

Guida allo sviluppo

Restful web service Java mediante l'utilizzo di Spring-boot

Carlo Lomello & Francesco Manzo

Introduzione

Cos'è Spring boot

Spring è un framework open source creato con l'intento di gestire la complessità nello sviluppo di applicazioni enterprise ma prevede una serie di configurazioni che molto spesso risultano essere tediose e complicate da impostare.

Spring Boot è una soluzione per il framework di Java che definisce una configurazione di base ed include le linee guida per l'uso del framework e tutte le librerie di terze parti rilevanti, rendendo quindi l'avvio di nuovi progetti il più semplice possibile.

Offre innumerevoli vantaggi, tra cui la configurazione Maven semplificata grazie all'utilizzo dei POM "Starter" (Project Object Models) e la configurazione automatica di Spring, dove vi è possibile.

Obiettivo

Per dimostrare la semplicità e la comodità nell'utilizzo di Spring boot, andremo a sviluppare una serie di microservizi REST che interagiscono con un database per effettuare le operazioni tipiche di CRUD.

Requisiti

Ambiente di sviluppo ed application server

Gli strumenti che noi andremo ad utilizzare per lo sviluppo del nostro restful web service sono:

- Eclipse;
- XAMPP (mySQL);
- Wildfly
- Postman;
- Integrazione con il framework SWAGGER 2.0;

Step 1

XAMPP e lo sviluppo del nostro Database MySQL

XAMPP è una distribuzione di Apache completamente gratuita e semplice da installare, contenente MySQL, PHP e Perl. Il pacchetto open source XAMPP è stato creato per essere estremamente facile da installare e utilizzare.

Andiamo a scaricare ed installare l'ultima versione di XAMPP da [qui](#).

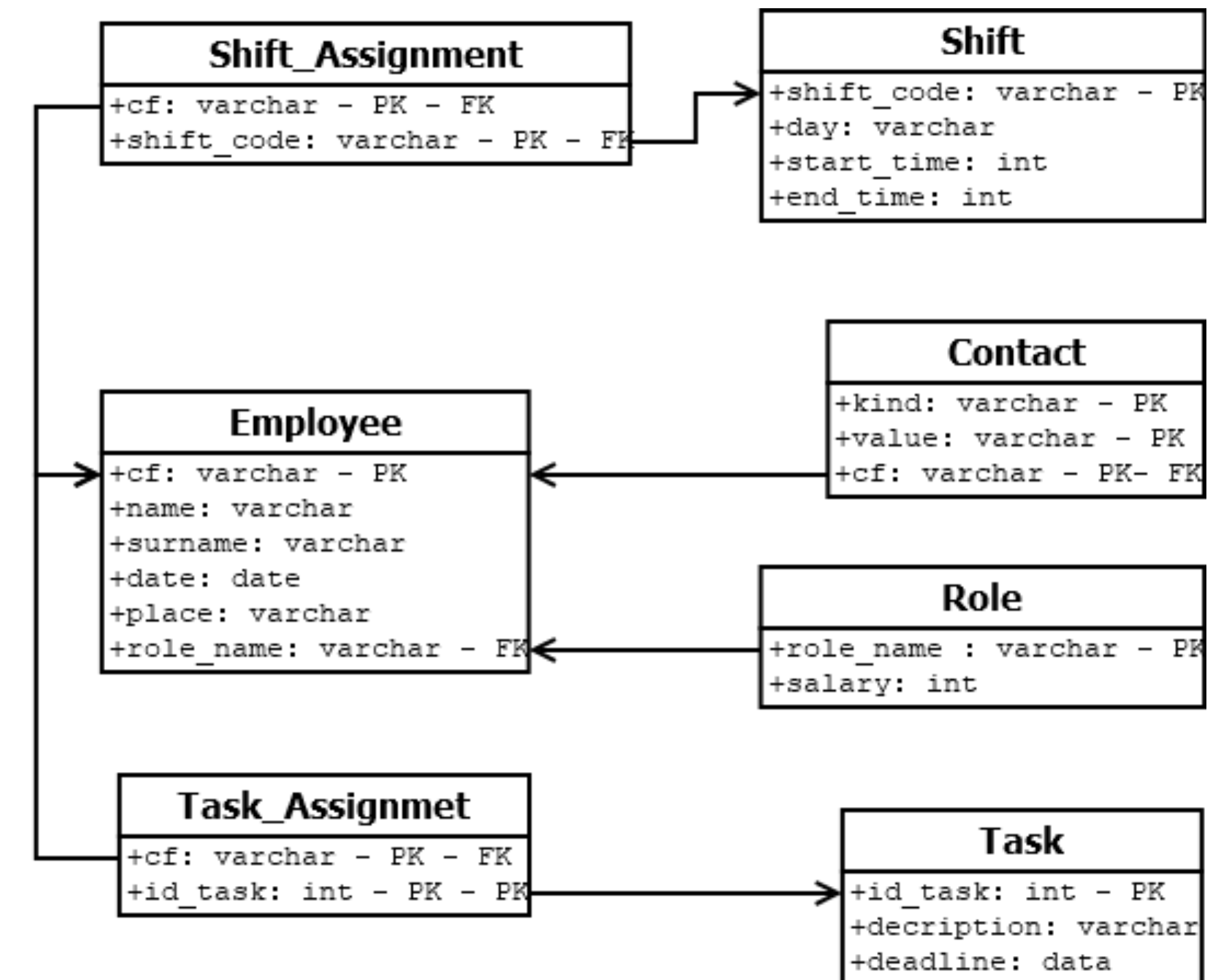
Una volta installato, startiamo MySQL, Apache Web Server ed avviamo l'applicazione.

Tramite MySQL possiamo sviluppare il nostro database sia scrivendo codice SQL che mediante una comoda ed intuitiva interfaccia grafica (phpMyAdmin).

Per poterci imbattere in più casi, realizziamo un database seguendo questo schema.

Andremo quindi ad inserire delle persone all'interno del nostro database con le relative informazioni, gli assegneremo dei turni di lavoro e dei task da svolgere.

(Se non vi va di implementare il database, potete scaricarlo da [qui](#) ed importarlo manualmente)



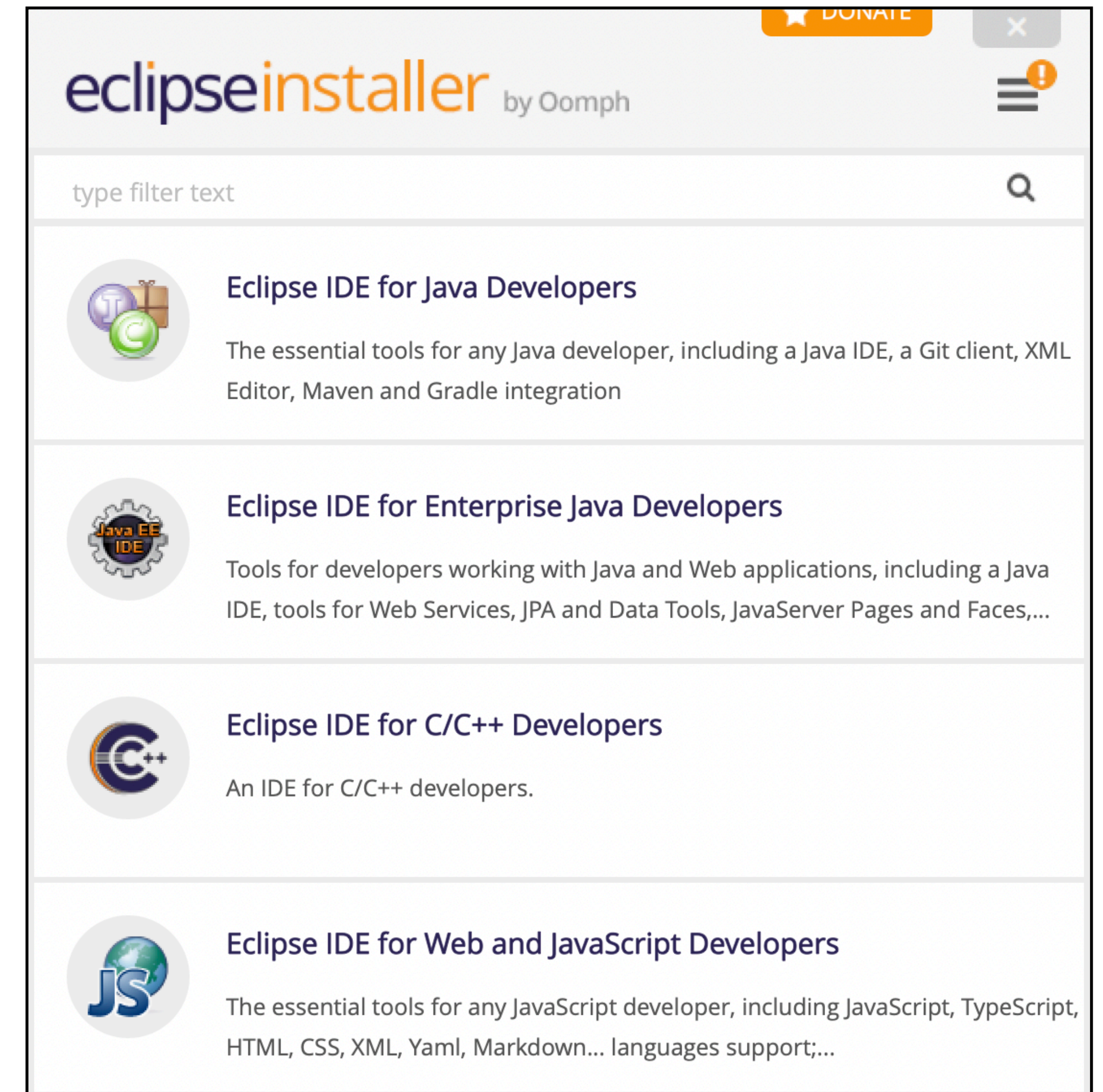
Step 2

Inizializzazione del progetto

Abbiamo bisogno di avere installato sul nostro sistema, una versione di java development kit (JDK) 8 o superiore e la scarichiamo da [qui](#) .

Andremo ad utilizzare Eclipse come nostro IDE disponibile [qui](#).

Andiamo ora ad installare la versione Enterprise che ci consente di realizzare dei dynamic web project.



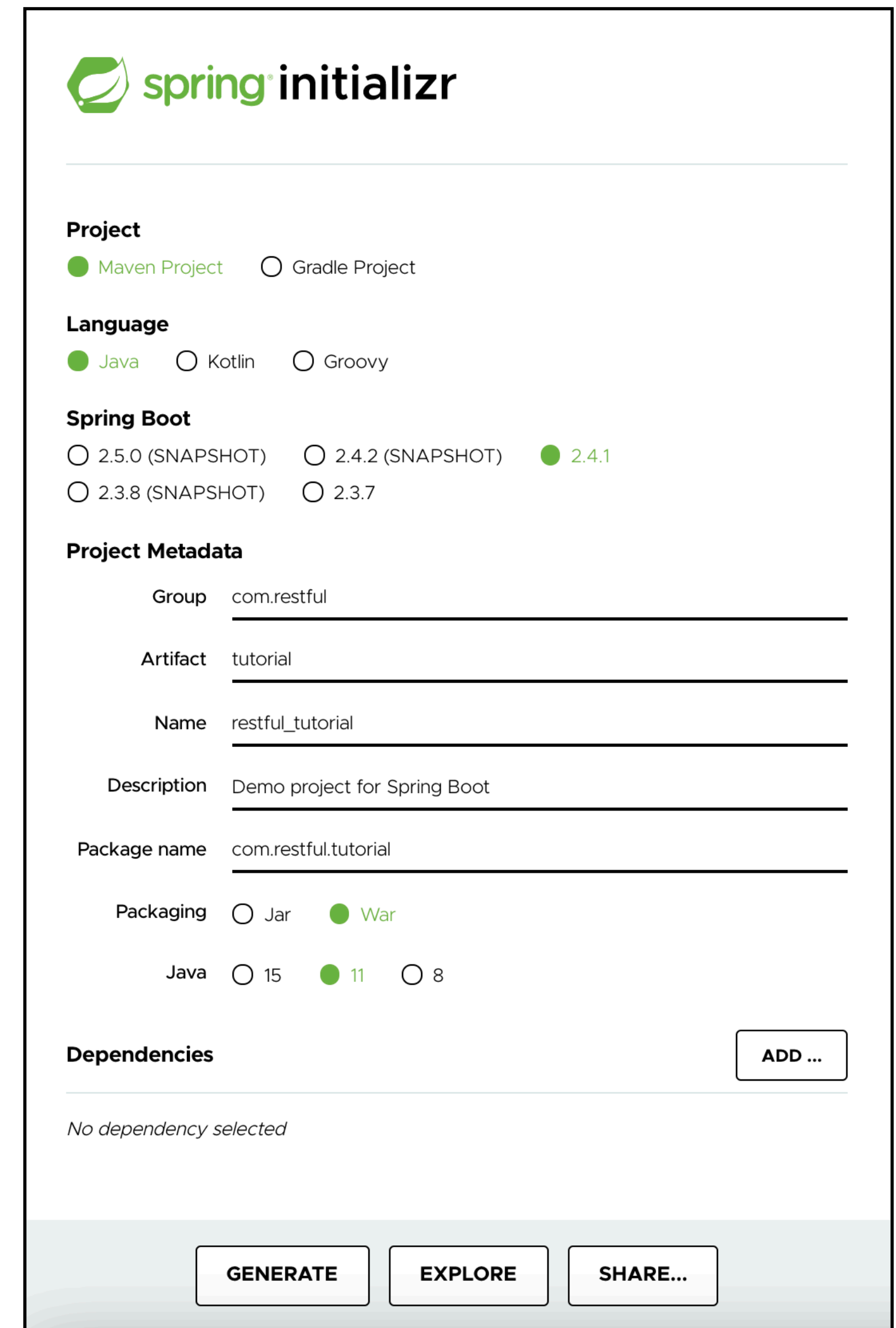
Tramite spring initializer andiamo a scaricare il progetto java preimpostato con spring boot

<https://start.spring.io/>

Scegliamo un progetto di tipo Maven, linguaggio Java e la versione di spring boot.

Successivamente inseriamo i metadati, scegliamo il package di tipo War (in caso vi serva in formato Jar, dopo vi mostreremo dove e come cambiarlo) e la versione di Java (scegliamo una versione compatibile con la nostra JDK).

Infine, generiamo il progetto e lo scarichiamo.



The screenshot shows the Spring Initializr web interface. At the top is the 'spring initializr' logo. Below it, the 'Project' section has 'Maven Project' selected. The 'Language' section has 'Java' selected. The 'Spring Boot' section has '2.4.1' selected. The 'Project Metadata' section includes fields for 'Group' (com.restful), 'Artifact' (tutorial), 'Name' (restful_tutorial), 'Description' (Demo project for Spring Boot), and 'Package name' (com.restful.tutorial). The 'Packaging' section has 'War' selected, and the 'Java' section has '11' selected. There is an 'ADD ...' button next to the 'Dependencies' section, which currently shows 'No dependency selected'. At the bottom, there are three buttons: 'GENERATE', 'EXPLORE', and 'SHARE...'.

Project
☒ Maven Project ☐ Gradle Project

Language
☒ Java ☐ Kotlin ☐ Groovy

Spring Boot
☐ 2.5.0 (SNAPSHOT) ☐ 2.4.2 (SNAPSHOT) ☒ 2.4.1
☐ 2.3.8 (SNAPSHOT) ☐ 2.3.7

Project Metadata

Group

Artifact

Name

Description

Package name

Packaging ☐ Jar ☒ War

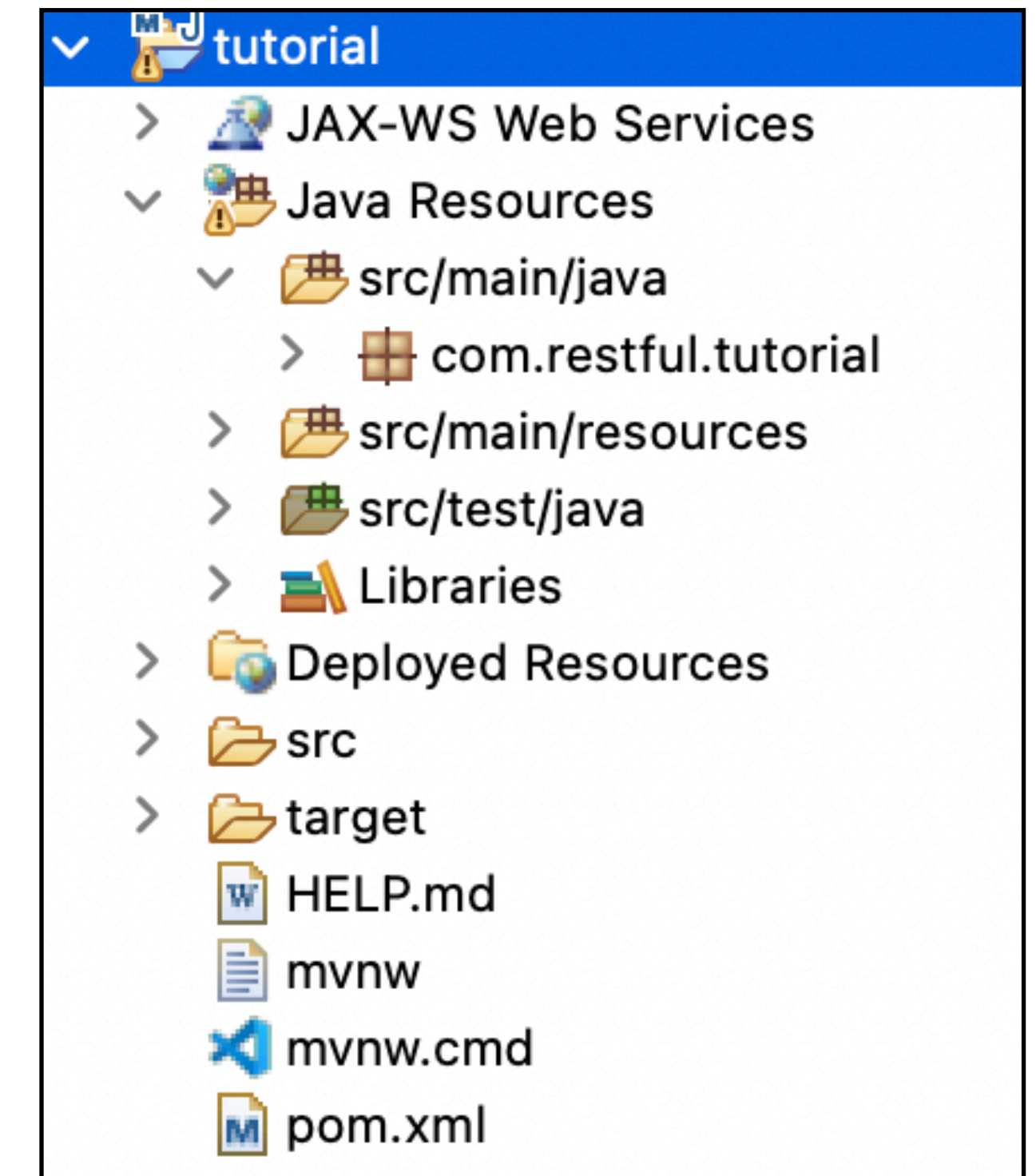
Java ☐ 15 ☒ 11 ☐ 8

Dependencies

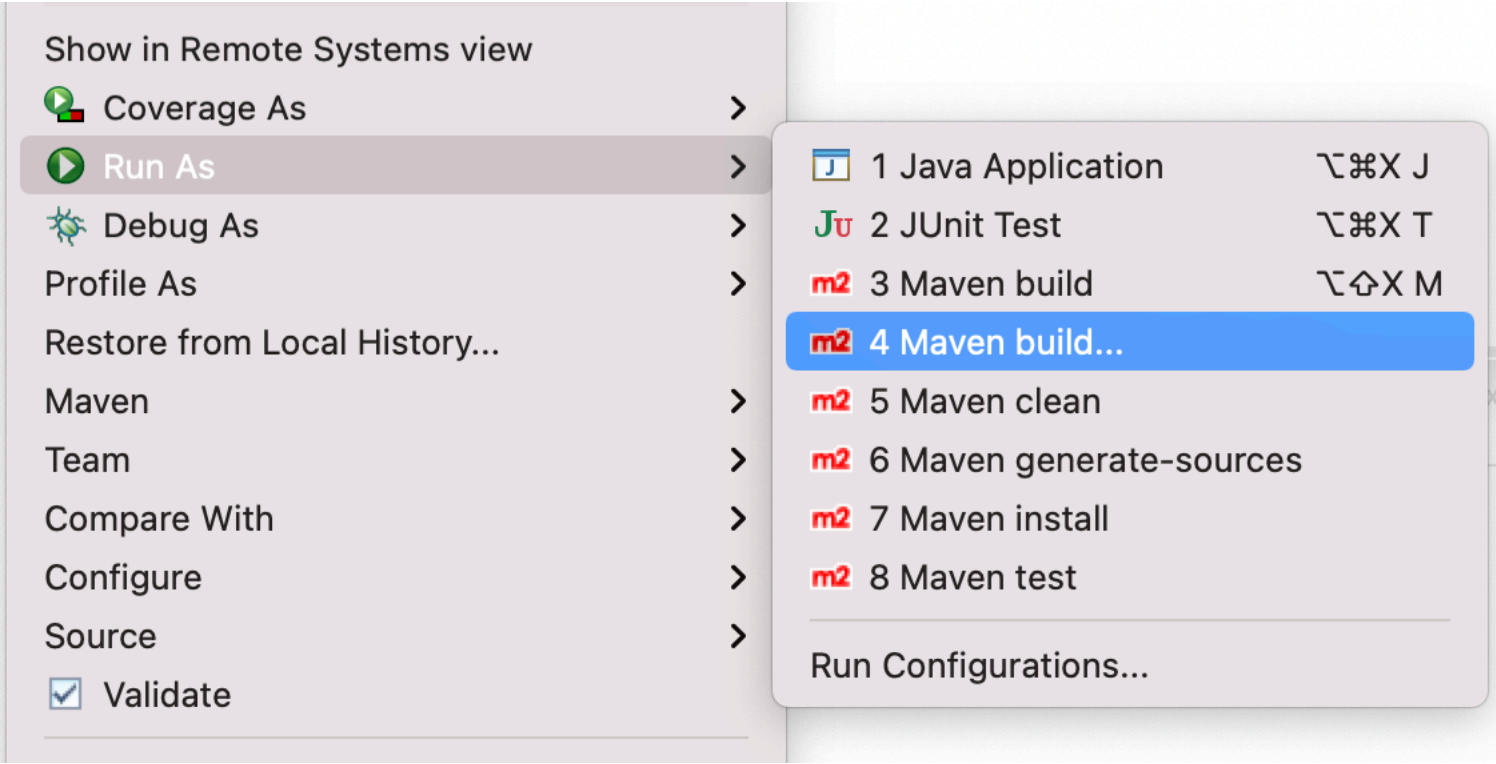
No dependency selected

Una volta estratto il progetto, andiamo ad importarlo su eclipse:
Clicchiamo quindi su file -> OpenProjects from File System...
Selezioniamo la Directory appena estratta e clicchiamo su Finish.

Importato il progetto, dovremmo avere una struttura simile a quella dell'immagine di fianco.

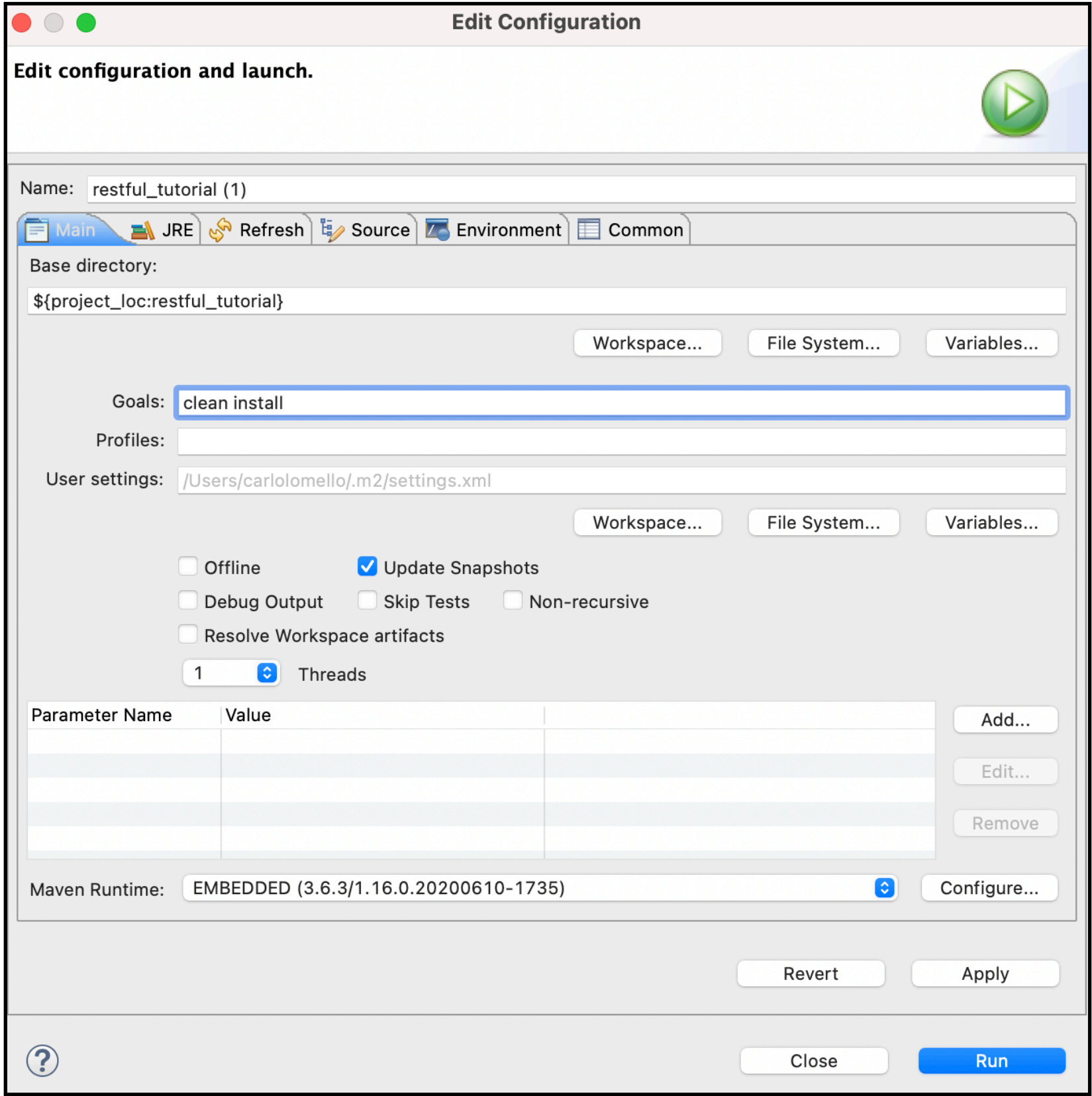


Facciamo una build su eclipse:
Clicchiamo col tasto destro del mouse sul nostro progetto in Project Explorer, selezioniamo Run As e clicchiamo su “Maven build...”

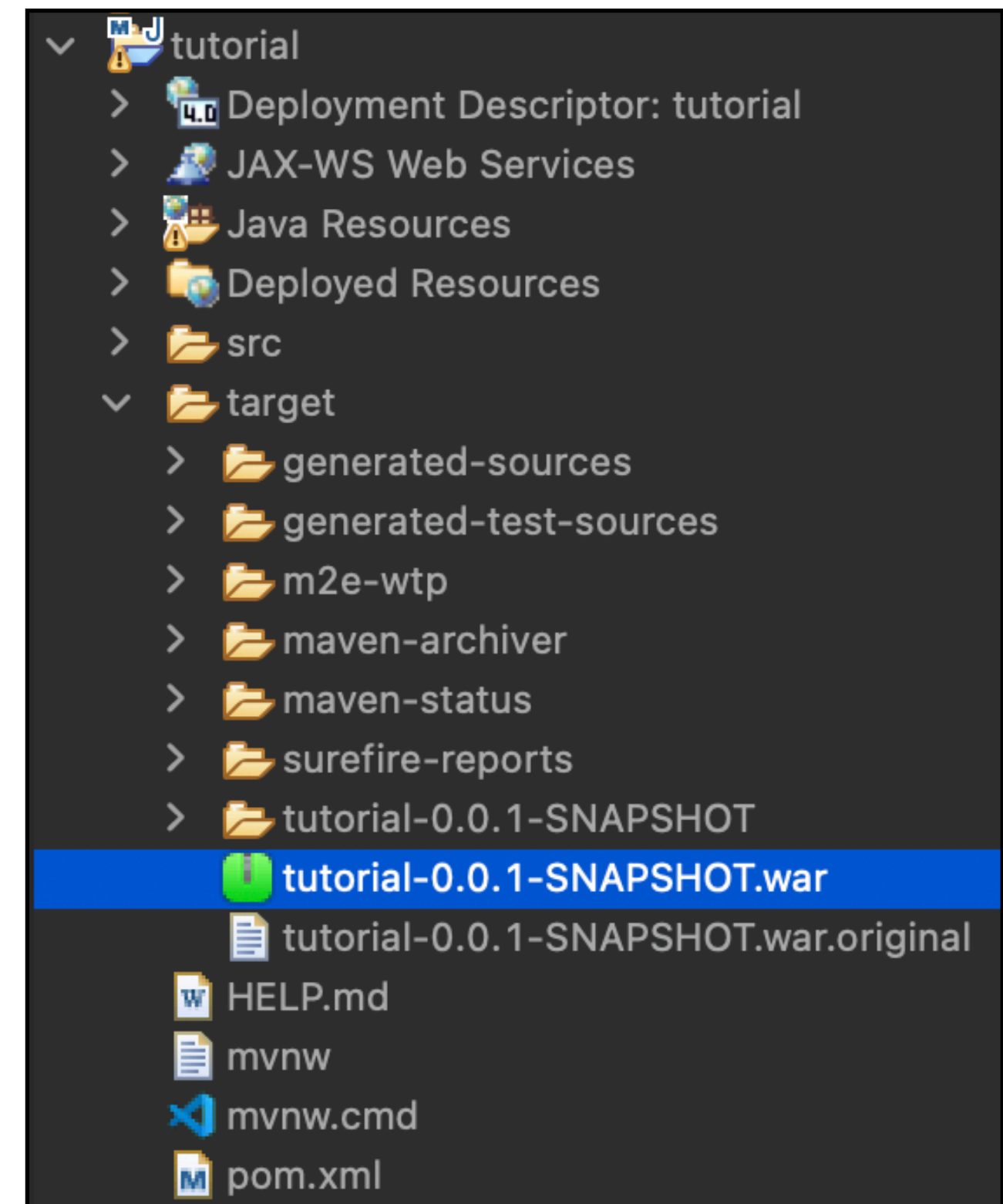


Ora, in Goals scriviamo “clean Install”, spuntiamo Update Snapshots ed infine clicchiamo su Run.

A questo punto il progetto verrà compilato e se non c’è nessun errore, in console comparirà la scritta BUILD SUCCESS.



Aggiornando la folder (F5), possiamo notare che, nella cartella “target” del progetto, è stata generata la nostra build in formato war che, a breve, andremo a deployare sul nostro server.



Step 3

Deploy su WildFly

WildFly è un application server multiplatforma creato da JBoss e sviluppato da Red Hat. È interamente realizzato in Java, che implementa le specifiche Java EE.

WildFly è un software gratuito e open source, soggetto ai requisiti della GNU Lesser General Public License (LGPL), versione 2.1.

Andiamo a scaricare l'ultima versione da <https://www.wildfly.org/downloads/> in formato zip e la estraiamo.

Andiamo, ora a creare un nuovo user per wildfly in modo da poter deployare la nostra build:

1. Apriamo la folder di wildfly prima estratta e rechiamoci all'interno della cartella "bin".
2. Nella directory bin sono disponibili diversi eseguibili "add-user" ma con estensioni diverse per i diversi sistemi operativi.
3. Quindi andiamo a creare un utente di tipo "Management User", inseriamo username e password seguendo le istruzioni e, se l'username è già esistente, andiamo a sovrascriverlo (ad esempio admin).

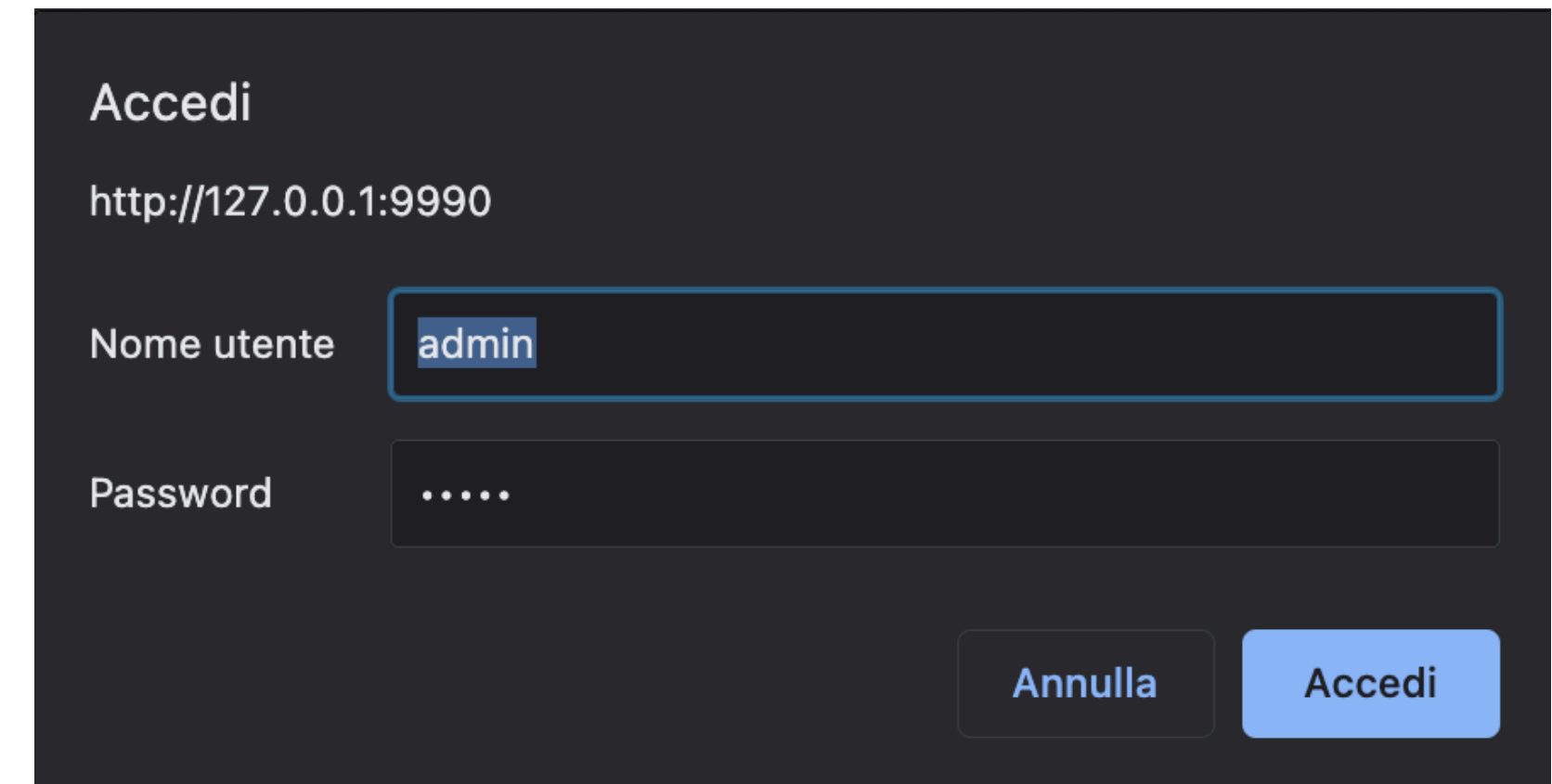
A questo punto, sempre nella cartella “bin” andiamo ad eseguire il file “standalone” per avviare il nostro server.

Se il server viene eseguito correttamente, alla fine ci apparirà il messaggio :

```
20:58:46,241 INFO [org.jboss.as] (Controller Boot Thread) WFLYSRV0060: Http  
management interface listening on http://127.0.0.1:9990/management  
20:58:46,241 INFO [org.jboss.as] (Controller Boot Thread) WFLYSRV0051: Admin  
console listening on http://127.0.0.1:9990
```

Quindi, tramite nostro browser preferito, navighiamo all'indirizzo <http://127.0.0.1:9990>

A questo punto, dopo aver inserito le credenziali d'accesso dell'utente appena creato, ci ritroveremo all'interno del pannello di controllo di Wildfly.

A screenshot of the Wildfly login interface. The background is dark. At the top, the word "Accedi" is displayed in a light color. Below it, the URL "http://127.0.0.1:9990" is shown. There are two input fields: "Nome utente" with the text "admin" and "Password" with five dots. At the bottom right, there are two buttons: "Annulla" and "Accedi".

Accedi

http://127.0.0.1:9990

Nome utente

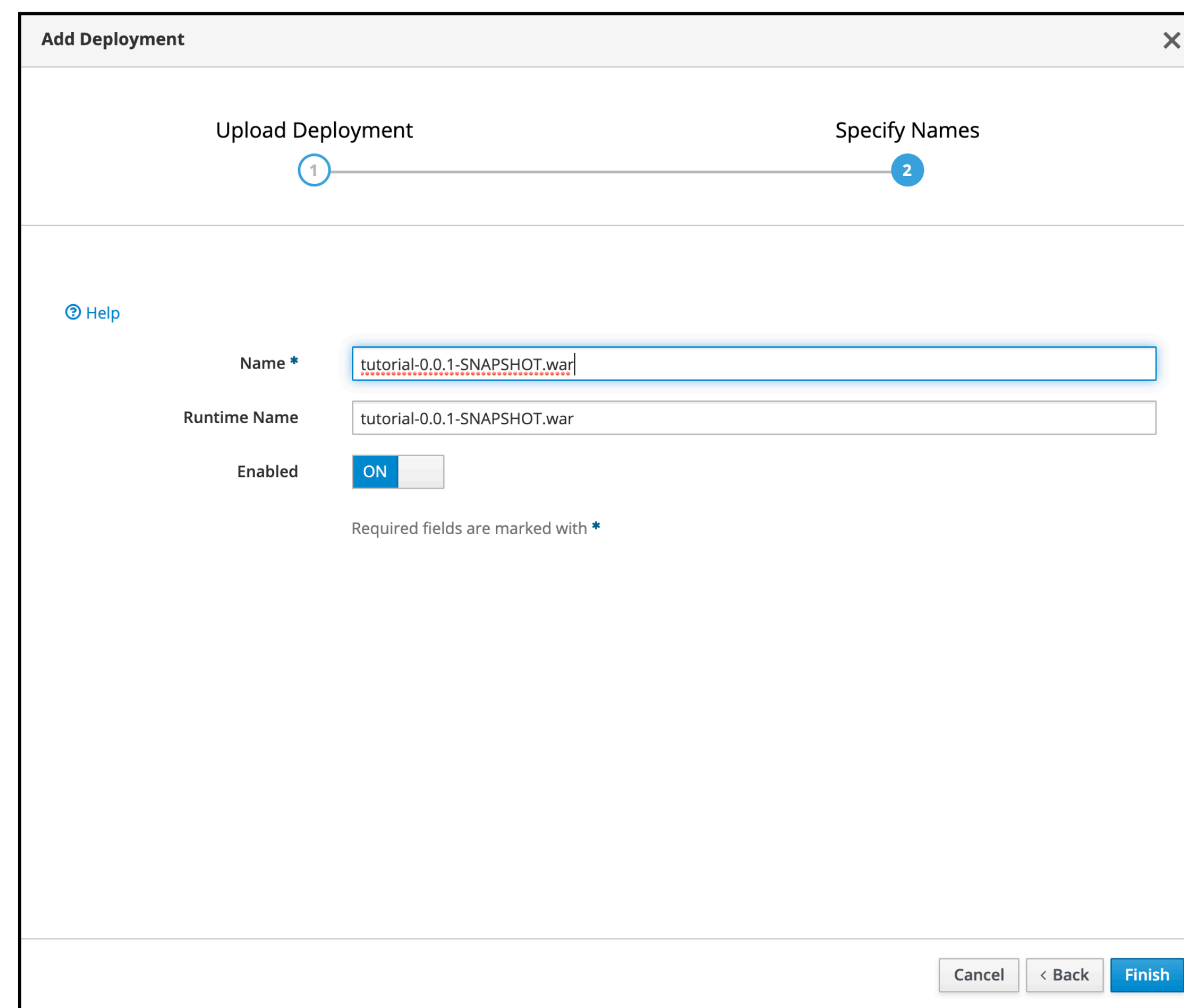
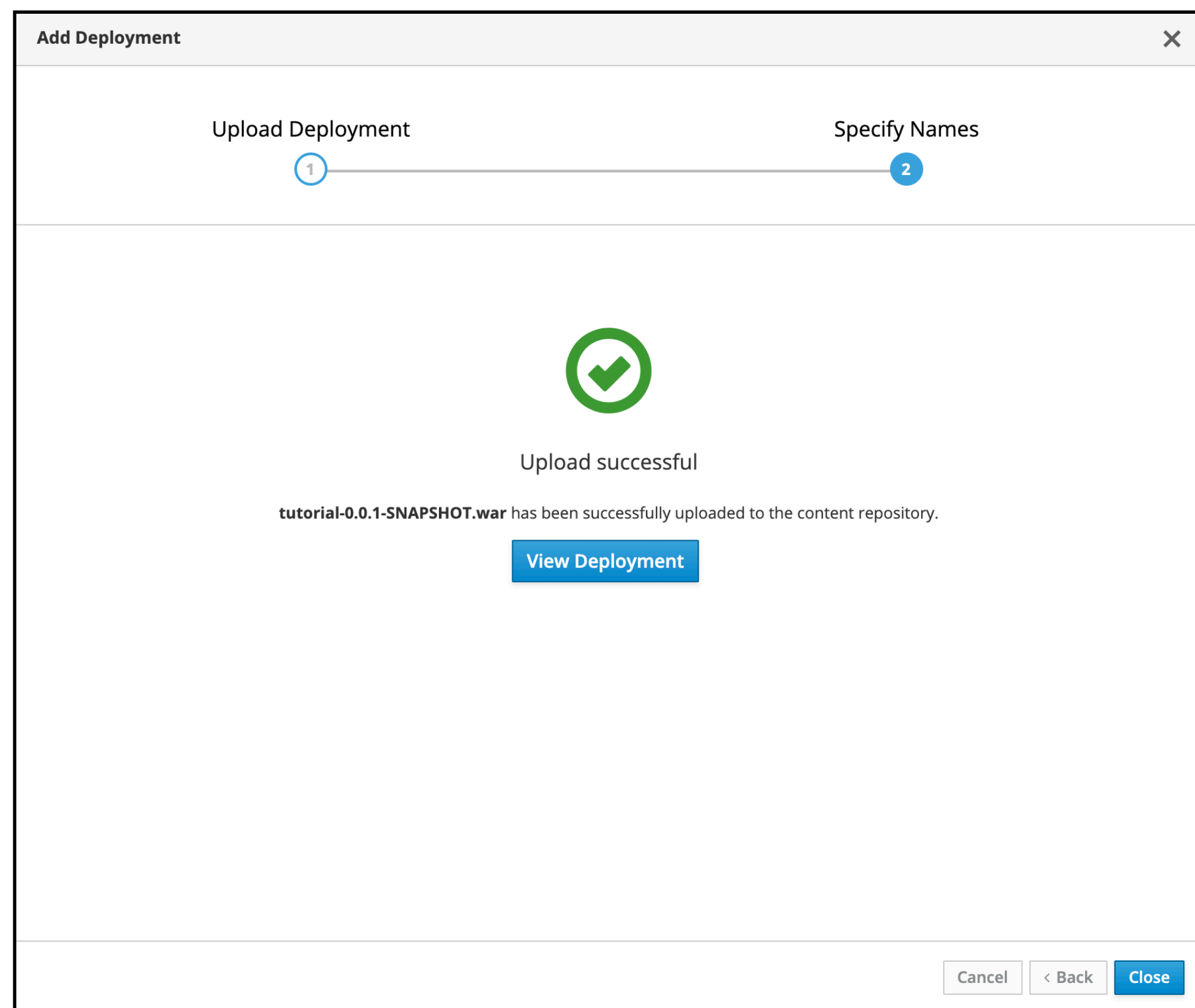
Password

Annulla Accedi

Navighiamo in Deployment, e clicchiamo su “add” in alto a sinistra, quindi “Upload Deployment” per poter aggiungere la build generata precedentemente su eclipse e situata all'interno della cartella target de progetto.

Quindi selezioniamo la build o la trasciniamo all'interno della schermata e procediamo col deploy.

Quindi, proseguiamo fino a scegliere il “Name” e “Runtime Name” per poi cliccare su Finish.



A questo punto, se non ci sono errori nella compilazione, il nostro file sarà caricato con successo ed rimarrà attivo sul server.

Step 4

Primi passi - connessione

In questo step, vedremo come importare le librerie in un maven project e vi guideremo alla configurazione dei parametri per stabilire la connessione al database.

```

<?xml version="1.0" encoding="UTF-8"?>
<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 https://maven.apache.org/xsd/
maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.4.1</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>
    <groupId>com.restful</groupId>
    <artifactId>tutorial</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <packaging>war</packaging>
    <name>restful_tutorial</name>
    <description>Demo project for Spring Boot</description>

    <properties>
        <java.version>11</java.version>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-tomcat</artifactId>
            <scope>provided</scope>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>

</project>

```

Apriamo il file di configurazione del progetto “pom.xml”, situato nella directory del progetto.

Come possiamo notare, questo foglio xml contiene informazioni del nostro progetto come: il nome stesso, il tipo di package (war) che andremo a generare con la build, la versione di java da utilizzare etc.

Le dependencies invece, sono le librerie che il nostro progetto maven va a scaricare ed installare automaticamente.

Di fatti, ci ritroviamo già pre-inserite quelle di Spring e Spring Boot.

Andiamo ad aggiungere queste altre dipendenze al nostro progetto:

```
<!-- MySQL -->
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>

<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <scope>runtime</scope>
</dependency>
```

- 1) Spring-boot-starter-data-jpa è la nostra libreria starter per l'utilizzo di Spring Data JPA con Hibernate.
- 2) mysql-connector-java invece ci consentirà di stabilire la connessione col nostro db e di poter effettuare le nostre operazioni di CRUD.

A questo punto, le dependencies nel nostro pom.xml dovrebbe essere così:

```
<dependencies>

  <!-- SpringBoot -->
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>

  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-tomcat</artifactId>
    <scope>provided</scope>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
  </dependency>

  <!-- MySQL -->
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
  </dependency>

  <dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <scope>runtime</scope>
  </dependency>

</dependencies>
```

Una volta effettuato il salvataggio delle modifiche apportate al pom, possiamo notare che in Java Resorces -> Libraries -> Maven Dependencies , sono state scaricate e quindi aggiunte le librerie spring-boot-starter-data-jpa e mysql-connector-java.

Infatti quando vorremmo aggiungere librerie esterne al nostro progetto ci basterà inserirle proprio in questo modo.

Consigliamo di cercare le librerie per i vostri progetti maven su <https://mvnrepository.com/>

Configuriamo ora la connessione al nostro database mysql.

Quindi navighiamo in resources ed apriamo il file application.properties ed aggiungiamo i seguenti parametri:

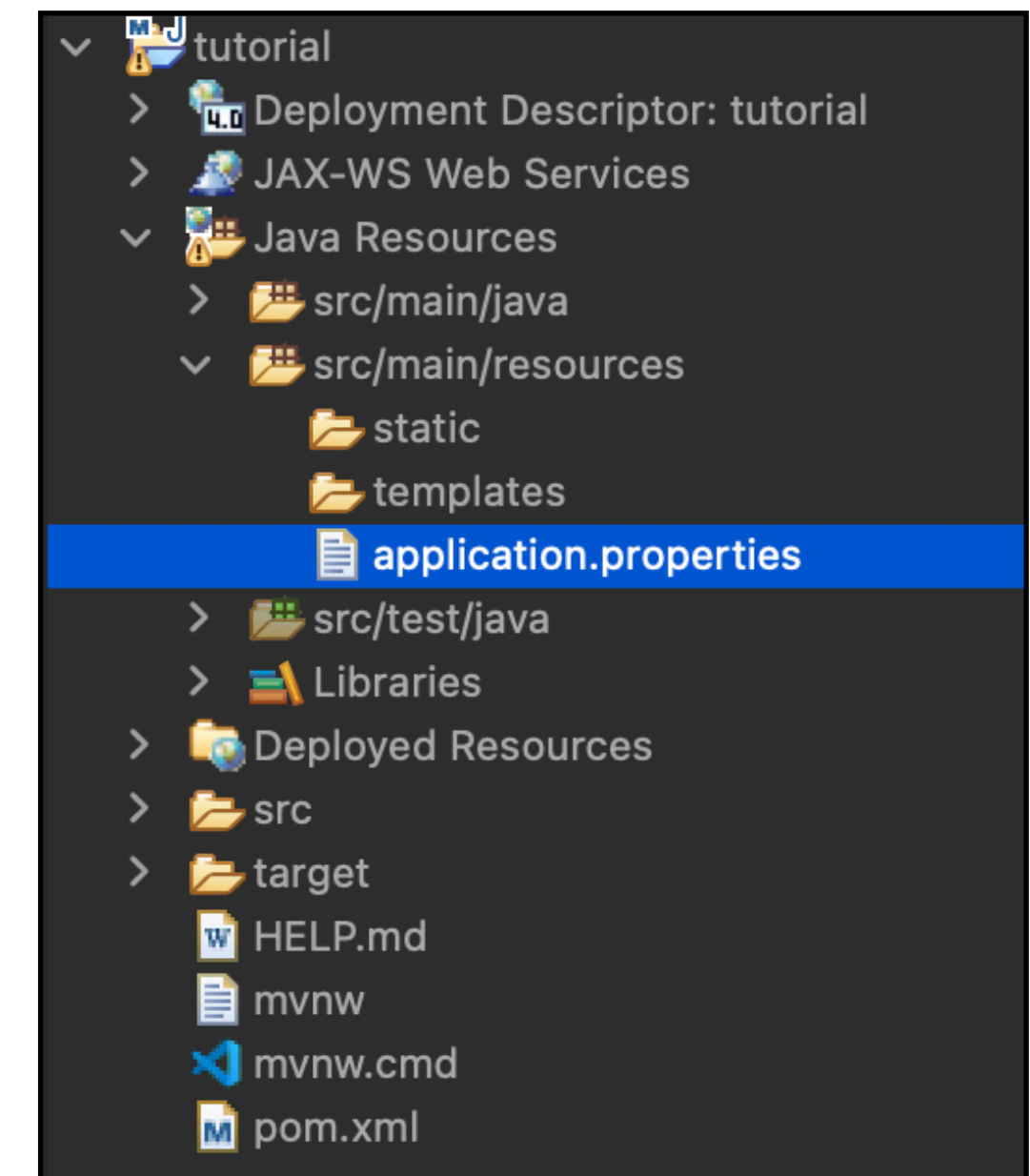
```
spring.jpa.hibernate.ddl-auto=none  
spring.datasource.url=jdbc:mysql://localhost:3306/restful_tutorial_db  
spring.datasource.username=root  
spring.datasource.password=
```

spring.jpa.hibernate.ddl-auto serve per poter effettuare in maniera automatica le operazioni di ddl in modo tale che le tabelle e le entities corrispondano.

Quindi, per evitare modifiche indesiderate al nostro database, la impostiamo a “none”.

Con spring.datasource.url vengono specificati
libreria:driver://indirizzo di connessione:numero della porta/nome del database .

Ovviamente spring.datasource.username e spring.datasource.password sono le credenziali d'accesso per il nostro database



Per poter verificare l'integrità del progetto, consigliamo di effettuare nuovamente una build ed un deploy su wildfly per poter controllare eventuali errori in console.

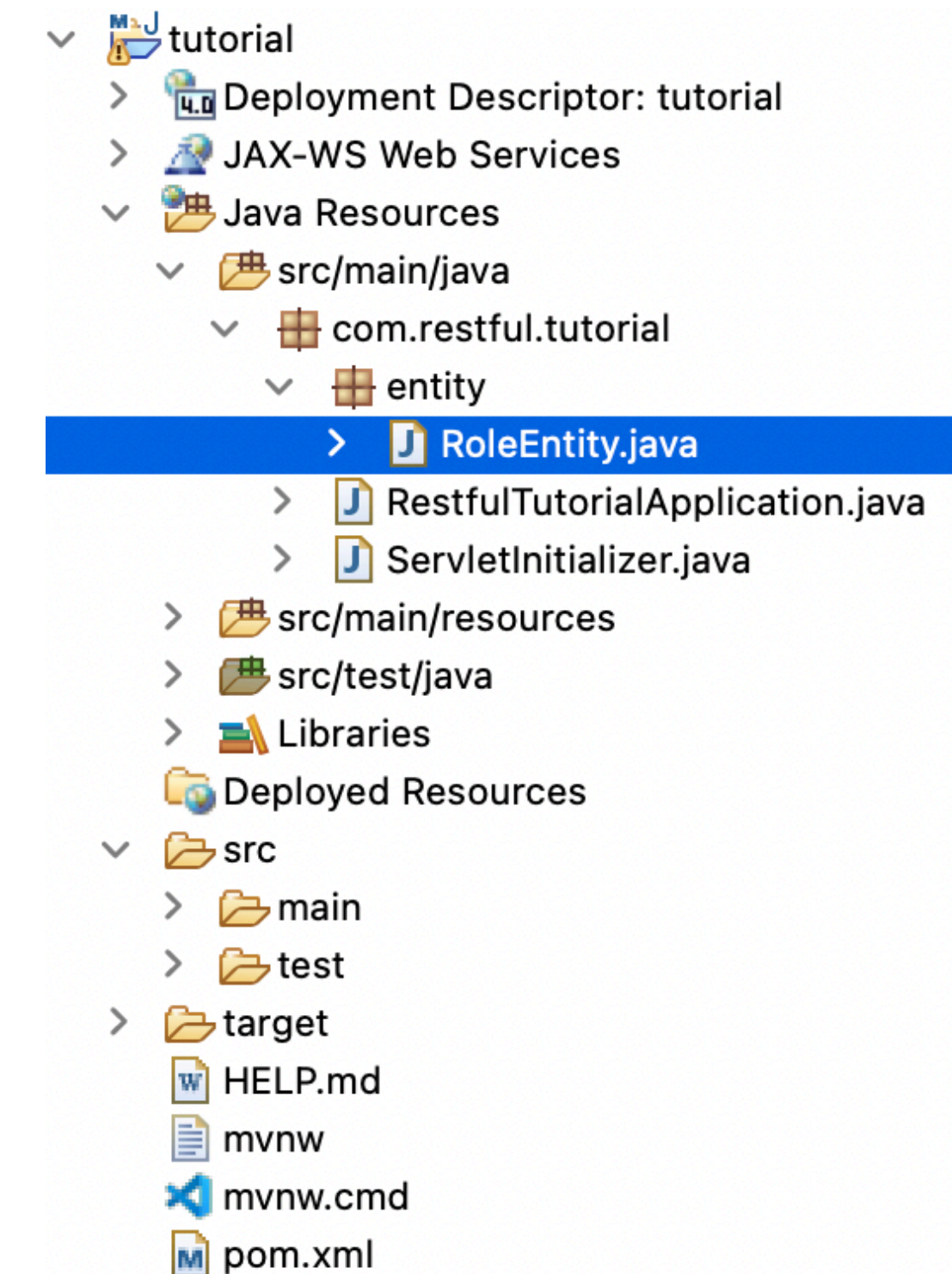
Step 5

Primi passi - prime operazioni di CRUD

In questa fase andremo a creare le classi necessarie per effettuare le operazioni di CRUD ed implementeremo i relativi servizi REST.

Iniziamo a creare un package “entity”, all’interno del package “com.restful.tutorial”, che conterrà tutte le entity corrispondenti alle tabelle del database.

Poi creiamo una Class RoleEntity che corrisponderà all’entità “role” del nostro database.



La nostra classe RoleEntity rispecchierà l'entità ad essa relativa:

```
package com.restful.tutorial.entity;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table(name = "role")
public class RoleEntity {

    @Id
    @Column(name = "role_name")
    private String roleName;

    @Column(name = "salary")
    private double salary;

    public String getRoleName() {
        return roleName;
    }

    public void setRoleName(String roleName) {
        this.roleName = roleName;
    }

    public double getSalary() {
        return salary;
    }

    public void setSalary(double salary) {
        this.salary = salary;
    }
}
```

Tramite l'annotazione **@Entity** marchiamo questa classe come entity.

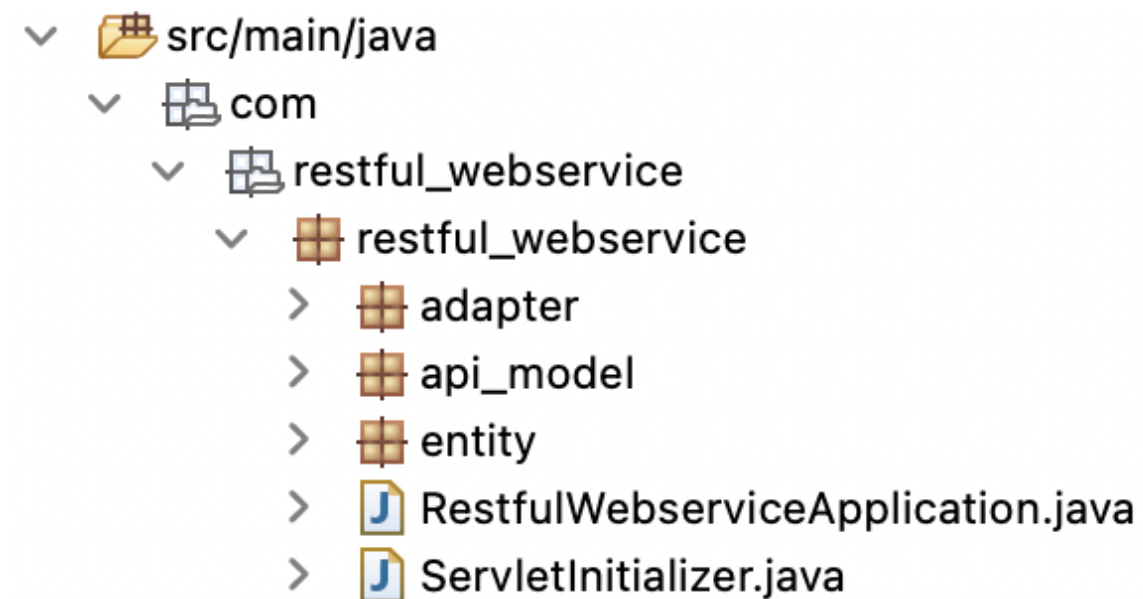
Tramite **@Table**, definiamo la corrispondenza della classe con l'entità role specificandone il nome.

Con l'annotation **@Id**, indichiamo la rispettiva chiave primaria della tabella

Con l'annotation **@Column**, andiamo a definire la corrispondenza con le colonne della tabella.

Per rispettare i principi dell'OOP, settiamo i modificatori d'accesso delle variabili “private” e definiamo i relativi metodi pubblici per poter accedere e settare le stesse.

Osserviamo che, i dati con i quali interagire, potrebbero essere differenti in base alle esigenze dei servizi FrontEnd e BackEnd, è buona prassi creare dei DTO (Data Transfer Object) che rappresentano le richieste in input ai servizi REST, così da essere separati dagli oggetti di business.



Andiamo quindi a creare i seguenti package allo stesso livello di entity :

- api_model
- adapter

Nel package api_model, creeremo le classi che utilizzeremo per rappresentare gli oggetti delle nostre API (ovvero i DTO), mentre nel package adapter, creeremo le classi che conterranno i metodi che ci permetteranno di effettuare la conversione dai modelli API alle Entity (gli oggetti di business) e viceversa.

Creiamo nel package `api_model`, la classe `RoleAPIModel` che rappresenta il DTO relativo alla classe `RoleEntity`.

```
public class RoleAPIModel {  
  
    private String roleName;  
    private double salary;  
  
    public String getRoleName() {  
        return roleName;  
    }  
  
    public void setRoleName(String roleName) {  
        this.roleName = roleName;  
    }  
  
    public double getSalary() {  
        return salary;  
    }  
  
    public void setSalary(double salary) {  
        this.salary = salary;  
    }  
}
```

Questa conterrà quindi i campi che dovranno essere riportati come richiesta o risposta ai servizi che andremo poi a creare relativi alla classe `role`.

Adesso creiamo anche la classe RoleAdapter che conterrà i metodi :

```
public class RoleAdapter {  
  
    public static RoleAPIModel to_API_model(RoleEntity roleEntity) {  
        RoleAPIModel roleAPI = new RoleAPIModel();  
        roleAPI.setRoleName(roleEntity.getRoleName());  
        roleAPI.setSalary(roleEntity.getSalary());  
        return roleAPI;  
    }  
  
    public static RoleEntity to_entity_model(RoleAPIModel roleAPI) {  
        RoleEntity roleEntity = new RoleEntity();  
        roleEntity.setRoleName(roleAPI.getRoleName());  
        roleEntity.setSalary(roleAPI.getSalary());  
        return roleEntity;  
    }  
  
    public static List<RoleAPIModel> to_API_model_list(List<RoleEntity> listRoleEntity){  
        List<RoleAPIModel> listRoleAPI = new ArrayList<RoleAPIModel>();  
        for (RoleEntity role : listRoleEntity) {  
            listRoleAPI.add(to_API_model(role));  
        }  
        return listRoleAPI;  
    }  
  
    public static List<RoleEntity> to_entity_model_list(List<RoleAPIModel> listRoleAPIModel){  
        List<RoleEntity> listRoleEntity = new ArrayList<RoleEntity>();  
        for (RoleAPIModel role : listRoleAPIModel) {  
            listRoleEntity.add(to_entity_model(role));  
        }  
        return listRoleEntity;  
    }  
}
```

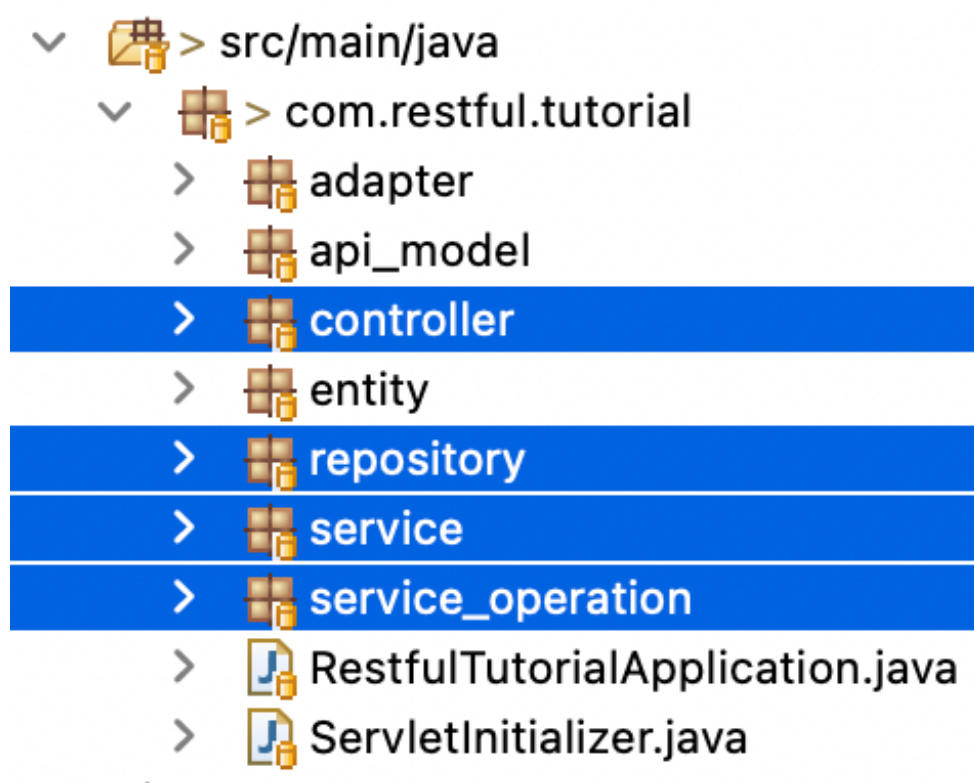
- to_API_model: permette convertire l'entity dell'role presa in input facendo così ritornare il DTO ad esso relativo

- to_entity_model: permette convertire il DTO dell'role preso in input facendo così ritornare l'entity ad esso relativo

- to_API_model_list: converte una lista di entity dell'role in una lista di DTO ad essa relativa.

- to_entity_model_list: converte una lista di DTO dell'role in una lista di entity ad essa relativa.

Adesso andiamo a creare tutti gli elementi necessari per poter implementare i servizi e le logiche di business.



A questo punto, creiamo degli ulteriori nuovi package:

- controller: conterrà le classi che espongono i servizi rest.
- repository: conterrà le interfacce di repository
- service: conterrà le classi ed i relativi metodi che si occuperanno di effettuare le varie operazioni di CRUD sul database
- service_operation: conterrà le classi che implementano le logiche di business

Procediamo col creare l'interfaccia RoleRepository.

```
@Repository  
public interface RoleRepository extends JpaRepository<RoleEntity, String> {  
}
```

@Repository è un'annotazione Spring che indica che la classe decorata è un repository.

Una repository è un meccanismo per incapsulare il comportamento di archiviazione, recupero e ricerca che emula una raccolta di oggetti.

Tutte le repository estendono l'interfaccia JpaRepository<T, ID> dove :

- T rappresenta la classe Entity
- ID rappresenta il tipo della relativa chiave primaria

Continuiamo col creare una classe RoleService in service:

```
@Service
@Transactional
public class RoleService {
    @Autowired
    private RoleRepository roleRepository;

    public List<RoleEntity> findAllR() {
        return roleRepository.findAll();
    }

    public RoleEntity saveR(RoleEntity roleEntity) {
        return roleRepository.save(roleEntity);
    }

    public RoleEntity findRole(String roleName) {
        return roleRepository.findById(roleName).get();
    }

    public void deleteRole(String roleEntity) {
        roleRepository.deleteById(roleEntity);
    }
}
```

Da come possiamo osservare, vanno inseriti sulla classe le annotazioni:

- **@Service**: viene utilizzata con classi che forniscono alcune funzionalità dove sono implementate le logiche di business e i metodi per interagire col database. Inoltre, Spring rileverà automaticamente queste classi quando viene utilizzata la configurazione basata su annotazioni e la scansione del percorso di classe

- **@Transactional**: è un'annotazione che specifica la semantica delle transazioni su un metodo. In questo caso viene utilizzato un approccio dichiarativo di default, il quale eseguire il rollback sulle eccezioni di runtime (per esempio SQLException)

Sempre in RoleService in service:

```
@Service
@Transactional
public class RoleService {
    @Autowired
    private RoleRepository roleRepository;

    public List<RoleEntity> findAllR() {
        return roleRepository.findAll();
    }

    public RoleEntity saveR(RoleEntity roleEntity) {
        return roleRepository.save(roleEntity);
    }

    public RoleEntity findRole(String roleName) {
        return roleRepository.findById(roleName).get();
    }

    public void deleteRole(String roleEntity) {
        roleRepository.deleteById(roleEntity);
    }
}
```

- **@Autowired**: fornisce un controllo più dettagliato su dove e come eseguire l'autowriting. È possibile utilizzare l'annotazione sulle proprietà per eliminare i metodi di impostazione così che, quando passerai i valori delle proprietà autowired usando <proprietà>, Spring assegnerà automaticamente quelle proprietà con i valori o i riferimenti passati.

Una volta istanziata la repository relativa all'entità interessata (in questo caso roleEntity), sarà possibile creare dei metodi wrapper ad hoc utilizzando i metodi già implementati nella repository e da Spring per interagire col database.

Nel package `service_operation` andiamo a definire una classe `RoleServiceOperation` che implementa le operazioni di business.

```
@Service
public class RoleServiceOperation {
    @Autowired
    private RoleService service;
    public List<RoleAPIModel> getRoleList() {
        return RoleAdapter.to_API_model_list(service.findAllR());
    }
    public RoleAPIModel getRoleByRoleName(String roleName) {
        RoleEntity roleEntity = service.findRole(roleName);
        if(roleEntity!=null)
            return RoleAdapter.to_API_model(roleEntity);
        else
            throw new NoSuchElementException(" ERROR: role "
                +roleEntity.getRoleName()+" not foud");
    }
    public void deleteRoleById(String roleName){
        service.deleteRole(roleName);
    }
    public RequestSaveRoleAPIModel saveRole(RequestSaveRoleAPIModel role) {
        return RequestSaveRoleAdapter.to_API_model(service.saveR(
            RequestSaveRoleAdapter.to_entity_model(role)));
    }
    public RequestSaveRoleAPIModel updateRole(RequestSaveRoleAPIModel role) {
        RoleEntity roleFound = service.findRole(role.getRoleName());
        if(roleFound!=null) {
            RoleEntity roleEntity = RequestSaveRoleAdapter.to_entity_model(role);
            return RequestSaveRoleAdapter.to_API_model(service.saveR(roleEntity));
        }else {
            throw new NoSuchElementException(" ERROR: role "
                +role.getRoleName()+" not foud");
        }
    }
}
```

Questa classe ci consente di avere uno strato intermedio per effettuare le logiche di business, in modo da separare e rielaborare i dati ottenendo così una migliore leggibilità del codice.

Possiamo osservare che questa classe mantiene le stesse peculiarità di `RoleService`, così da poter istanziare quest'ultima al suo interno e quindi sfruttarne i metodi per ottenere i dati a noi interessati, per poi rielaborarli tramite gli adapter precedentemente creati.

All'interno del package controller definiamo la classe RoleController che ha lo scopo di esporre i servizi REST.

```
@RestController
public class RoleController {

    @Autowired
    private RoleServiceOperation service;

    @GetMapping("/role_list")
    public List<RoleAPIModel> list() {
        return service.getRoleList();
    }
    @DeleteMapping("/delete_role")
    public void deleteById(@RequestBody String roleName) {
        service.deleteRoleById(roleName);
    }
    @GetMapping("/role_by_roleName")
    public RoleAPIModel getRoleByRoleName (String roleName) {
        return service.getRoleByRoleName(roleName);
    }
    @PostMapping("/insert_role")
    public RequestSaveRoleAPIModel saveR(@RequestBody RequestSaveRoleAPIModel roleAPI){
        return service.saveRole(roleAPI);
    }
    @PutMapping("/update_role")
    public RequestSaveRoleAPIModel updateR(@RequestBody RequestSaveRoleAPIModel roleAPI){
        return service.updateRole(roleAPI);
    }
}
```

La prima cosa che dobbiamo fare è dire a Spring che si tratta di un controller REST, quindi è necessario aggiungere un'annotazione `@RestController`. Questa annotazione fa sì che questa classe venga rilevata dalla scansione dei componenti che quindi la renderà disponibile al contesto dell'applicazione.


Per ogni tipo di operazione REST (quali get, post, put, delete, etc), in Spring ci sarà un'annotazione corrispondente.

L'annotazione `@RequestBody` mappa il corpo `HttpRequest` su un oggetto di trasferimento o dominio, consentendo la deserializzazione automatica del corpo in entrata su un oggetto Java.

È giunto finalmente il momento di testare le nostre API!
Procediamo quindi con lo scaricare Postman da [qui](#) e lo installiamo.

Per prima cosa, facciamo una build della nostra applicazione ed effettuiamone un nuovo deploy sul server wildfly come precedentemente spiegato.

tutorial-0.0.1-SNAPSHOT.war

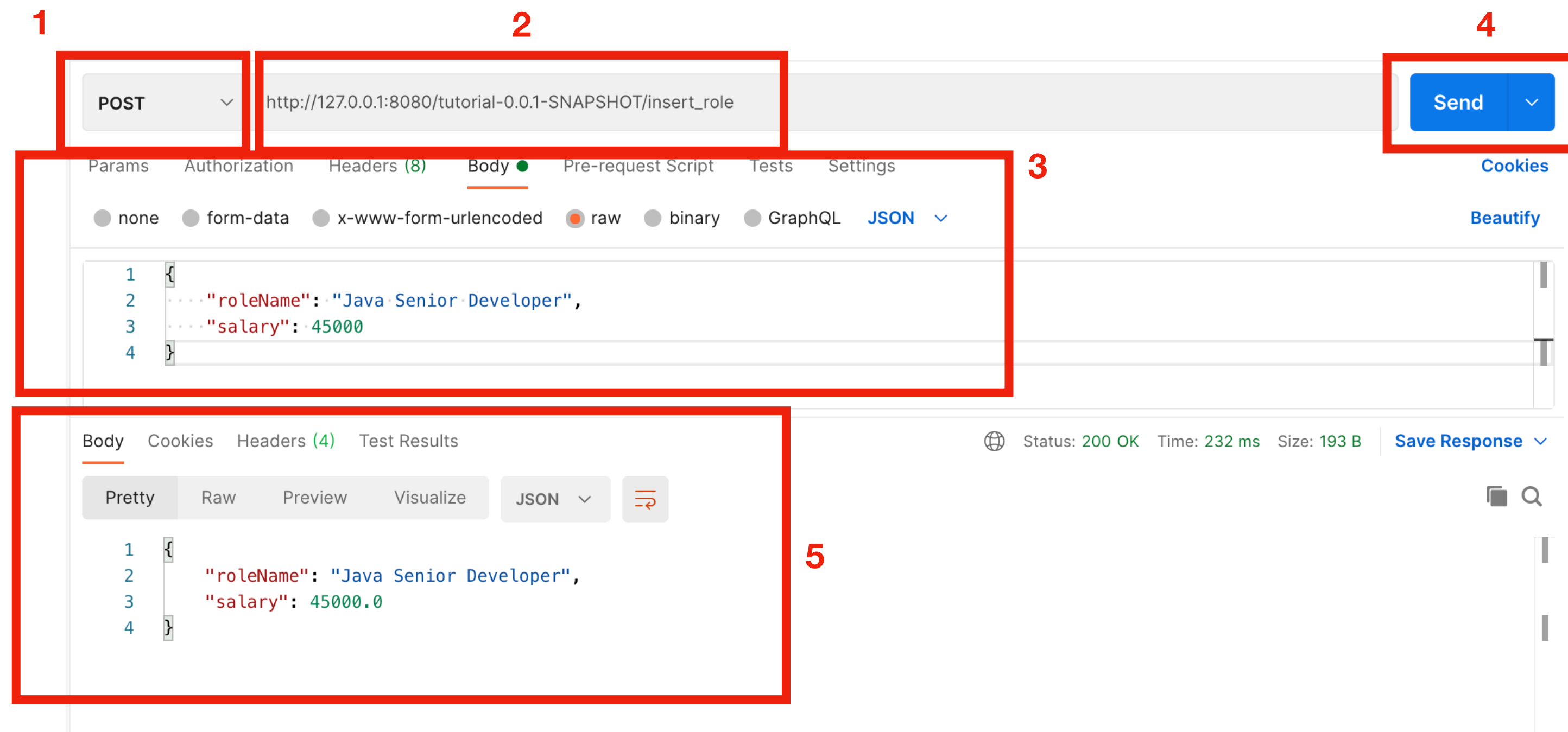
 The deployment **tutorial-0.0.1-SNAPSHOT.war** is enabled and active. [Disable](#)

Main Attributes

Name:	tutorial-0.0.1-SNAPSHOT.war
Runtime Name:	tutorial-0.0.1-SNAPSHOT.war
Context Root:	/tutorial-0.0.1-SNAPSHOT
Enabled, Managed, Exploded:	✓ ✓ ✕
Status:	OK
Last enabled at:	2/5/22, 12:19 PM
Last disabled at:	n/a

Una volta terminato con esito positivo il deploy, potete cliccare sulla voce Context Root che aprirà una nuova scheda nel browser contenente appunto il path per accedere ai nostri servizi
(es: <http://127.0.0.1:8080/tutorial-0.0.1-SNAPSHOT/>)

Utilizziamo Postman per popolare il nostro database tramite operazioni di POST:



- 1: selezioniamo il tipo del servizio
- 2: inseriamo l'url precedentemente ricavato e gli concateniamo il path del servizio che vogliamo utilizzare (es: context_root + /insert_role)
- 3: selezioniamo body, raw e JSON ed andiamo a compilare il nostro JSON con i dati che vogliamo inserire
- 4: inviamo la richiesta

- 5: visualizziamo la response del servizio utilizzato sul nostro server

Ora effettuiamo un'altra operazione di tipo GET:

The screenshot shows a REST client interface with the following components highlighted by red boxes and numbered 1 through 5:

- 1:** The HTTP method dropdown menu set to "GET".
- 2:** The URL input field containing "http://127.0.0.1:8080/tutorial-0.0.1-SNAPSHOT/role_by_roleName".
- 3:** The "Body" tab selected in the request configuration panel, with "form-data" chosen as the content type. Below it, a table lists parameters:

	KEY	VALUE	DESCRIPTION
<input checked="" type="checkbox"/>	roleName	Java Senior Developer	
	Key	Value	Description

- 4:** The "Send" button to execute the request.
- 5:** The "Body" tab in the response section, showing the JSON response in "Pretty" format:

```
1 {
2   "roleName": "Java Senior Developer",
3   "salary": 45000.0,
4   "employeeList": []
5 }
```

At the bottom of the response section, the status is "200 OK", time is "20 ms", and size is "211 B".

- 1: selezioniamo il tipo del servizio
- 2: inseriamo l'url precedentemente ricavato e gli concateniamo il path del servizio che vogliamo utilizzare (es: context_root + / role_by_roleName)
- 3: selezioniamo body, form-data ed andiamo ad inserire i parametri richiesti
- 4: inviamo la richiesta

- 5: visualizziamo la response del servizio utilizzato sul nostro server

Step 6

Completiamo le operazioni di CRUD

In questa fase andremo ad illustrare e spiegare come implementare le varie relazioni tra le entità e gestirne le relative operazioni.

Ispirandoci alla tabella dell'Employee presente sul nostro database, andremo ora a creare una classe EmployeeEntity, nel relativo package (entity), con tutti gli attributi e le relative relazioni.

```
@Entity
@Table(name = "employee")
public class EmployeeEntity {
```

```
    @Id
    @Column(name = "cf")
    private String cf;
    @Column(name = "name")
    private String name;
    @Column(name = "surname")
    private String surname;
    @Column(name = "date")
    private Date date;
    @Column(name = "place")
    private String place;
```

Ora vogliamo focalizzare la nostra attenzione sulle relazioni che si creano tra le varie entità:

La prima che andremo ad analizzare è la relazione *molti a molti*:

```
    @ManyToMany(fetch = FetchType.LAZY, cascade = CascadeType.PERSIST)
    @JoinTable(
        name = "shift_assignment",
        joinColumns = {@JoinColumn(name = "cf")},
        inverseJoinColumns = {@JoinColumn(name = "shift_code")}
    )
    private List<ShiftEntity> shiftList = new ArrayList<ShiftEntity>();

    @ManyToMany(fetch = FetchType.LAZY, cascade = CascadeType.PERSIST)
    @JoinTable(
        name = "task_assignment",
        joinColumns = {@JoinColumn(name = "cf")},
        inverseJoinColumns = {@JoinColumn(name = "id_task")}
    )
    private List<TaskEntity> taskList = new ArrayList<TaskEntity>();
```

Usiamo l'annotazione @ManyToMany per creare una relazione molti-a-molti tra due entità. In un'associazione bidirezionale, l'annotazione @ManyToMany viene utilizzata su entrambe le entità, ma solo un'entità può essere proprietaria della relazione.

Consideriamo ad esempio la relazione con shift:

```
@ManyToMany(fetch = FetchType.LAZY, cascade = CascadeType.PERSIST)
@JoinTable(
    name = "shift_assignment",
    joinColumns = {@JoinColumn(name = "cf")},
    inverseJoinColumns = {@JoinColumn(name = "shift_code")}
)
private List<ShiftEntity> shiftList = new ArrayList<ShiftEntity>();
```

@ManyToMany(fetch = FetchType.LAZY, cascade = CascadeType.PERSIST)

- Indichiamo una relazione molti a molti con il FetchType.LAZY nel caso in cui si voglia stabilire che i dati relativi all'entità vengano “joinati” automaticamente.

Altrimenti, specifichiamo una relazione con il FetchType.EAGER per far sì che i dati della relazione vengano ricavati solo ed esclusivamente quando viene invocato il metodo GET.

- Con CascadeType andiamo a specificare che, quando eseguiamo un'azione sull'entità, la stessa azione verrà applicata all'entità associata.

In questo caso, abbiamo scelto di utilizzare PERSIST così che, quando vengono effettuate le operazioni di insert o update, queste vengono automaticamente riflesse sull'entità associata.


```
@ManyToMany(fetch = FetchType.LAZY, cascade = CascadeType.PERSIST)
@JoinTable(
    name = "shift_assignment",
    joinColumns = {@JoinColumn(name = "cf")},
    inverseJoinColumns = {@JoinColumn(name = "shift_code")}
)
private List<ShiftEntity> shiftList = new ArrayList<ShiftEntity>();
```

@JoinTable può essere utilizzata per mappare le associazioni sia bidirezionali sia unidirezionali tra le tabelle del database dove:

- name specifica il nome della tabella many to many
- joinColumns indica la chiave che possiede l'associazione alla many to many
- InverseJoinColumn indica la chiave che collega l'entità Shift alla tabella many to many ma che non è presente nell'entità corrente

L'annotazione @JoinColumn definisce la colonna che useremo per unire un'associazione di entità o una raccolta di elementi passandogli il nome della chiave della tabella destinataria.

AGGIUNGERE MANY TO ONE E ONE TO MANY

Creiamo la classe ispirandoci allo schema del database, seguendo gli stessi criteri utilizzati per la classe RoleEntity.

Aggiungiamo il vincoloManyToOne relativo all'entity EmployeeEntity con le seguenti annotation:

- @ManyToOne(fetch = FetchType.LAZY)
Indichiamo una relazione molti ad uno con il FetchType LAZY nel caso in cui si voglia stabilire che i dati relativi all'entità vengano “joinati” automaticamente.
Altrimenti, specifichiamo una relazione con il FetchType Eager per far sì che i dati della relazione vengano ricavati solo ed esclusivamente quando viene invocato il metodo GET.
- @JoinColumn(name = “role_name”)
Indichiamo che l'attributo role è una ForeignKey relativa all'entity “RoleEntity” con PrimaryKey su “role_name”.

AGGIUNGERE MANY TO ONE E ONE TO MANY

Andiamo ad aggiungere ora il vincolo OneToMany relativo all'entity Employee nella classe RoleEntity con le seguenti annotation:

- `@OneToMany(fetch = FetchType.LAZY)`
Indichiamo una relazione uno a molti con il FetchType LAZY nel caso in cui si voglia stabilire che i dati relativi all'entità vengano “joinati” automaticamente.
Tramite la proprietà mappedBy, rendiamo l'associazione bidirezionale.
Qui, il valore della proprietà, è il nome dell'attributo di mapping dell'associazione sul lato proprietario.

AGGIUNGERE TUTTE LE RELAZIONI

```
@Entity
@Table(name = "employee")
public class EmployeeEntity {

    @Id
    @Column(name = "cf")
    private String cf;
    @Column(name = "name")
    private String name;
    @Column(name = "surname")
    private String surname;
    @Column(name = "date")
    private Date date;
    @Column(name = "place")
    private String place;

    @ManyToOne(fetch = FetchType.LAZY,
                cascade = CascadeType.PERSIST,
                targetEntity = RoleEntity.class)
    @JoinColumn(name="role_name")
    private RoleEntity role;

    public String getCf() {
        return cf;
    }

    public void setCf(String cf) {
        this.cf = cf;
    }

    // ... altri getter e setter generati tramite eclipse ...
}
```

Creiamo la classe ispirandoci allo schema del database, seguendo gli stessi criteri utilizzati per la classe RoleEntity.

Aggiungiamo il vincolo ManyToOne relativo all'entity RoleEntity con le seguenti annotation:

- **@ManyToOne(fetch = FetchType.LAZY)**
Indichiamo una relazione uno a molti con il FetchType LAZY nel caso in cui si voglia stabilire che i dati relativi all'entità vengano “joinati” automaticamente.
Altrimenti, specifichiamo una relazione con il FetchType Eager per far sì che i dati della relazione vengano ricavati solo ed esclusivamente quando viene invocato il metodo GET.
- **@JoinColumn(name = “role_name”)**
Indichiamo che l'attributo role è una ForeignKey relativa all'entity “RoleEntity” con PrimaryKey su “role_name”.

AGGIUNGERE TUTTE LE RELAZIONI

```
@Entity
@Table(name="role")
public class RoleEntity {

    @Id
    @Column(name = "role_name")
    private String roleName;

    @Column(name = "salary")
    private double salary;

    @OneToMany(fetch = FetchType.LAZY,
        targetEntity = EmployeeEntity.class,
        cascade = CascadeType.PERSIST,
        mappedBy = "role")
    private List<EmployeeEntity> employeeList
        = new ArrayList<EmployeeEntity>();

    public String getRoleName() {
        return roleName;
    }

    public void setRoleName(String roleName) {
        this.roleName = roleName;
    }

    //... altri getter and setter ...
}
```

Modifichiamo la classe RoleEntity aggiungendole una lista di employee che conterrà tutti i lavoratori associati a quel ruolo:

Aggiungiamo il vincolo OneToMany relativo all'entity EmployeeEntity con le seguenti annotation:

- @OneToMany(fetch = FetchType.LAZY)
Indichiamo una relazione uno a molti con il FetchType LAZY nel caso in cui si voglia stabilire che i dati relativi all'entità vengano “joinati” automaticamente.

Creiamo nel package `api_model`, la classe `EmployeeAPIModel` che rappresenta il DTO relativo alla classe `EmployeeEntity`.

```
public class EmployeeAPIModel {  
  
    private String cf;  
    private String name;  
    private String surname;  
    private Date date;  
    private String place;  
  
    private RoleForEmployeeAPIModel role;  
  
    public String getCf() {  
        return cf;  
    }  
  
    public void setCf(String cf) {  
        this.cf = cf;  
    }  
  
    // ... altri getter e setter generati ...  
}
```

Questa conterrà per l'appunto solo ed esclusivamente i campi che dovranno essere riportati come richiesta o risposta ai servizi che andremo poi a creare.

Ad esempio, in questo caso andremo ad inserire i dati dell'employee ed il ruolo ad esso relativo.

Dato che nell'entità dell'employee, c'è una relazione `ManyToOne` con l'entità del ruolo, abbiamo la necessità di creare un apposito DTO per rappresentarlo.

Nel package adapter, procediamo a creare la classe EmployeeAdapter con i metodi :

```
public class EmployeeAdapter {
    public static EmployeeAPIModel to_API_model(EmployeeEntity employeeEntity) {
        EmployeeAPIModel employeeAPI = new EmployeeAPIModel();
        employeeAPI.setCf(employeeEntity.getCf());
        employeeAPI.setName(employeeEntity.getName());
        employeeAPI.setSurname(employeeEntity.getSurname());
        employeeAPI.setDate(employeeEntity.getDate());
        employeeAPI.setPlace(employeeEntity.getPlace());

        if(employeeEntity.getRole() != null) {
            employeeAPI.setRole(RoleForEmployeeAdapter.to_API_model(employeeEntity.getRole()));
        }

        return employeeAPI;
    }

    public static EmployeeEntity to_entity_model(EmployeeAPIModel employeeAPI) {
        EmployeeEntity employeeEntity = new EmployeeEntity();
        employeeEntity.setCf(employeeAPI.getCf());
        employeeEntity.setName(employeeAPI.getName());
        employeeEntity.setSurname(employeeAPI.getSurname());
        employeeEntity.setDate(employeeAPI.getDate());
        employeeEntity.setPlace(employeeAPI.getPlace());

        if(employeeAPI.getRole() != null) {
            employeeEntity.setRole(RoleForEmployeeAdapter.to_entity_model(employeeAPI.getRole()));
        }
        return employeeEntity;
    }

    public static List<EmployeeAPIModel> to_API_model_list(List<EmployeeEntity>
listEmployeeEntity){
        List<EmployeeAPIModel> listEmployeeAPI = new ArrayList<EmployeeAPIModel>();
        for (EmployeeEntity employee : listEmployeeEntity) {
            listEmployeeAPI.add(to_API_model(employee));
        }
        return listEmployeeAPI;
    }

    public static List<EmployeeEntity> to_entity_model_list(List<EmployeeAPIModel>
listEmployeeAPIModel){
        List<EmployeeEntity> listEmployeeEntity = new ArrayList<EmployeeEntity>();
        for (EmployeeAPIModel employee : listEmployeeAPIModel) {
            listEmployeeEntity.add(to_entity_model(employee));
        }
        return listEmployeeEntity;
    }
}
```

- to_API_model: permette di convertire l'entity dell'empolyee presa in input facendo così ritornare il DTO ad esso relativo

- to_entity_model: permette di convertire il DTO dell'empolyee preso in input facendo così ritornare l'entity ad esso relativo

- to_API_model_list: converte una lista di entity dell'employee in una lista di DTO ad essa relativa.

- to_entity_model_list: converte una lista di DTO dell'employee in una lista di entity ad essa relativa.

Procediamo col creare le interfacce EmpolyeeRepository.

```
@Repository  
public interface EmployeeRepository extends JpaRepository<EmployeeEntity, String> {  
}
```

@Repository è un'annotazione Spring che indica che la classe decorata è un repository.

Una repository è un meccanismo per incapsulare il comportamento di archiviazione, recupero e ricerca che emula una raccolta di oggetti.

Tutte le repository estendono l'interfaccia JpaRepository<T, ID> dove :

- T rappresenta la classe Entity
- ID rappresenta il tipo della relativa chiave primaria

Quando abbiamo invece una chiave primaria composta, come ad esempio per la nostra entità dei contatti, procediamo in maniera differente. Creiamo una classe apposita *ContactKey* per la chiave primaria composta.

JPA fornisce l'annotazione **@Embeddable** per dichiarare che una classe sarà incorporata da altre entità. Definiamo una classe per astrarre i dettagli della persona di contatto:

```
@Embeddable
public class ContactKey implements Serializable{

    /**
     *
    */
    private static final long serialVersionUID = -7699047239283212700L;

    @Column(name = "kind")
    private String kind;
    @Column(name = "value")
    private String value;
    @Column(name = "cf")
    private String cf;

    public ContactKey() {
    }

    public String getKind() {
        return kind;
    }

    public void setKind(String kind) {
        this.kind = kind;
    }

    // ... vari metodi getter e setter ...
}
```

La serializzazione è la conversione dello stato di un oggetto in un flusso di byte, ovvero è la conversione di un oggetto Java in un flusso statico (sequenza) di byte, che possiamo quindi salvare su un database o trasferire su una rete.

Le classi idonee per la serializzazione devono implementare un'interfaccia marcatore speciale, *Serializable*.

Poi, quando andremo a creare la classe per rappresentare il contatto, procediamo col passare nell'annotazione `@IdClass`, la classe che rappresenta la chiave primaria composta per poi procedere con l'implementazione di routine.

```
@Entity
@Table(name="contact")
@IdClass(ContactKey.class)
public class ContactEntity {

    @Id
    @Column(name="kind")
    private String kind;

    @Id
    @Column(name="value")
    private String value;

    @Id
    @Column(name="cf")
    private String cf;

    @Id
    @ManyToOne(fetch = FetchType.LAZY,
        targetEntity = EmployeeEntity.class,
        optional=false)
    @JoinColumn(name = "cf",
        insertable=false,
        updatable=false)
    private EmployeeEntity employee;

    public String getKind() {
        return kind;
    }

    public void setKind(String kind) {
        this.kind = kind;
    }

    // ... vari getter & setter ...
}
```

Anche quando andremo a creare l'apposita Repository, passeremo come ID non le singole componenti annotate con `@Id` nell'entità ma la classe serializzata.

```
@Repository
public interface ContactRepository extends JpaRepository<ContactEntity, ContactKey>{

}
```

Tutto quello che è stato implementato fino ad ora, dovrà essere esteso anche per le altre relazioni presenti all'interno del nostro UML del database.

Quindi andremo a creare delle specifiche classi di entity, model, adapter e repository per rappresentare le altre entità e poter eseguire operazioni di CRUD su di esse.

Clicchiamo [qui](#) per vedere l'implementazione completa del nostro applicativo.

Step 7

Completiamo le operazioni di CRUD

Una volta sviluppate le relazioni come spiegato nello step precedente, non ci resta che spiegare come implementare altre operazioni di CRUD.

Precedentemente sono state implementate le operazioni di **POST** per il *role* e **GET byId** di un ruolo. Adesso a titolo dimostrativo, effettueremo una **GET** di tutti gli *employee*, una **UPDATE** ed una **DELETE** di un *employee*.

Se non è già stata creata, nel package *com.restful.tutorial.service* creiamo una classe di transazione `EmployeeService` con le annotation `@Service` e `@Transactional` che conterranno le operazioni basilari di CRUD:

```
@Service
@Transactional
public class EmployeeService {
    @Autowired
    private EmployeeRepository employeeRepository;

    public List<EmployeeEntity> findAllE() {
        return employeeRepository.findAll();
    }

    public EmployeeEntity saveE(EmployeeEntity employeeEntity) {
        return employeeRepository.save(employeeEntity);
    }

    public void deleteById(String cf) {
        employeeRepository.deleteById(cf);
    }

    public EmployeeEntity getById(String cf) {
        return employeeRepository.findById(cf).get();
    }
}
```


Ora invece, nel package *com.restful.tutorial.service_operation* creiamo una classe *EmployeeServiceOperation* con l'annotation *@Service* che conterrà i metodi elaborati relativi all'entità

```
@Service
public class EmployeeServiceOperation {
    @Autowired
    private EmployeeService employeeService;
    @Autowired
    private RoleService     roleService;
    @Autowired
    private TaskService     taskService;
    @Autowired
    private ShiftService    shiftService;
}
```

dell'employee che implementeranno i metodi della classe in service proprio come abbiamo fatto precedentemente con l'entità *role*.

Per implementare un metodo che restituisca tutti gli employee nel formato in cui desideriamo, quindi dopo aver implementato una classe adapter che rispetti le nostre esigenze (per osservare come sono stati implementati cliccare [qui](#)), scriviamo il metodo come segue richiamando al suo interno il metodo *findAllE()* implementato precedentemente all'interno della classe service:

```
public List<EmployeeAPIModel> getEmployeeList(){
    return EmployeeAdapter.to_API_model_list(employeeService.findAllE());
}
```

Per implementare invece un metodo che aggiorni le informazioni di un employee, procediamo analogamente implementando un metodo nella classe *ServiceOperation* in questo modo:

```
public RequestSaveEmployeeAPIModel updateEmployee(RequestSaveEmployeeAPIModel employeeAPI) {
    EmployeeEntity employeeFound = employeeService.getById(employeeAPI.getCf());
    if(employeeFound!=null) {
        EmployeeEntity employeeEntity = RequestSaveEmployeeAdapter.to_entity_model(employeeAPI);
        RoleEntity roleE = roleService.findRole(employeeEntity.getRole().getRoleName());
        if(roleE!=null) {
            employeeEntity.setRole(roleE);
            return RequestSaveEmployeeAdapter.to_API_model(employeeService.saveE(employeeEntity));
        }else {
            throw new NoSuchElementException(" ERROR: role "+employeeEntity.getRole().getRoleName()+" not foud");
        }
    }else {
        throw new NoSuchElementException(" ERROR: employee "+employeeAPI.getCf()+" not foud");
    }
}
```

Praticamente funziona in questo modo: andiamo innanzitutto a cercare se l'employee inserito con le relative informazioni esiste e sono corrette. In caso di successo, andiamo a “sovrascrivere” le informazioni del vecchio employee con quelle passate in input con il metodo `saveE(employeeEntity)`. Altrimenti, verrà “triggerata” un’eccezione.

Procediamo analogamente per l'implementazione di un metodo che assegni il *task* o un *turno* all'*operaio*:

```
public ResponceTaskAssignmentAPIModel taskAssignment(RequestTaskAssignmentAPIModel taskAssignmentAPIModel) {
    EmployeeEntity employee = employeeService.getById(taskAssignmentAPIModel.getCf());

    if(employee!=null) {
        TaskEntity task = taskService.getById(taskAssignmentAPIModel.getIdTask());
        if(task != null) {
            employee.getTaskList().add(task);
            employee = employeeService.saveE(employee);

            return ResponceTaskAssignmentAdapter.to_API_model(employee, task);
        }
        else
            throw new NoSuchElementException(" ERROR: task "+taskAssignmentAPIModel.getIdTask()+" not foud");
    }
    else
        throw new NoSuchElementException(" ERROR: employee "+taskAssignmentAPIModel.getCf()+" not foud");
}
```

Mentre per il metodo di delete di una specifica unità procediamo in tal modo:

```
public void deleteById(String cf){
    employeeService.deleteById(cf);
}
```

All'interno del package controller, andiamo a definire la classe EmployeeController con lo scopo di esporre i nostri servizi REST come già spiegato precedentemente:

```
@RestController
public class EmployeeController {

    @Autowired
    private EmployeeServiceOperation empOp;

    @GetMapping("/employee_list")
    public List<EmployeeAPIModel> list() {
        return empOp.getEmployeeList();
    }

    @DeleteMapping("/delete_employee")
    public void deleteById(@RequestBody String cf) {
        empOp.deleteById(cf);
    }

    @PutMapping("update_employee")
    public RequestSaveEmployeeAPIModel updateEmployee(@RequestBody RequestSaveEmployeeAPIModel employeeAPI) {
        return empOp.updateEmployee(employeeAPI);
    }

    @PostMapping("/task_assignment")
    public ResponceTaskAssignmentAPIModel taskAssignment(@RequestBody RequestTaskAssignmentAPIModel taskAssignment) {
        return empOp.taskAssignment(taskAssignment);
    }
}
```

notiamo anche che, tramite le annotazioni @GetMapping, @DeleteMapping, @PutMapping e @PostMapping, andiamo a definire la tipologia del servizio esposto.

A questo punto, non ci resta che implementare tutti i servizi di cui abbiamo bisogno per le varie entità ed effettuare nuovamente il deploy della nostra applicazione per poterli testare tramite postman.

Step 8

SWAGGER

Per lo sviluppo delle API è essenziale una documentazione adeguata e comprensibile. Senza quest'ultima gli sviluppatori non possono utilizzare le interfacce. Ciò è essenziale soprattutto per le API di pubblico dominio: senza documentazione, queste sono inutili per la comunità, non possono essere distribuite e di conseguenza non riscuotono alcun successo.

Utilizzare Swagger è attualmente il modo migliore per documentare le API REST, poiché è in grado di mappare quasi tutti i servizi web e le informazioni relative all'interfaccia. Swagger evolve con il sistema e tutte le modifiche apportate vengono documentate automaticamente. La capacità di Swagger di memorizzare la documentazione dell'interfaccia REST direttamente sul codice è il segreto del suo buon funzionamento.

Per utilizzare questa libreria, la prima cosa da fare è quello di importarla nel nostro progetto sempre mediante l'utilizzo del file *pom.xml*

Procediamo con copiare all'interno delle dependencies il seguente codice :

```
<!-- SWAGGER -->
<dependency>
  <groupId>io.springfox</groupId>
  <artifactId>springfox-swagger2</artifactId>
  <version>2.9.2</version>
</dependency>

<dependency>
  <groupId>io.springfox</groupId>
  <artifactId>springfox-swagger-ui</artifactId>
  <version>2.9.2</version>
</dependency>
```

A questo punto, possiamo utilizzarlo all'interno del nostro progetto.

Ora seguiamo col creare una Classe denominata *SwaggerConfig* in *com.restful.tutorial*:



Aggiungendo l'annotazione *@EnableSwagger2* nell'applicazione, Spring Boot abiliterà la libreria sulla classe.

```
@Configuration
@EnableSwagger2
public class SwaggerConfig {

    @Bean
    public Docket api() {
        /*
         * return new Docket(DocumentationType.SWAGGER_2) .select()
         * .apis(RequestHandlerSelectors.any()) .paths(PathSelectors.any()) .build()
         * .apiInfo(this.apiInfo()) .useDefaultResponseMessages(false);
         */

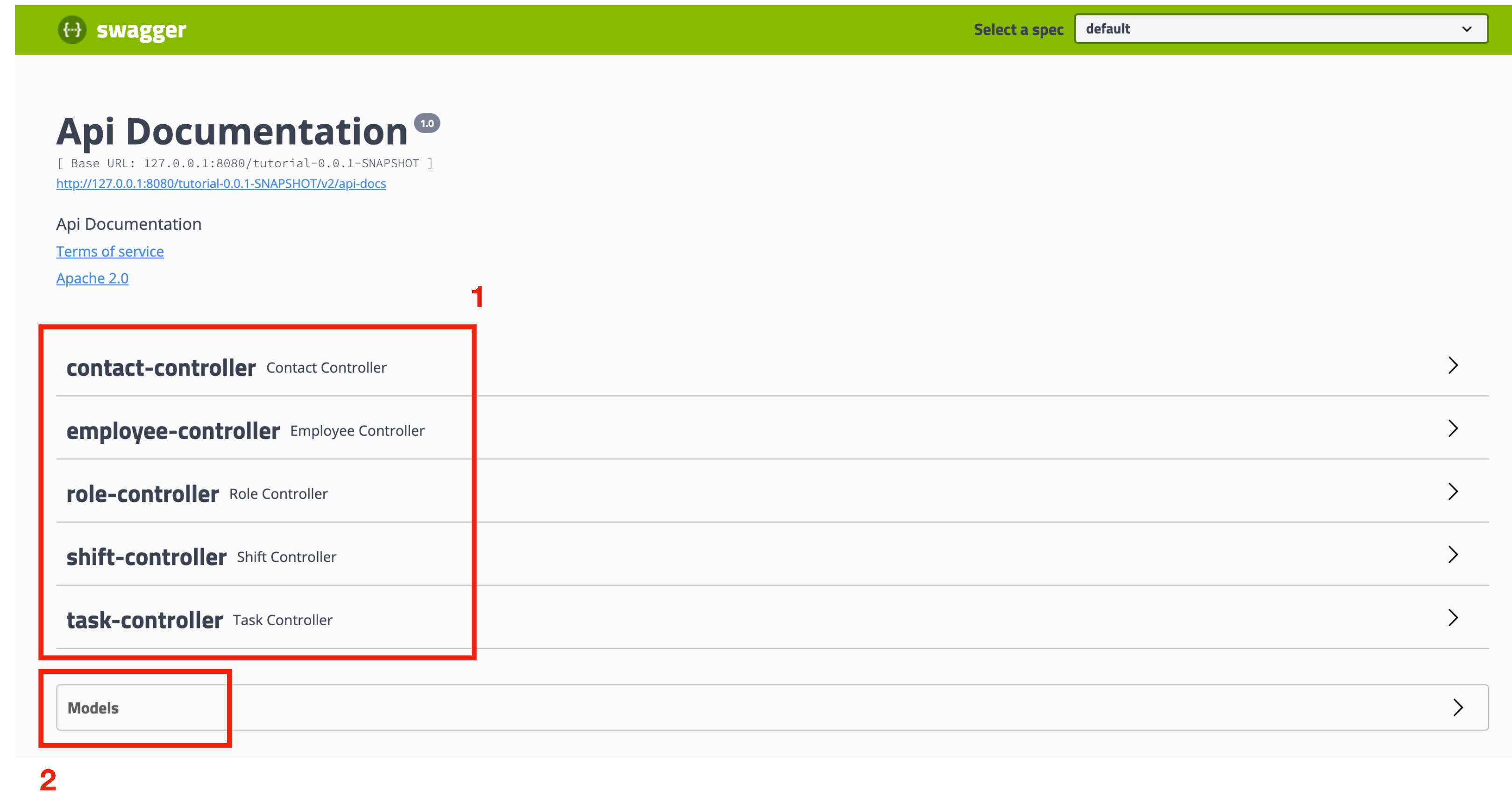
        // Remove Basic Error Controller In SpringFox SwaggerUI
        return new Docket(DocumentationType.SWAGGER_2).select().apis(RequestHandlerSelectors.any())
            .paths(PathSelectors.any()).paths(Predicates.not(PathSelectors.regex("/error.*"))).build();
    }

    @SuppressWarnings("unused")
    private ApiInfo apiInfo() {
        ApiInfoBuilder apiInfoBuilder = new ApiInfoBuilder();
        apiInfoBuilder.title("REST API");
        apiInfoBuilder.description("REST API Generation");
        apiInfoBuilder.version("1.0.0");
        apiInfoBuilder.license("GNU GENERAL PUBLIC LICENSE, Version 3");
        apiInfoBuilder.licenseUrl("https://www.gnu.org/licenses/gpl-3.0.en.html");
        return apiInfoBuilder.build();
    }
}
```

Tramite il metodo *Docket api()* andiamo a targhettare l'intero progetto e successivamente a rimuovere gli errori di Controller di base in SpringFox SwaggerUI.

Tramite il metodo *ApiInfo apiInfo()* andiamo ad inserire eventuali informazioni utili del progetto.

Una volta effettuato il deploy, basterà aggiungere /swagger-ui.html al context root per navigare su una pagina simile :



- Nel rettangolo **1** vengono riportati tutti i controller presenti all'interno del progetto e cliccando su uno di essi verranno mostrati i relativi servizi.
- Nel rettangolo **2** vengono riportati tutti i model creati all'interno del progetto.

employee-controller Employee Controller		
DELETE	/delete_employee	deleteById
GET	/employee_by_cf	getById
GET	/employee_list	list
POST	/save_employee	saveE
POST	/shift_assignment	shiftAssignment
POST	/task_assignment	taskAssignment
PUT	/update_employee	updateEmployee

Cliccando ad esempio su **employee-controller** possiamo accedere a tutti i servizi ad esso relativi ed avere informazioni riguardo:

- Il tipo di metodo dell'http
- La sua request
- La sua response

In più possiamo utilizzare i vari servizi da interfaccia grafica mediante il bottone **try it out**.

Grazie

Carlo Lomello & Francesco Manzo