

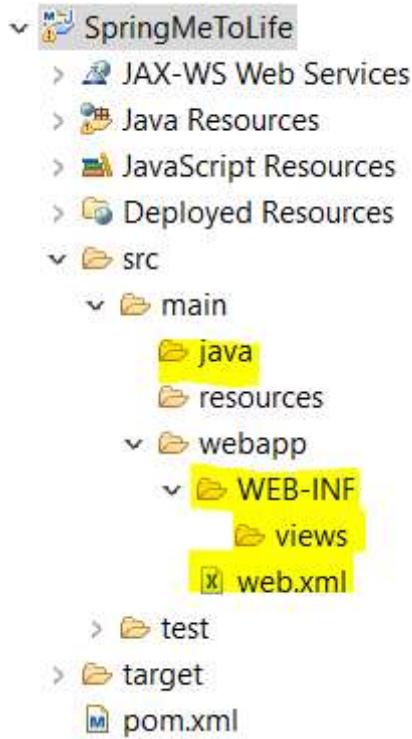
Setup nuova web application con Spring usando Maven senza Archetype

Con Eclipse Neon per EE developer proviamo a generare un nuovo progetto che utilizzi Spring web MVC e Spring REST:

File > Nuovo Progetto Maven > Skip Archetype (Create simple project)

NOTA: Quando creiamo nuovi file, folder, o altro, usiamo la cartella **src e non la cartella **JavaResources>src**.**

Dopo generiamo le seguenti cartelle (evidenziate in giallo):



Fatto questo aggiustiamo il pom e il web xml:

pom.xml:

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>it.euris.example</groupId>
<artifactId>SpringMeToLife</artifactId>
<version>0.0.1-SNAPSHOT</version>
<packaging>war</packaging>

<properties>
    <java-version>1.7</java-version>
    <springframework.version>4.3.1.RELEASE</springframework.version>
    <jackson.version>2.7.5</jackson.version>
</properties>

<dependencies>
    <dependency>
        <groupId>javax</groupId>
        <artifactId>javaee-web-api</artifactId>
        <version>7.0</version>
        <scope>provided</scope>
    </dependency>
    <dependency>
        <groupId>javax.servlet</groupId>
        <artifactId>jstl</artifactId>
        <version>1.2</version>
    </dependency>
</dependencies>

<build>
    <finalName>SpringMeToLife</finalName>
    <pluginManagement>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
```

```
<version>2.3.2</version>

<configuration>
    <source>${java-version}</source>
    <target>${java-version}</target>
</configuration>

</plugin>

<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-war-plugin</artifactId>
    <version>2.4</version>
    <configuration>
        <warSourceDirectory>src/main/webapp</warSourceDirectory>
        <warName>SpringMeToLife</warName>
        <failOnMissingWebXml>false</failOnMissingWebXml>
    </configuration>
</plugin>
</plugins>
</pluginManagement>
</build>

</project>
```

web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app id="WebApp_ID" version="2.4"
    xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
    http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">

    <display-name>Spring me to life!</display-name>

</web-app>
```

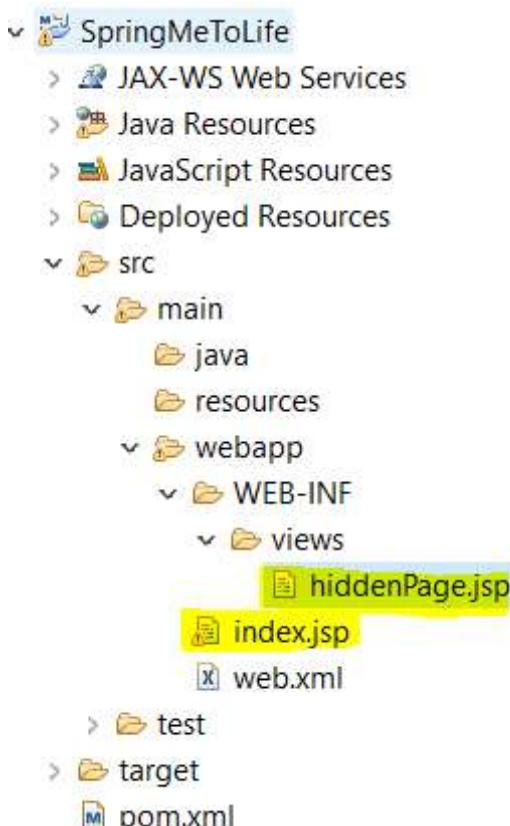
Clicco col destro sull'albero del progetto e vado in Proprietà > Java Compiler e setto 1.7.

Poi vado in Server Facets e setto Tomcat 7 e 9.

Poi faccio salva, refresh e Maven > Update Project.

E siamo pronti per iniziare!

Creiamo delle pagine JSP per la visualizzazione di dati: una pubblica in webapp e una nascosta in webapp>WEB-INF>views



Con la seguente struttura (facendo attenzione che il linguaggio EL sia abilitato grazie all'apposito TAG (in rosso)):

```

<%@ page language="java" contentType="text/html; charset=ISO-8859-1" pageEncoding="ISO-8859-1"%>
<%@ page isELIgnored="false" %>
<!DOCTYPE html>
<html>
<head>

```

```
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Hidden Page</title>
</head>
<body>
<p> Hidden Page </p>
</body>
</html>
```

Creiamo una servlet classica prima di inserire Spring, così da confrontare i due approcci.

andiamo su main/java e clicchiamo col destro. Facciamo New >Other>Java Class.

Si apre la finestra di creazione della classe Java e inseriamo il package: it.euris.example.servlets e la Classe la chiamiamo MyClassicServlet:

```
package it.euris.example.servlets;

import java.io.IOException;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@WebServlet(
    name = "myClassicServlet",
    urlPatterns = {" /ciao", " /salve" },
    loadOnStartup = 1
)
public class MyClassicServlet extends HttpServlet{

    @Override
    public void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException,
    IOException{

    }
}
```

```
@Override

    public void doPost(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException{
        this doGet(request, response);
    }

}
```

Adesso creiamo un'entità cliccando sulla cartella main>java e facendo new >other>Java Class:

```
package: it.euris.example.models
name: User.
```

```
package it.euris.example.models;

public class User {

    private String name;
    private String surname;

    public User(String _name, String _surname){
        this.name = _name;
        this.surname = _surname;
    }

    public User(){}
}

public String getName() {
    return name;
}

public void setName(String name) {
    this.name = name;
}

public String getSurname() {
```

```
        return surname;  
    }  
  
    public void setSurname(String surname) {  
        this.surname = surname;  
    }  
}
```

Adesso usiamo questa classe nelle servlet e rimandiamo i dati alle pagine JSP:

```
public class MyClassicServlet extends HttpServlet{  
  
    @Override  
    public void doGet(HttpServletRequest request, HttpServletResponse response) throws  
    ServletException, IOException{  
        User myUser = new User("Armando", "Donzelli");  
        request.setAttribute("mioUtente", myUser);  
        request.getRequestDispatcher("/WEB-INF/views/hiddenPage.jsp").forward(request, response);  
    }  
  
    @Override  
    public void doPost(HttpServletRequest request, HttpServletResponse response) throws  
    ServletException, IOException{  
        this.doGet(request, response);  
    }  
}
```

E inseriamo il parametro appena passato alla pagina hiddenPage.jsp:

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1" pageEncoding="ISO-8859-1"%>
```

```
<%@ page isELIgnored="false" %>
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
        <title>Hidden Page</title>
    </head>
    <body>
        <p> Hidden Page </p>
        <p> ${mioUtente.name} </p>
    </body>
</html>
```

L'Expression Language nella pagina JSP va a recuperare direttamente i dati dalla request attraverso le variabili implicite.

Non c'è bisogno di nominare la request oppure neanche di usare il getName() dell'entità User, e neanche di importare User.

Testiamo il sistema effettuando l'esport cliccando col destro sull'albero del progetto e facendo: Export > WAR file.

Prendo il WAR file generato e lo inserisco nella cartella webapp di Tomcat. Avvio Tomcat da bin/startup e l'applicazione viene deployata e posso andare da Browser a testarla con i seguenti url:

```
http://localhost:8080/SpringMeToLife/ciao
http://localhost:8080/SpringMeToLife/salve
```

AGGIUNGIAMO SPRING

Spring utilizza una Servlet che il programmatore non deve creare ex-novo, perchè già presente nel framework.

Questa servlet, come qualsiasi servlet intercetta le richieste HTTP e le gestisce. Nel caso di Spring le intercetta e le invia ad un Controller, una classe java che smista le richieste in base all'url e ai parametri e compie delle azioni o chiama dei "beans".

I beans sono delle entità di Java che invece di istanziare noi, vengono istanziati dal framework (attraverso l'ApplicationContext) e vengono soltanto iniettati da noi dove servono, utilizzando l'annotation @Autowire Object nome_istanza;

Tali beans sono però elencati o in un file xml, o nel nostro caso nell'equivalente (e sostitutivo) file java di configurazione, annotato con @Configuration e in cui sono presenti i metodi/bean con l'annotation: @Bean public Object nome_istanza(){};

La DispatcherServlet di Spring non è da noi modificabile: non potendo aggiungere l'annotation che la fa istanziare dal ServletContainer, usiamo una classe apposita che si chiama DispatcherInitializer, che effettivamente fa partire non solo la DispatcherServlet ma, con essa, tutto il Framework di Spring. In alternativa la DispatcherServlet può essere istanziata dal web.xml.

Detto questo possiamo inserire il framework Spring. Faremo partire la DispatcherServlet da DispatcherInitializer.java (e non da web.xml) e useremo la configurazione dei beans da java (e non da file xml).

Però prima di tutto questo dobbiamo aggiornare le dipendenze del **pom.xml di maven**:

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

    <modelVersion>4.0.0</modelVersion>

    <groupId>it.euris.example</groupId>

    <artifactId>SpringMeToLife</artifactId>

    <version>0.0.1-SNAPSHOT</version>

    <packaging>war</packaging>

    <properties>
        <java-version>1.7</java-version>
        <springframework.version>4.3.1.RELEASE</springframework.version>
        <jackson.version>2.7.5</jackson.version>
    </properties>

    <dependencies>
        <!-- J2EE Classic -->
        <dependency>
            <groupId>javax</groupId>
            <artifactId>javaee-web-api</artifactId>
            <version>7.0</version>
        
```

```
<scope>provided</scope>

</dependency>

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>jstl</artifactId>

<version>1.2</version>

</dependency>

<!-- Spring -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-core</artifactId>

<version>${springframework.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-web</artifactId>

<version>${springframework.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-webmvc</artifactId>

<version>${springframework.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-tx</artifactId>

<version>${springframework.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-orm</artifactId>

<version>${springframework.version}</version>

</dependency>

</dependencies>
```

```

<build>

    <finalName>SpringMeToLife</finalName>

    <pluginManagement>
        <plugins>

            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>2.3.2</version>
                <configuration>
                    <source>${java-version}</source>
                    <target>${java-version}</target>
                </configuration>
            </plugin>

            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-war-plugin</artifactId>
                <version>2.4</version>
                <configuration>
                    <warSourceDirectory>src/main/webapp</warSourceDirectory>
                    <warName>SpringMeToLife</warName>
                    <failOnMissingWebXml>false</failOnMissingWebXml>
                </configuration>
            </plugin>
        </plugins>
    </pluginManagement>

</build>

</project>

```

procediamo ad aggiornare le dipendenze di progetto dopo aver salvato le modifiche del pom.xml con **Maven>Update Project** cliccando col destro sul nome del progetto.

Adesso creiamo il file di Configurazione Java contentente l'istanziazione dei vari Java Beans che Spring andrà ad iniettare.

Facciamo main>java> new Class:

package: it.euris.example.configurations

class: SpringConfiguration.java

```
package it.euris.example.configurations;

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.WebMvcConfigurerAdapter;
import org.springframework.web.servlet.view.InternalResourceViewResolver;
import org.springframework.web.servlet.view.JstlView;

import it.euris.example.models.User;

/**Classe che istanzia i vari Beans. Si definisce un package in cui Spring deve scansionare per trovare le classi entità dalle quali creare i Bean qui definiti **/
```

```
@Configuration
@EnableWebMvc
@ComponentScan(basePackages = "it.euris.example")

public class SpringConfiguration extends WebMvcConfigurerAdapter{
```

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

```
/**Questo BEAN si occupa di collegare il nome logico nel Controller ad una View esistente: NON DOBBIAMO INIETTARLO NOI**/

@Bean
public ViewResolver viewResolver() {
```

```

InternalResourceViewResolver viewResolver = new InternalResourceViewResolver();

viewResolver.setViewClass(JstlView.class);

viewResolver.setPrefix("/WEB-INF/views/");

viewResolver.setSuffix(".jsp");

return viewResolver;

}

/** INIZIO DEFINIZIONE DEI BEANS CHE CI INTERESSANO **/


@Bean

public User mioUtente(){

    return new User();

}

@Bean

public User luciaUser(){

    return new User("Lucia", "Distante");

}

}

```

Adesso andiamo a far partire la DispatcherServlet di Spring (non possiamo usare la annotation sulla definizione della classe DispatcherServlet purtroppo), quindi facciamo partire questa servlet speciale con questa classe.

Facciamo main>java> new Class:

```

package: it.euris.example.configurations

class:      DispatcherInitializer

```

```

package it.euris.example.configurations;

import javax.servlet.ServletContext;
import javax.servlet.ServletException;
import javax.servlet.ServletRegistration;

import org.springframework.web.WebApplicationInitializer;

```

```

import org.springframework.web.context.support.AnnotationConfigWebApplicationContext;
import org.springframework.web.servlet.DispatcherServlet;

/**Classe che si occupa di inizializzare la Dispatcher Servlet di Spring senza dover usare il Deployment
Descriptor*/
public class DispatcherInitializer implements WebApplicationInitializer {

    public void onStartup(ServletContext container) throws ServletException {
        AnnotationConfigWebApplicationContext ctx = new AnnotationConfigWebApplicationContext();
        ctx.register(SpringConfiguration.class);
        ctx.setServletContext(container);

        ServletRegistration.Dynamic servlet = container.addServlet("dispatcher", new DispatcherServlet(ctx));

        servlet.setLoadOnStartup(1);
        servlet.addMapping("/");
    }
}

```

Adesso manca da definire il Controller di Spring, una classe annotata a cui la DispatcherServlet va ad inoltrare tutte le richieste http a seconda del request mapping, ovvero dell'url della richiesta.

Creiamo una nuova classe in: main>java> new Class:

```

package: it.euris.example.controllers
class:      SpringController.java

```

```

package it.euris.example.controllers;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;

```

```

import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.servlet.ModelAndView;

import it.euris.example.models.User;

@Controller
public class SpringController {

    @Autowired
    private User mioUtente;

    @Autowired
    public User luciaUser;

    @RequestMapping(value={"/ciaoSpring", "/salveSpring"}, method=RequestMethod.GET)
    public String goToPage(){
        return "hiddenPage";
    }

    @RequestMapping(value={"/ciaoSpringUser", "/salveSpringUser"}, method=RequestMethod.GET)
    public ModelAndView goToPage(@RequestParam("name") String nome, @RequestParam("surname") String cognome){
        //istanza/bean iniettata
        mioUtente.setName(nome);
        mioUtente.setSurname(cognome);

        ModelAndView modAndView = new ModelAndView("hiddenPage"); //pagina jsp associata al modello
        modAndView.addObject("mioUtente", mioUtente); //aggiungiamo i dati col nome
        mioUtente

        return modAndView;
    }

    @RequestMapping(value={"/tuuts"}, method=RequestMethod.GET)
    public ModelAndView goToLucia(){
        ModelAndView modAndView = new ModelAndView("hiddenPage"); //pagina jsp associata al
        modello
    }
}

```

```

        modAndView.addObject("mioUtente", luciaUser);
                                            //aggiungiamo i dati col nome
        mioUtente

        return modAndView;

    }

}

```

Ricordiamo che la pagina jsp in main>webapp>WEB-INF>views>hiddenPage.jsp

```

<%@ page language="java" contentType="text/html; charset=ISO-8859-1" pageEncoding="ISO-8859-1"%>
<%@ page isELIgnored="false" %>
<!DOCTYPE html>
<html>

    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
        <title>Hidden Page</title>
    </head>
    <body>
        <p> Hidden Page </p>
        <p> ${mioUtente.name} </p>
    </body>
</html>

```

Le chiamate a dover esser provate sono:

```

http://localhost:8080/SpringMeToLife/ciaoSpringUser?name=Andrea&surname=Fornaro
http://localhost:8080/SpringMeToLife/tuuts

```

AGGIUNGIAMO I SERVIZI REST

Se volessimo fare in modo da garantire anche dei servizi REST secondo il paradigma tipico dei Web Service RESTful, utilizzando Spring e accedendo alle entità attraverso delle chiamate REST ed ottenendo dei messaggi JSON di risposta invece di una pagina HTML dobbiamo fare una serie di modifiche alla nostra applicazione.

1. Aggiungiamo la dipendenza nel **pom.xml** di **Jackson Library**, una libreria comodissima che mi trasforma un oggetto Java (la classica entità che vogliamo comunicare all'utente) in un oggetto JSON creando un mapping.

Questo ci eviterà di costruire manualmente un messaggio JSON di risposta che contenga tutti i dati presenti nell'oggetto Java.

nel pom.xml

```
<!-- Jackson Library -->
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
    <version>${jackson.version}</version>
</dependency>
```

La **jackson.version** l'abbiamo specificata nelle **properties** all'inizio dello stesso file pom.xml.

2. Modifichiamo la DispatcherServletInitializer.java prima definita con gli elementi in blu:

```
package it.euris.example.configurations;

import javax.servlet.ServletContext;
import javax.servlet.ServletException;
import javax.servlet.ServletRegistration;

import org.springframework.web.WebApplicationInitializer;
import org.springframework.web.context.support.AnnotationConfigWebApplicationContext;
import org.springframework.web.servlet.DispatcherServlet;
import
org.springframework.web.servlet.support.AbstractAnnotationConfigDispatcherServletInitializer;

/**Classe che si occupa di inizializzare la Dispatcher Servlet di Spring senza dover usare il Deployment Descriptor**/
```

```
public class DispatcherInitializer extends AbstractAnnotationConfigDispatcherServletInitializer
implements WebApplicationInitializer {  
  
    /** Override dell'interfaccia WebApplicationInitializer usata per Spring web mvc **/  
  
    @Override  
    public void onStartup(ServletContext container) throws ServletException {  
        AnnotationConfigWebApplicationContext ctx = new AnnotationConfigWebApplicationContext();  
        ctx.register(SpringConfiguration.class);  
        ctx.setServletContext(container);  
        ServletRegistration.Dynamic servlet = container.addServlet("dispatcher", new  
        DispatcherServlet(ctx));  
        servlet.setLoadOnStartup(1);  
        servlet.addMapping("/");  
    }  
  
    /** Override dei metodi di AbstractAnnotationConfigDispatcherServlet per i servizi REST **/  
  
    @Override  
    protected Class[] getRootConfigClasses() {  
        return new Class[] { SpringConfiguration.class };  
    }  
  
    @Override  
    protected Class[] getServletConfigClasses() {  
        return null;  
    }  
  
    @Override  
    protected String[] getServletMappings() {  
        return new String[] { "/" };  
    }  
}
```

3. Adesso creiamo una classe di accesso ad una lista di entità di tipo User. Simulerà l'accesso ad un database, la query e la lista di risultati. La chiamiamo UserDao.java dove DAO sta per Data Access Object, un oggetto Java contenente vari metodi che recuperano le entità di tipo Utente e la restituiscono in qualche modo, nel nostro caso un ArrayList.

Andiamo in main > java > new Class:

```
package: it.euris.example.dao  
class: UserDao.java
```

```
package it.euris.example.dao;  
  
import java.util.ArrayList;  
import java.util.List;  
import org.springframework.stereotype.Component;  
  
import it.euris.example.models.User;
```

@Component

```
public class UserDao {  
  
    private static List<User> userslist = new ArrayList<User>();  
  
    /** COSTRUTTORE: riempie la lista statica di elementi, come se fossero acquisiti da un DB**/  
    public UserDao(){  
        userslist.add(new User("Mino", "Faita"));  
        userslist.add(new User("Francesco", "Giannotta"));  
        userslist.add(new User("Luciana", "Pellizzati"));  
    }  
  
    /*-----*/
```

```
public List list() {
    return userslist;
}

/*
-----*/



public User get(String name) {
    for (User u : userslist) {
        if (u.getName().equals(name)) {
            return u;
        }
    }
    return null;
}

/*
-----*/



public User create(User user) {
    userslist.add(user);
    return user;
}

/*
-----*/



public String delete(String name) {
    for (User u : userslist) {
        if (u.getName().equals(name)) {
            userslist.remove(u);
            return name;
        }
    }
}
```

```

        }

        return null;

    }

    /*-----*/
}

public User update(String name, User user) {
    for (User u : userslist) {
        if (u.getName().equals(name)) {
            user.setName(u.getName());
            userslist.remove(u);
            userslist.add(user);
            return user;
        }
    }
    return null;
}

/*-----*/
}

```

Questa classe, segnalata a Spring con l'annotation **@Component**, non va istanziata nel file java delle configurazioni di Spring, dove sono istanziati gli altri Bean.

Adesso creiamo un nuovo Controller di Spring che come il precedente si occuperà di ricevere le richieste HTTP inoltrate dalla DispatcherServlet. Questo Controller è un RestController perchè si occupa dei servizi REST.

Andiamo in main > java > new Class:

```

package:      it.euris.example.controllers
class:        SpringRestController.java

```

```
package it.euris.example.controllers;
```

```
import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.DeleteMapping;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.PutMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.ResponseBody;
import org.springframework.web.bind.annotation.RestController;

import it.euris.example.dao.UserDAO;
import it.euris.example.models.User;
```

@RestController

```
public class SpringRestController {
```

```
    @Autowired
```

```
    private UserDAO userDAO;
```

```
//usato solo per "/esempio" e non perchè sto usando userDAO
```

```
    @Autowired
```

```
    private User luciaUser;
```

```
//esempio di metodo che non utilizza JacksonLibrary: componiamo noi il messaggio JSON
```

```
    @RequestMapping(value="/esempio", method=RequestMethod.GET)
```

```
    @ResponseBody
```

```
    public String esempio(){
```

```
String esempioJSON = "[";
esempioJSON = esempioJSON + "name' : '" + luciaUser.getName() + "', ";
esempioJSON = esempioJSON + "surname' : '" + luciaUser.getSurname() + "'}]";
return esempioJSON;
}

@GetMapping("/users")
public List getUsers() {
    return userDAO.list();
}

@GetMapping("/users/{name}")
public ResponseEntity getCustomer(@PathVariable("name") String name) {
    User user = userDAO.get(name);
    if (user == null) {
        return new ResponseEntity("No User found for name " + name, HttpStatus.NOT_FOUND);
    }
    return new ResponseEntity(user, HttpStatus.OK);
}

@PostMapping(value = "/users")
public ResponseEntity createCustomer(@RequestBody User user) {
    userDAO.create(user);
    return new ResponseEntity(user, HttpStatus.OK);
}

@DeleteMapping("/users/{name}")
public ResponseEntity deleteUser(@PathVariable String name) {
    if (null == userDAO.delete(name)) {
        return new ResponseEntity("No User found for name " + name, HttpStatus.NOT_FOUND);
    }
    return new ResponseEntity(name, HttpStatus.OK);
}
```

```
@PutMapping("/users/{name}")  
public ResponseEntity updateUser(@PathVariable String name, @RequestBody User user) {  
    user = userDAO.update(name, user);  
    if (null == user) {  
        return new ResponseEntity("No User found for name " + name, HttpStatus.NOT_FOUND);  
    }  
    return new ResponseEntity(user, HttpStatus.OK);  
}  
}
```

testiamo il servizio con i seguenti url:

```
http://localhost:8080/SpringMeToLife/esempio  
http://localhost:8080/SpringMeToLife/users  
http://localhost:8080/SpringMeToLife/users/Luciana
```

I restanti test si possono fare con **PostMan**, il plugin di Chrome che permette di fare delle richieste anche diverse dalle richieste GET, quindi anche POST, PUT, DELETE.