classe Java:

web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.0"
      xmlns="http://java.sun.com/xml/ns/javaee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
       xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
                   http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd">

<!-- Configurazione della servlet dispatcher -->
    <servlet>
        <servlet-name>springmvc</servlet-name>
        <servlet-class>
            org.springframework.web.servlet.DispatcherServlet
        </servlet-class>

       <init-param>
         <param-name>contextConfigLocation</param-name>
         <!-- Indirizzo in cui trovo il file di configurazione di Spring -->
         <param-value>/WEB-INF/spring/springmvc-config.xml</param-value>
      </init-param>
        <load-on-startup>1</load-on-startup>
   </servlet>

    <servlet-mapping>
        <servlet-name>springmvc</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>
</web-app>
<!-- L'uso di "/" in url-pattern ci dice che tutte le richieste,
  comprese quelle riferite alle risorse statiche,
  (risorse che non vengono modificate durante l'uso del software
  come ad esempio file CSS o JavaScript) sono dirette alla
  servlet dispatcher quindi, affinché le risorse statiche
  siano gestite correttamente, è necessario aggiungere alcuni elementi
  mvc:resource nel file di configurazione di Spring -->
```

Ora vediamo la classe java che sostituisce il file web.xml.

WebAppInitializer.java

```
package configurazione;

import org.springframework.web.servlet.support.
AbstractAnnotationConfigDispatcherServletInitializer;

public class WebAppInitializer extends
      AbstractAnnotationConfigDispatcherServletInitializer {

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

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

   @Override
   protected String[] getServletMappings() {
```

```
        return new String[] { "/" };
    }
}
```

classe AbstractAnnotationConfigDispatcherServletInitializer
in ambiente Servlet 3.0 il container (contenitore) di Spring, cerca tutte le classi (nel classpath cioè nell'insieme di classi che formano l'applicazione) che implementano l'interfaccia `javax.servlet.ServletContainerInitializer` .
Se le trova, le utilizza per configurare il contenitore delle Servelet (Servelet container).

Spring fornisce un'implementazione della interfaccia `javax.servlet.ServletContainerInitializer` di nome `SpringServletContainerInitializer` che, a sua volta, cerca eventuali classi che implementano `WebApplicationInitializer` a cui delegare la configurazione dell'applicazione Web.
Spring 3.2 fornisce una comoda implementazione di `WebApplicationInitializer` di nome `AbstractAnnotationConfigDispatcherServletInitializer`.

Poiché la nostra classe (`WebInitializer`) estende `AbstractAnnotationConfigDispatcherServletInitializer` (e quindi implementa `WebApplicationInitializer`) essa, quando è impiegata con un Servlet container 3.0, viene automaticamente rilevata ed è utilizzata per configurare il context delle servlet.

metodo `getServletMappings()`
identifica uno o più percorsi in cui sarà mappata la Servlet `DispatcherServlet.`

metodo `getServletConfigClasses()`
quando viene avviata la servlet `DispatcherServlet,` essa crea l'application context di Spring e inizia a caricare i beans dichiarati nei file di configurazione.
Con questo metodo, voi chiedete alla `DispatcherServlet` di caricare il suo application context con i bean definiti nella classe di configurazione `WebConfig.class`.
Anche `WebConfig.class` (che vedremo in seguito), utilizza Java per effettuare le varie configurazione.

Nota: `WebConfig.class` è la classe restituita dal metodo `getServletConfigClasses()` ;
`WebConfig` è un nome arbitrario ma deve riferirsi ad una classe esistente

metodo `getServletConfigClasses()`
in generale in una applicazione Web, oltre all 'application context creato dalla `DispatcherServlet,` esiste un altro application context creato da `ContextLoaderListener.`
La classe di configurazione di questo altro application context è quella restituita dal metodo `getServletConfigClasses().`
Nel nostro caso l'altra classe di configurazione è la classe `RootConfig.class`

NOTA
`WebConfig.class`
contiene le configurazioni dei bean che operano sul Web come:
controllori, view resolver, handler mapping.

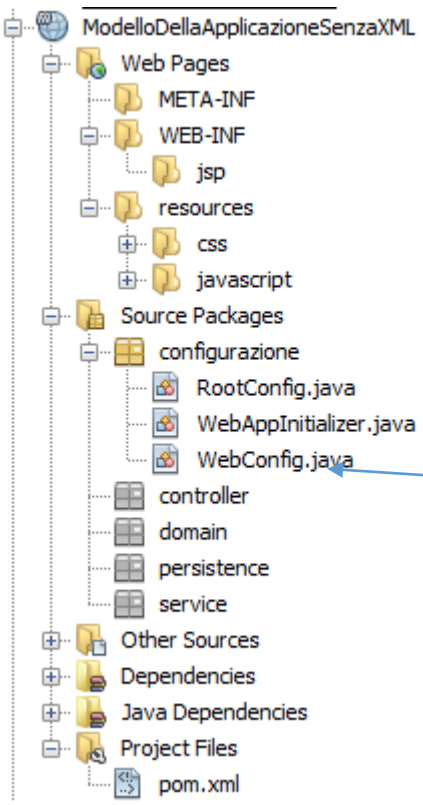
`RootConfig.class`
contiene le configurazioni dei bean relativi al livello dati (data-tier) ed al livello intermedio (middle-tier).

I nomi delle classi sono arbitrari, gli ho scelti perché sono quelli utilizzati dal libro `Spring in action` che ho preso come riferimento.

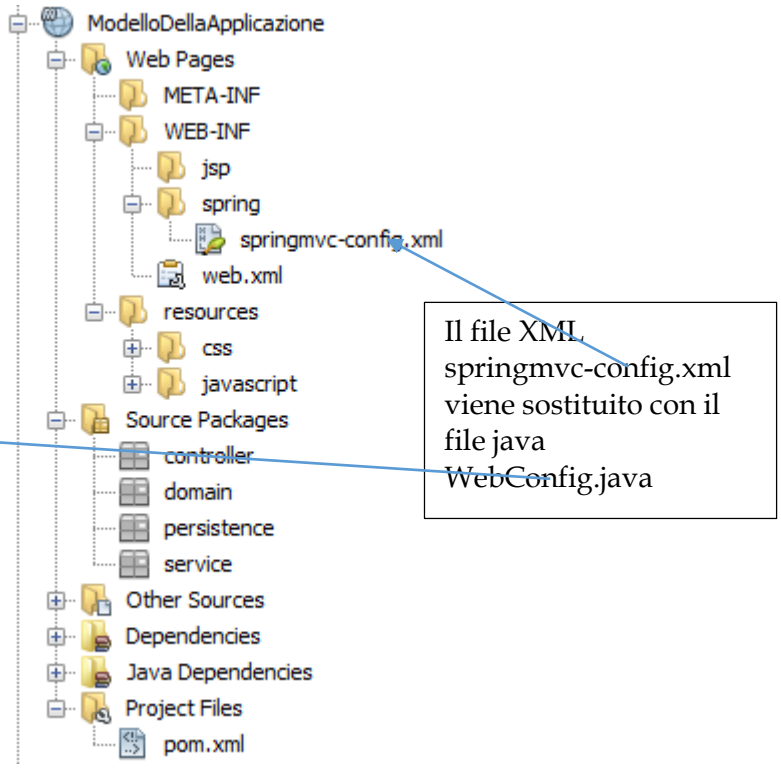
File XML di configurazione di Spring sostituito con una classe Java

Di seguito vediamo il codice del file `springmvc-config.xml` (file di configurazione di Spring) che sostituiamo con una classe Java:

progetto senza XML



progetto con XML



springmvc-config.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:p="http://www.springframework.org/schema/p"
       xmlns:mvc="http://www.springframework.org/schema/mvc"
       xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="
         http://www.springframework.org/schema/beans
         http://www.springframework.org/schema/beans/spring-beans.xsd
         http://www.springframework.org/schema/mvc
         http://www.springframework.org/schema/mvc/spring-mvc.xsd
         http://www.springframework.org/schema/context
         http://www.springframework.org/schema/context/spring-context.xsd">

    <!-- Questo elemento dice a Spring MVC di fare la scansione delle classi
```

```
      contenute in un certo package -->

   <context:component-scan base-package="controller" />

   <!-- Questo elemento registra i bean che supportano l'elaborazione della
           richiesta dei controllori con l'annotazioni e fa altre cose; senza
           questo elemento, gli elementi mvc:resources impediranno a qualsiasi
           controllore di essere chiamato -->

   <mvc:annotation-driven />


   <!-- Questo elemento ci dice quali risorse statiche (CSS, JavaScript,
       HTML, Pdf) devono essere servite in maniera indipendente rispetto alla
       dispatcher servlet.
       con la notazione scritta sotto, Spring trasforma l'URL
       /resources/css/main.css in /resources/css/main.css .
       Se avessi avuto:
       mapping="/resources/**" location="/resources/mytheme/"
    Spring avrebbe trasformato l'URL /resources/css/main.css in
       /resources/mytheme/css/main.css
       cioè avrebbe sostituito =/resources/ con "/resources/mytheme/ -->

   <mvc:resources mapping="/resources/**" location="/resources/" />

   <!-- view resolver -->
   <bean id="viewResolver" class="org.springframework.web.servlet.view.
                        InternalResourceViewResolver">
      <property name="prefix" value="/WEB-INF/jsp/" />
      <property name="suffix" value=".jsp" />
   </bean>
</beans>
```

Ora vediamo la classe java che sostituisce il file xml di configurazione di Spring.

WebConfig.java (WebConfig.class)

```
package configurazione;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.ViewResolver;
import org.springframework.web.servlet.config.annotation.
DefaultServletHandlerConfigurer;
import org.springframework.web.servlet.config.annotation.EnableWebMvc;
import org.springframework.web.servlet.config.annotation.
ResourceHandlerRegistry;
import org.springframework.web.servlet.config.annotation.
WebMvcConfigurerAdapter;
import org.springframework.web.servlet.view.InternalResourceViewResolver;

@Configuration
@EnableWebMvc
@ComponentScan("controllori")
public class WebConfig extends WebMvcConfigurerAdapter {
```

```
  @Bean
  public ViewResolver viewResolver() {
    InternalResourceViewResolver resolver = new InternalResourceViewResolver();
    resolver.setPrefix("/WEB-INF/views/");
    resolver.setSuffix(".jsp");
    return resolver;
  }

  @Override
  public void configureDefaultServletHandling(DefaultServletHandlerConfigurer
                                                                    configurer) {
    configurer.enable();
  }

}
```

`@Configuration`
indica al framework che la classe può essere usata dallo IoC Container di Spring come una sorgente di definizioni di bean.

`@EnableWebMvc`
permette di abilitare SpringMVC.
Equivale alla istruzione XML `<mvc:annotation-driven>`

`@ComponentScan("controllori")`
effettua una scansione del package controllori alla ricerca di classi annotate con @Component.
Nota: l'annotazione `@Controller` è un caso particolare di `@Component`.
Equivale a `<context:component-scan/>`

`@Bean`
si applica ai metodi; dice a Spring che il metodo ritorna un oggetto che deve essere registrato come bean nell'application context di Spring.
Nell'esempio precedente, l'annotazione @Bean sul metodo `viewResolver()` aggiunge il bean viewResolver all'application context di Spring.
Se non è specificato diversamente, il nome del metodo è il nome con cui il bean viene aggiunto all'application context.

`configureDefaultServletHandling ()`
La nostra classe estende la classe `WebMvcConfigurerAdapter.`
Il metodo `configureDefaultServletHandling()` va a sovrascrivere lo stesso metodo presente nella classe `WebMvcConfigurerAdapter.`

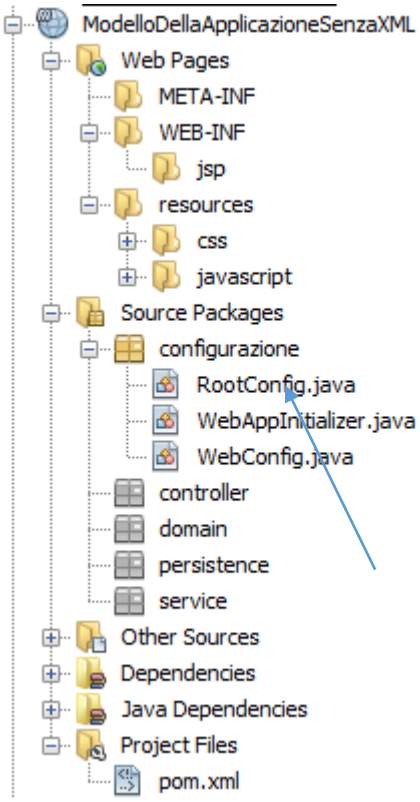
`configurer.enable()`
`configurer` è un oggetto che viene passato al metodo ConfigureDefaultServletHandling() da Spring.
Il metodo enable() richiamato su `configurer`, chiede alla DispatcherServlet di inoltrare le richieste, relative alle risorse statiche, al contenitore di default delle servlet e di non provare a gestirle essa stessa.

Classe RootConfig.java

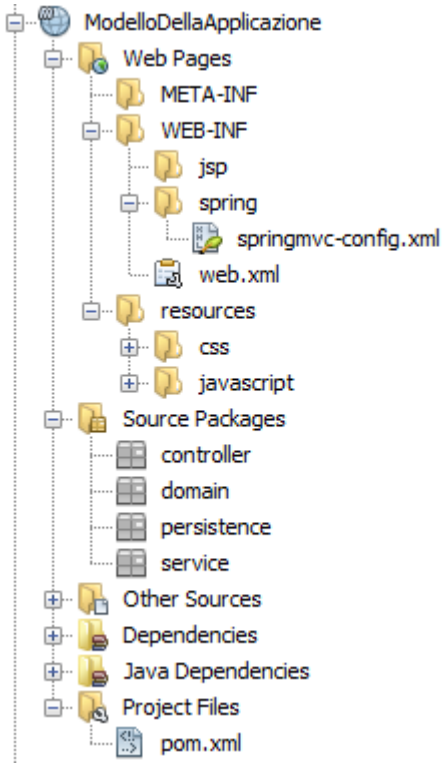
Questa classe, non ha un corrispettivo nel progetto senza xml e al momento non viene utilizzata; quindi è una classe vuota creata perché nel file WebAppInitializer.java c'è una parte del codice che fa riferimento ad essa:

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

progetto senza XML



progetto con XML



il codice della classe RootConfig.java è il seguente

RootConfig.java

```
package configurazione;

import org.springframework.context.annotation.Configuration;

@Configuration

public class RootConfig {
}
```

Progetto con Spring senza XML