APPUNTI JAVA SPRING E HIBERNATE

Indice generale

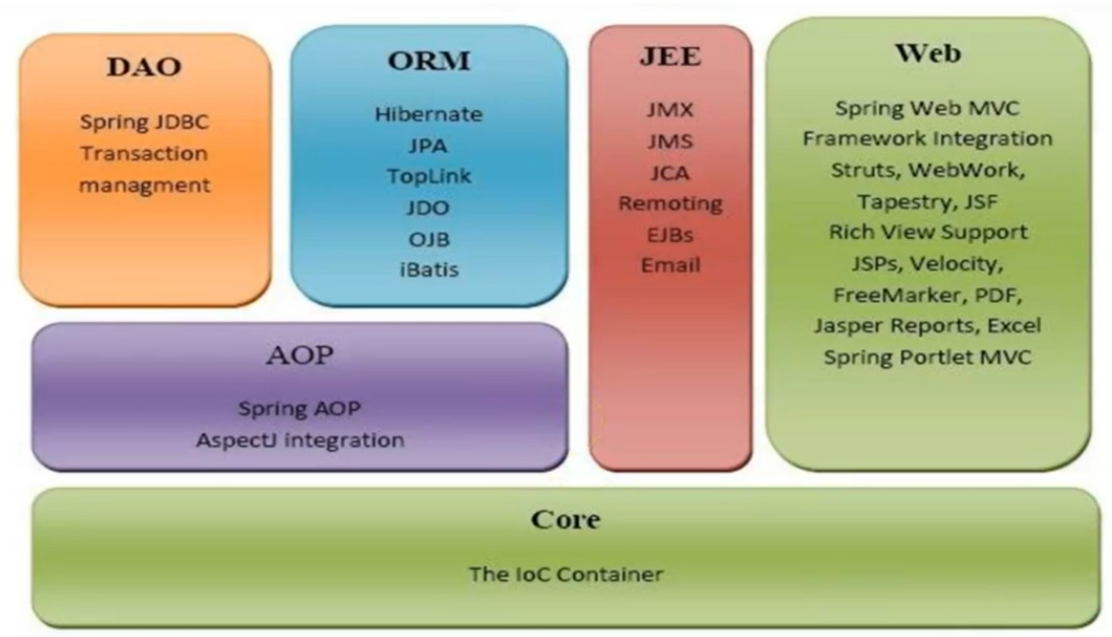
SPRING E SPRING BOOT

Spring è un framework di JAVA.

Dalla versione 5.0.2 diventa il framework più popolare per lo sviluppo di applicazioni JAVA EE. Infatti, quando si sente parlare di framework di JAVA, si sente sempre e solo parlare di Spring.

Spring è un framework con una struttura modulare, dove una parte dei moduli sono facoltativi, ma uno in particolare è obbligatorio. I moduli possiamo utilizzarli in parte, o tutti, senza stravolgere l'architettura del progetto.

Qui di seguito possiamo vedere la rappresentazione dei moduli di Spring:

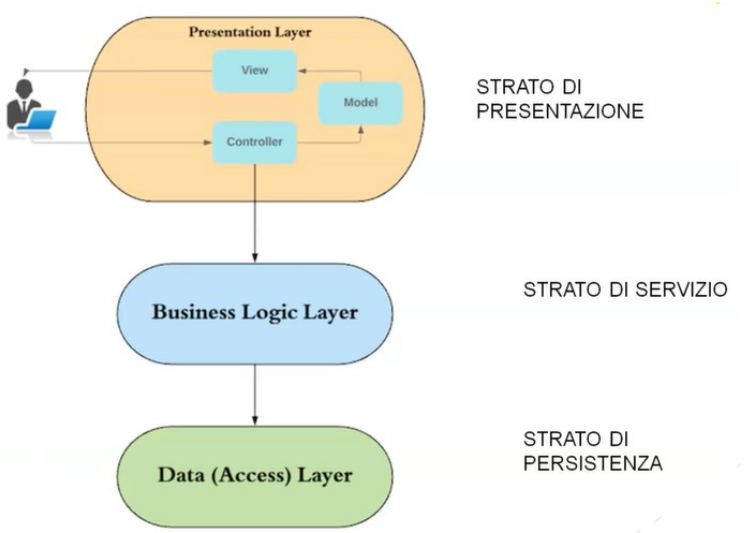


Il modulo obbligatorio di cui parlavamo è il Core, che è il motore del framework. Gli altri moduli possono essere utilizzati facoltativamente.

DESIGN PATTERN

Un design pattern è uno schema architetturale da seguire che semplifica lo sviluppo di applicazioni enterprise. In pratica è una linea guida per scrivere codice pulito, manutenibile e riutilizzabile.

L'MVC (Model View Controller) è un design pattern che ha come scopo la divisione del modello, della vista e del controller.



Quando si sviluppa un'applicazione web, si hanno 3 strati:

- strato di presentazione o presentation layer;
- strato di servizio o business logic layer;
- strato di persistenza o data access layer;

Quando un utente manda una richiesta, essa viene intercettata da un controller. Il controller comunica con lo strato di servizio, al cui interno sono presenti tutte le logiche di business. I dati che elaborerà lo strato di servizio, verranno presi dallo strato di persistenza, che a sua volta comunicherà con un database. Quindi, all'interno dello strato di persistenza verranno effettuate delle query per interagire con il database. La risposta verrà recuperata dallo strato di persistenza, dopodichè verrà richiamata dallo strato di servizio per elaborarla. Il risultato di tale elaborazione viene restituito al controller che, a sua volta, lo passerà alla view, in maniera tale che l'utente possa visualizzare la risposta.

SPRING BOOT

Sebbene Spring offra numerosi vantaggi allo sviluppatore, uno dei suoi punti negativi è la sua difficoltà di configurazione. Spring Boot nasce con lo scopo di risolvere questo problema.

Spring Boot è un progetto Spring che ne semplifica la configurazione e ne permette l'esecuzione senza server, avendone già uno integrato.

CONFIGURAZIONE DI SPRING

La configurazione di Spring può avvenire in 3 modi:

- file .xml;
- classi di configurazione JAVA;
- annotation (che utilizzeremo in questo corso).

Vediamo una lista di annotation più utilizzate in Spring:

- @Component
- @Controller (@RestController)
- @Service
- @Repository
- @RequestParam
- @RequestMapping
- @PathVariable
- @RequestBody
- @Autowired
- @ComponentScan
- @Bean
- @Configuration

IOC (INVERSION OF CONTROL) E DI (DEPENDENCY INJECTION)

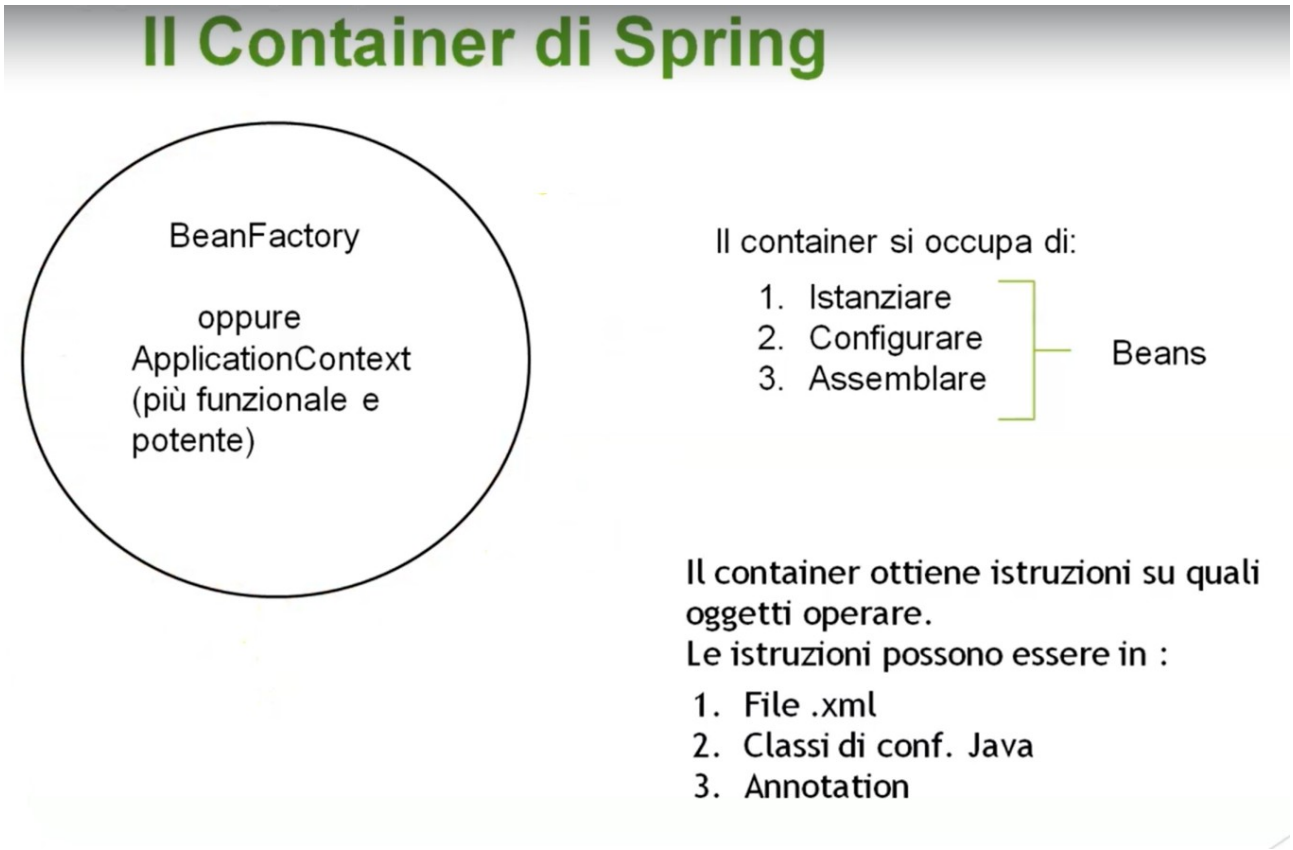
I due concetti che andremo a vedere di seguito sono fondamentali per il funzionamento di Spring.

L'inversion of control è un principio architetturale (o design pattern) basato sull'inversione del flusso del sistema. Come già spiegato, un design pattern è uno schema da seguire per permettere un più facile sviluppo, manutenibilità e riusabilità del codice.

Attraverso l'inversion of control, non è più il programmatore a doversi occupare di creare e istanziare oggetti, o invocare metodi, ma lo farà Spring attraverso una certa configurazione.

La dependency injection è un'implementazione dell'inversion of control e permette di iniettare le dipendenze di una classe all'esterno, senza che tali dipendenze vengano inizializzate dalla classe stessa.

Partiamo con un esempio teorico:

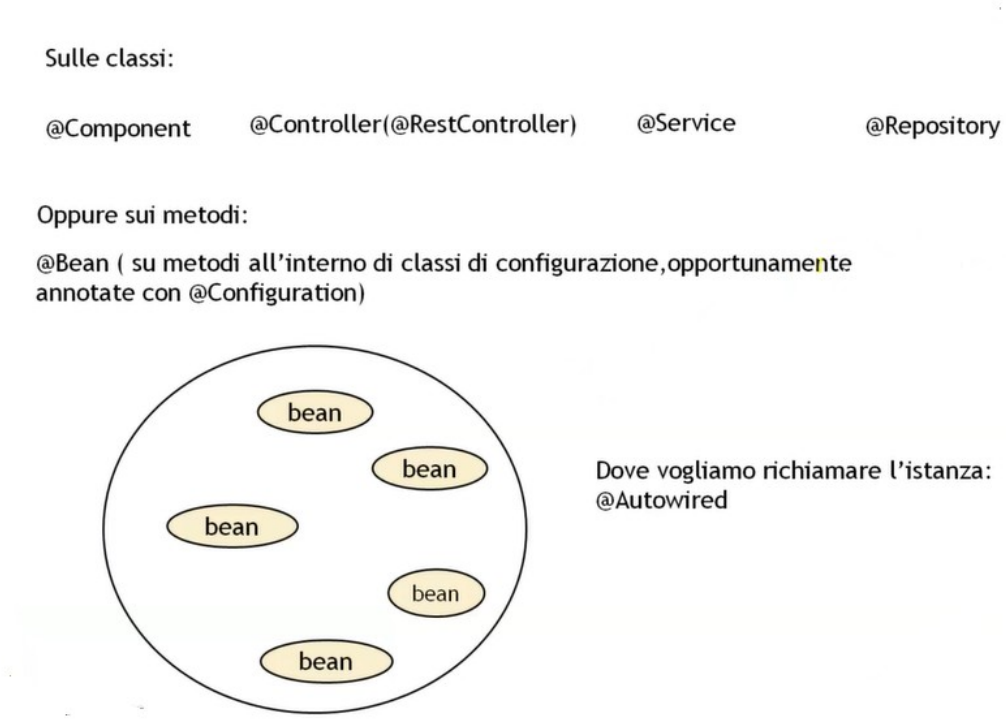


Spring ha un container che sta dietro le quinte. Il container è un'istanza di BeanFactory, oppure di ApplicationContext. Tale container, con il fatto che applica la dependency injection, si occupa di istanziare, configurare e assemblare i Beans. Il container deve essere però configurato in uno dei 3 modi elencati quando abbiamo parlato delle configurazioni di Spring, ossia:

- file .xml;
- classi di configurazione JAVA;
- annotation.

Sulle classi, ogni qualvolta mettiamo un annotation tra @Component, @Controller(@RestController), @Service, o @Repository, Spring le trasforma in un bean e lo mette all'interno del container. Quando poi ci serve l'istanza di quella classe, utilizziamo l'annotation @Autowired per andare a pescare il bean messo precedentemente nel container.

La stessa operazione può essere fatta mettendo l'annotation @Bean sopra alcuni metodi che si trovano nelle classi di configurazione, opportunamente annotate con @Configuration.



Vediamo dal punto di vista pratico questo concetto:

Persona.java

```java
package dependency_injection;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

/* Quando inizializziamo il progetto, Spring va a vedere tutte le classi
 * con l'annotation @Component, ne crea un bean e lo inserisce nel suo
 * cantainer
 */
@Component
public class Persona {
    String nome;
    String cognome;

    @Autowired
    Indirizzo indirizzo;
}
```

Indirizzo.java

```java
package dependency_injection;
import org.springframework.stereotype.Component;

@Component
public class Indirizzo {
    String via;
    Integer civico;
}
```

MainDependencyInjection.java

```java
package com.example.Corso_Spring_Hibernate;

import dependency_injection.Persona;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class CorsoSpringHibernateApplication {

    /* L'annotation @Autowired va sopra la dichiarazione
    * dell'istanza di una classe annotata precedentemente
    * con l'annotation @Component (vale anche per classi
    * annotate con @Controller(@RestController), @Service,
    * o @Repository). In questo modo non abbiamo bisogno
    * di inizializzare l'istanza con new,
    * dato che lo fa già Spring dietro le quinte */
    @Autowired
    static Persona p;

    public static void main(String[] args) {
        SpringApplication.run(CorsoSpringHibernateApplication.class, args);
        p.nome = "";
        p.cognome = "";
    }
}
```

ALTRO ESEMPIO DI DEPENDENCY INJECTION

Nel paragrafo precedente abbiamo visto come avviene la dependency injection quando annotiamo una classe con @Component. Lo stesso principio vale per @Controller(@RestController), @Service e @Repository.

Abbiamo però anche un altro tipo di dependency injection, che serve perlopiù a configurare e fa riferimento ad uno dei 3 modi per configurare Spring (o per meglio dire, il container di Spring), ossia il metodo con le annotation. Configurare attraverso le annotation, si intende utilizzando l'annotation @Bean su metodi di classi di configurazione, opportunamente annotate con @Configuration.
Vediamo un esempio:

Veicolo.java

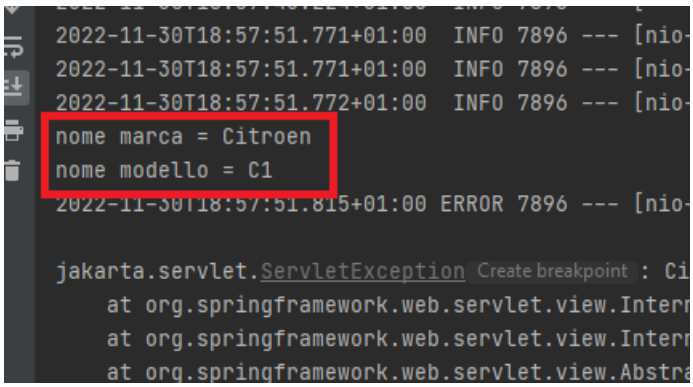
```
package com.example.Corso_Spring_Hibernate.dependency_injection;

public class Veicolo {
    String nomeMarca;
    String nomeModello;

    public String getNomeMarca() {
        return nomeMarca;
    }

    public void setNomeMarca(String nomeMarca) {
        this.nomeMarca = nomeMarca;
    }

    public String getNomeModello() {
        return nomeModello;
    }

    public void setNomeModello(String nomeModello) {
        this.nomeModello = nomeModello;
    }
}
```

VeicoloConfiguration.java

```
package com.example.Corso_Spring_Hibernate.dependency_injection;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class VeicoloConfiguration {

    /*All'interno di una classe di configurazione (annotata con @Configuration)
      si devono creare uno, o più metodi, annotati con @Bean che generano delle classi.
      All'interno del metodo, o dei metodi, si configura la classe che devono restituire*/
    @Bean
    public Veicolo configuraVeicolo(){
        Veicolo veicolo = new Veicolo();
        veicolo.setNomeMarca("Citroen");
        veicolo.setNomeModello("C1");
        return veicolo;
    }
}
```

CorsoSpringHibernateApplication.java

```java
package com.example.Corso_Spring_Hibernate;

import com.example.Corso_Spring_Hibernate.dependency_injection.Persona;
import com.example.Corso_Spring_Hibernate.dependency_injection.Veicolo;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;

@SpringBootApplication
@Controller
@RequestMapping("/")
public class CorsoSpringHibernateApplication {

    /* L'annotation @Autowired va sopra la dichiarazione
     * dell'istanza di una classe annotata precedentemente
     * con l'annotation @Component (vale anche per classi
     * annotate con @Controller(@RestController), @Service,
     * o @Repository). In questo modo non abbiamo bisogno
     * di inizializzare l'istanza con new,
     * dato che lo fa già Spring dietro le quinte */
    @Autowired
    Persona p;

    /*In questo caso, invece, l'annotation @Autowired sta
     * sopra la dichiarazione dell'istanza di una classe
     * che è stata configurata mediante un metodo @Bean
     * all'interno di una classe annotata con @Configuration */
    @Autowired
    Veicolo v;

    public static void main(String[] args) {
        SpringApplication.run(CorsoSpringHibernateApplication.class, args);
    }

    @RequestMapping("/")
    public void stampaVeicolo(){
        System.out.println("nome marca = " + v.getNomeMarca() + "\n" +
                           "nome modello = " + v.getNomeModello());
    }
}
```



ANNOTATION @QUALIFIER

Nel paragrafo precedente abbiamo visto un esempio di configurazione mediante una classe con l'annotation @Configuration, dove abbiamo definito un metodo @Bean che restituisce un oggetto di tipo Veicolo. In seguito, tramite @Autowired, abbiamo recuperato i dati dell'oggetto di tipo Veicolo e li abbiamo stampati.

Il problema però sorge nel momento in cui andiamo a creare più metodi @Bean che restituiscono oggetti dello stesso tipo (in questo caso Veicolo), ma configurati in maniera diversa. Quando andiamo a fare @Autowired per dichiarare un'istanza dello stesso tipo degli oggetti restituiti dai metodi @Bean, Spring ci da errore perché non sa a quale dei metodi @Bean deve fare riferimento. Per questo motivo ci viene in aiuto l'annotation @Qualifier, tramite il quale è possibile assegnare un nome ad ogni metodo @Bean, in modo da poterli distinguere.

Vediamo un esempio:

Veicolo.java

```
package com.example.Corso_Spring_Hibernate.dependency_injection;

public class Veicolo {
    String nomeMarca;
    String nomeModello;

    public String getNomeMarca() {
        return nomeMarca;
    }

    public void setNomeMarca(String nomeMarca) {
        this.nomeMarca = nomeMarca;
    }

    public String getNomeModello() {
        return nomeModello;
    }

    public void setNomeModello(String nomeModello) {
        this.nomeModello = nomeModello;
    }
}
```

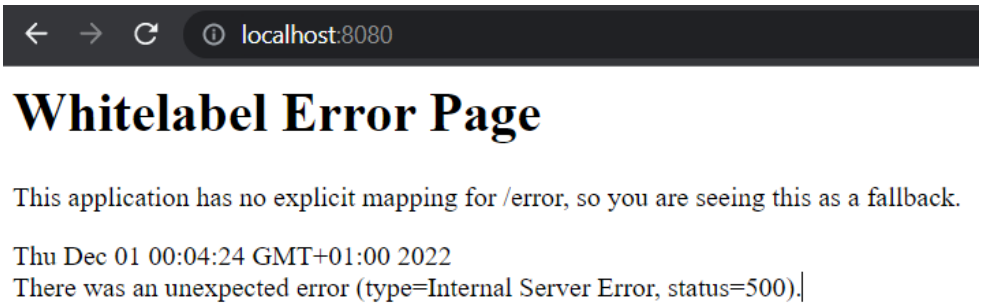
VeicoloConfiguration.java

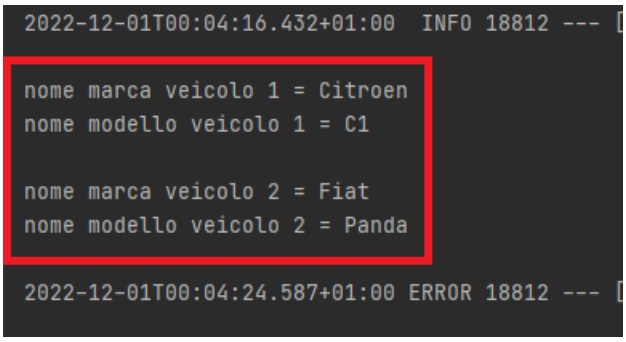
```java
package com.example.Corso_Spring_Hibernate.dependency_injection;

import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class VeicoloConfiguration {

    /*All'interno di una classe di configurazione (annotata con @Configuration)
      si devono creare uno, o più metodi, annotati con @Bean che generano delle classi.
      All'interno del metodo, o dei metodi, si configura la classe che devono restituire*/
    @Bean
    @Qualifier("1")
    public Veicolo configuraVeicolo(){
        Veicolo veicolo = new Veicolo();
        veicolo.setNomeMarca("Citroen");
        veicolo.setNomeModello("C1");
        return veicolo;
    }

    @Bean
    @Qualifier("2")
    public Veicolo configuraVeicolo2(){
        Veicolo veicolo = new Veicolo();
        veicolo.setNomeMarca("Fiat");
        veicolo.setNomeModello("Panda");
        return veicolo;
    }
}
```

CorsoSpringHibernateApplication.java

```java
package com.example.Corso_Spring_Hibernate;

import com.example.Corso_Spring_Hibernate.dependency_injection.Persona;
import com.example.Corso_Spring_Hibernate.dependency_injection.Veicolo;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;

@SpringBootApplication
@Controller
@RequestMapping("/")
public class CorsoSpringHibernateApplication {

    /* L'annotation @Autowired va sopra la dichiarazione
    * dell'istanza di una classe annotata precedentemente
    * con l'annotation @Component (vale anche per classi
    * annotate con @Controller(@RestController), @Service,
    * o @Repository). In questo modo non abbiamo bisogno
    * di inizializzare l'istanza con new,
    * dato che lo fa già Spring dietro le quinte */
    @Autowired
    Persona p;
```

```java
    /*In questo caso, invece, l'annotation @Autowired sta
    * sopra la dichiarazione dell'istanza di una classe
    * che è stata configurata mediante un metodo @Bean
    * all'interno di una classe annotata con @Configuration */
    @Autowired
    @Qualifier("1")
    Veicolo v;

    @Autowired
    @Qualifier("2")
    Veicolo v2;

    public static void main(String[] args) {
        SpringApplication.run(CorsoSpringHibernateApplication.class, args);
    }

    @RequestMapping("/")
    public void stampaVeicolo(){
        System.out.println("\nnome marca veicolo 1 = " + v.getNomeMarca() + "\n" +
                           "nome modello veicolo 1 = " + v.getNomeModello() + "\n");

        System.out.println("nome marca veicolo 2 = " + v2.getNomeMarca() + "\n" +
                           "nome modello veicolo 2 = " + v2.getNomeModello() + "\n");
    }
}
```



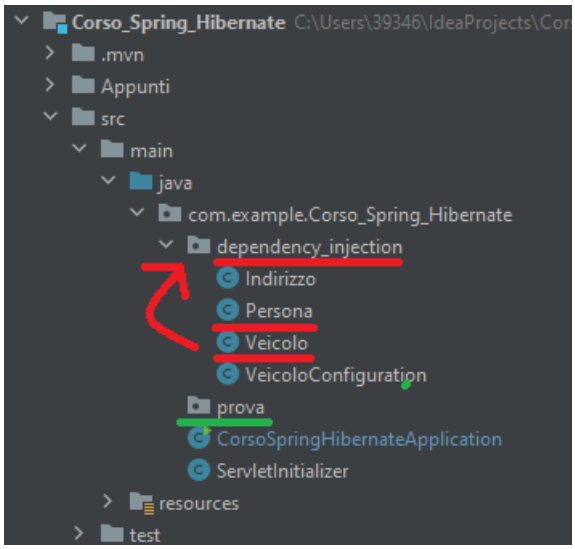


ANNOTATION @COMPONENTSCAN

Quando eseguiamo un applicazione creata il framework Spring, quest'ultimo di default parte dal package principale e va ad ispezionare tutte le sottocartelle. In particolare, va a cercare tutte le classi annotate con @Component, @Bean e le altre annotation già viste, le prende a suo carico e le immagazzina nel suo container, sotto forma di bean.

Possiamo comunque cambiare quello che è il comportamento di default di Spring, attraverso l'annotation @ComponentScan. Tramite @ComponentScan diciamo a Spring dove andare a trovare le classi che ci servono, quindi quelle che si prenderà in carico nel suo container e che deve trasformare in bean.

Per fare un esempio, possiamo creare un package "prova" senza classi al suo interno. Se inseriamo l'annotation @ComponentScan (con il path del package creato tra parentesi) nella classe dell'esempio precedente (CorsoSpringHibernateApplication.java), dove abbiamo istanziato Persona e Veicolo tramite @Autowired, Spring ci restituisce errore perché non riesce a trovare le classi Persona e Veicolo di cui richiediamo un'istanza.



CorsoSpringHibernateApplication.java

```
package com.example.Corso_Spring_Hibernate;

import com.example.Corso_Spring_Hibernate.dependency_injection.Persona;
import com.example.Corso_Spring_Hibernate.dependency_injection.Veicolo;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;

@SpringBootApplication
@Controller
@RequestMapping("/")
@ComponentScan("com.example.Corso_Spring_Hibernate.prova")
public class CorsoSpringHibernateApplication {
```

```
    /* L'annotation @Autowired va sopra la dichiarazione
    * dell'istanza di una classe annotata precedentemente
    * con l'annotation @Component (vale anche per classi
    * annotate con @Controller(@RestController), @Service,
    * o @Repository). In questo modo non abbiamo bisogno
    * di inizializzare l'istanza con new,
    * dato che lo fa già Spring dietro le quinte */
    @Autowired
    Persona p;



    /*In questo caso, invece, l'annotation @Autowired sta
    * sopra la dichiarazione dell'istanza di una classe
    * che è stata configurata mediante un metodo @Bean
    * all'interno di una classe annotata con @Configuration */
    @Autowired
    @Qualifier("1")
    Veicolo v;

    @Autowired
    @Qualifier("2")
    Veicolo v2;

    public static void main(String[] args) {
        SpringApplication.run(CorsoSpringHibernateApplication.class, args);
    }

    @RequestMapping("/")
    public void stampaVeicolo(){
        System.out.println("\nnome marca veicolo 1 = " + v.getNomeMarca() + "\n" +
                           "nome modello veicolo 1 = " + v.getNomeModello() + "\n");

        System.out.println("nome marca veicolo 2 = " + v2.getNomeMarca() + "\n" +
                           "nome modello veicolo 2 = " + v2.getNomeModello() + "\n");
    }
}
```

```
***************************
APPLICATION FAILED TO START
***************************

Description:

Field p in com.example.Corso_Spring_Hibernate.CorsoSpringHibernateApplication required
a bean of type 'com.example.Corso_Spring_Hibernate.dependency_injection.Persona' that
could not be found.

The injection point has the following annotations:
- @org.springframework.beans.factory.annotation.Autowired(required=true)
```

ANNOTATION @SCOPE

Come già detto, quando facciamo partire un'applicazione Spring, il compito dell'@Autowired è quello di trovare l'istanza di una classe nel container. Di default, questa istanza è singleton, cioè l'@Autowired va a chiamare sempre la stessa istanza e mai una nuova.

Possiamo decidere però di cambiare questo comportamento tramite l'annotation @Scope("prototype") (da inserire sopra la classe per la quale vogliamo creare più istanze), grazie al quale verrà creata un'istanza nuova ogni qualvolta si farà un'@Autowired. Da notare che se viene utilizzato @Scope("singleton"), il comportamento dell'@Autowired sarà uguale a quello di default.

Possiamo verificare ciò inserendo un contatore nel costruttore della classe istanziata (nel nostro caso Persona). Ogni qualvolta verrà creata una nuova istanza della classe, si avvierà il costruttore e conteggerà quante volte viene istanziata una classe. Mettiamo in pratica l'esempio con e senza @Scope:

Persona.java

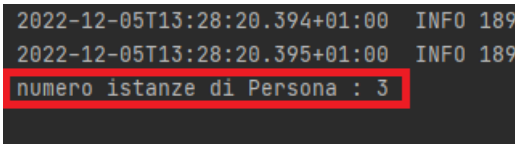
```
package com.example.Corso_Spring_Hibernate.dependency_injection;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Scope;
import org.springframework.stereotype.Component;


/* @Scoope("prototype") permette di creare un'istanza della classe ogni
 * qualvolta si farà l'@Autowired. Senza @Scope, o con @Scope("singleton"),
 * l'@Autowired chiama sempre la stessa istanza (singleton pattern)*/
@Scope("prototype")

/* Quando inizializziamo il progetto, Spring va a vedere tutte le classi
 * con l'annotation @Component, ne crea un bean e lo inserisce nel suo cantainer*/
@Component
public class Persona {
    public String nome;
    public String cognome;
    public static int contatore = 0;

    @Autowired
    Indirizzo indirizzo;

    public Persona() {
        contatore ++;
    }
}
```

ScopeAnnotation.java

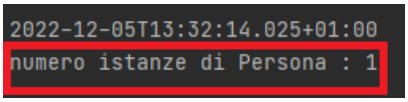
```java
package com.example.Corso_Spring_Hibernate.scoope_annotation;

import com.example.Corso_Spring_Hibernate.dependency_injection.Persona;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
@RequestMapping("/scope")
public class ScopeAnnotation {

    /* L'annotation @Autowired va sopra la dichiarazione
     * dell'istanza di una classe annotata precedentemente
     * con l'annotation @Component (vale anche per classi
     * annotate con @Controller(@RestController), @Service,
     * o @Repository). In questo modo non abbiamo bisogno
     * di inizializzare l'istanza con new,
     * dato che lo fa già Spring dietro le quinte */
    @Autowired
    Persona p;

    @Autowired
    Persona p2;

    @Autowired
    Persona p3;

    @RequestMapping("/stampa")
    public void stampa(){
        System.out.println("numero istanze di Persona : " + Persona.contatore);
    }
}
```

I vari comportamenti dell'esempio sono:

1) Con @Scope("prototype"):



2)Con @Scope("singleton"), o senza @Scope:



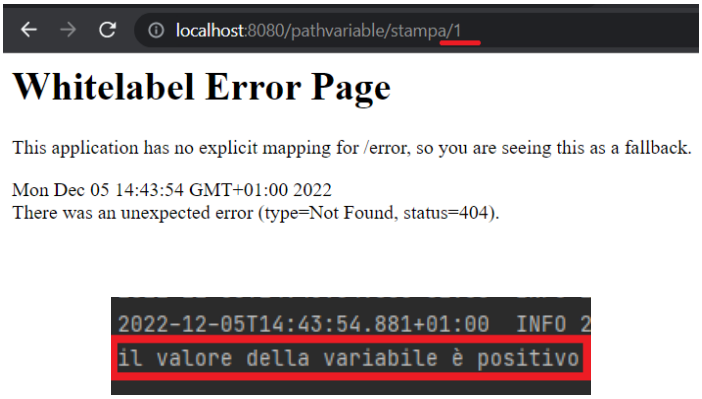
ANNOTATION @PATHVARIABLE

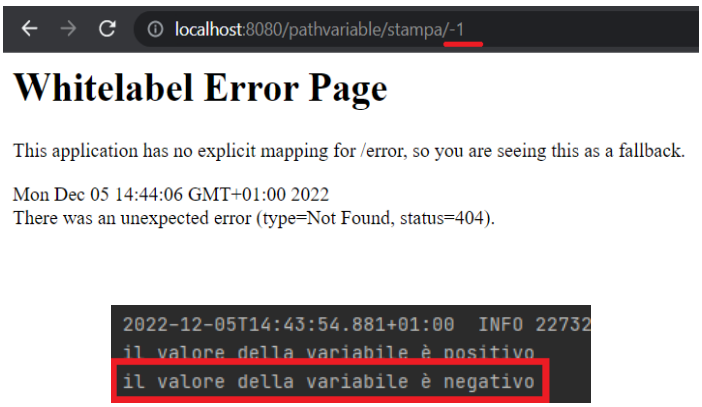
L'annotation @PathVariable permette di passare valori in ingresso ai metodi tramite il path impostato nel @RequestMapping. Vediamo un esempio:

PathVariableAnnotation.java

```
package com.example.Corso_Spring_Hibernate.pathvariable_annotation;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
@RequestMapping("/pathvariable")
public class PathVariableAnnotation {

    @RequestMapping("/stampa/{variabile}")
    public void stampa(@PathVariable int variabile){
        if(variabile < 0){
            System.out.println("il valore della variabile è negativo");
        } else {
            System.out.println("il valore della variabile è positivo");
        }
    }
}
```

Il risultato di questo esempio è il seguente:





ANNOTATION @REQUESTPARAM

L'Annotation @RequestParam è molto simile a @PathVariable, cioè permette di passare valori in ingresso ai metodi tramite path impostato nel @RequestMapping, con la differenza che questa volta si utilizzano le query string. Vediamo un esempio:

RequestParamAnnotation.java

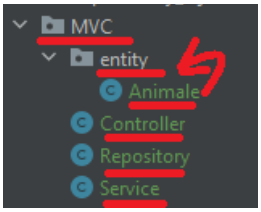
```
package com.example.Corso_Spring_Hibernate.requestparam_annotation;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;

@Controller
@RequestMapping("/requestparam")
public class RequestParamAnnotation {

        @RequestMapping("/stampa")
        public void stampa(@RequestParam int variabile) {
        if (variabile < 0) {
            System.out.println("il valore della variabile è negativo");
        } else {
            System.out.println("il valore della variabile è positivo");
        }
    }
}
```

Il risultato di questo esempio è il seguente:

ANNOTATION @CONTROLLER, @SERVICE, @REPOSITORY (MVC)

Le annotation @Controller, @Service e @Repository sono il cuore del pattern MVC di Spring.

Vediamo di seguito un esempio di programma costruito seguendo il pattern MVC in Spring:



Animale.java

```
package com.example.Corso_Spring_Hibernate.MVC.entity;

public class Animale {
    String classificazione;
    String razza;

    public Animale() {
    }

    public String getClassificazione() {
        return classificazione;
    }

    public void setClassificazione(String classificazione) {
        this.classificazione = classificazione;
    }

    public String getRazza() {
        return razza;
    }

    public void setRazza(String razza) {
        this.razza = razza;
    }
}
```

Repository.java

```java
package com.example.Corso_Spring_Hibernate.MVC;

import com.example.Corso_Spring_Hibernate.MVC.entity.Animale;

/* Normalmente nella classe annotata con @Repository, ci troviamo all'interno
* del Data Access Layer di un'applicazione web, dove di usano tecnologie
* (come Hibernate) per fare le query al database. Per questa volta però
* facciamo una finta chiamata ad un database ideale, giusto per capire il
* concetto che riguarda il design pattern MVC*/
@org.springframework.stereotype.Repository
public class Repository {


    public Animale getAnimaleRepository(){
        Animale animale = new Animale();
        animale.setClassificazione("Mammifero");
        animale.setRazza("Cane");

        return animale;
    }
}
```

Service.java

```java
package com.example.Corso_Spring_Hibernate.MVC;
import com.example.Corso_Spring_Hibernate.MVC.entity.Animale;
import org.springframework.beans.factory.annotation.Autowired;

/*La classe annotata con @Service deve restituire il risultato
* del metodo che abbiamo all'interno della classe annotata con @Repository*/
@org.springframework.stereotype.Service
public class Service {

    @Autowired
    Repository repository;

    public Animale getAnimaleService(){
        return repository.getAnimaleRepository();
    }
}
```

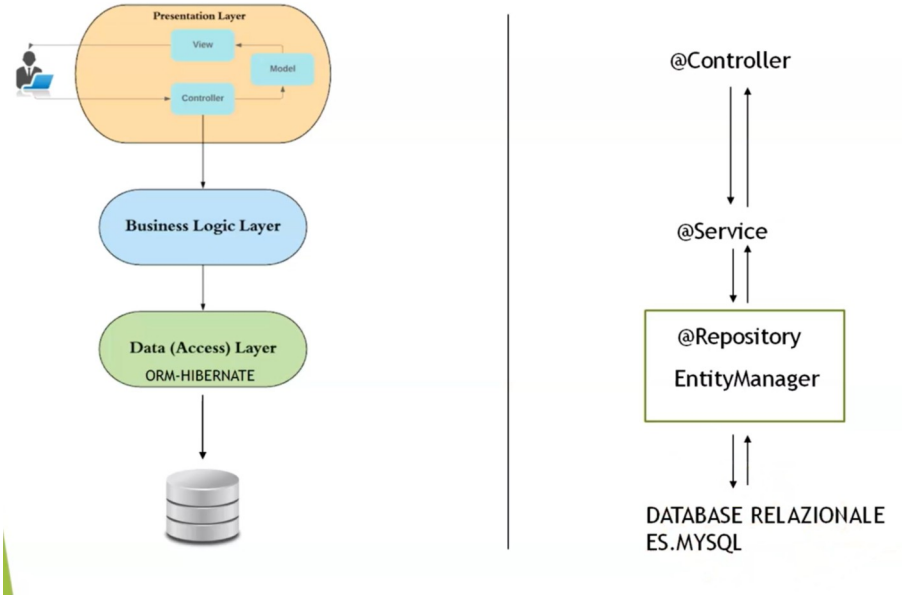
Controller.java

```java
package com.example.Corso_Spring_Hibernate.MVC;
import com.example.Corso_Spring_Hibernate.MVC.entity.Animale;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.RequestMapping;

/*La classe annotata con @Controller deve restituire il risultato
 * del metodo che abbiamo all'interno della classe annotata con @Service*/
@org.springframework.stereotype.Controller
@RequestMapping("/")
public class Controller {

    @Autowired
    Service service;

    @RequestMapping("/animale")
    public Animale getAnimale() {

        Animale animale = service.getAnimaleService();
        System.out.println("\nClassificazione = " + animale.getClassificazione());
        System.out.println("Razza = " + animale.getRazza() + "\n");

        return service.getAnimaleService();
    }
}
```

Il risultato dell'esempio è il seguente:


```
Classificazione = Mammifero
Razza = Cane
```

SPIEGAZIONE ESEMPIO PRECEDENTE MVC TRAMITE @CONTROLLER, @SERVICE E @REPOSITORY



Tramite client (per esempio un browser) effettuiamo una request, che viene intercettata da un Controller, che nel caso di Spring è annotata con:

1. l'annotation @Controller, da inserire sulla dichiarazione della classe ;
2. l'annotation @RequestMapping, da inserire sulla dichiarazione della classe e sulla dichiarazione di ogni metodo all'interno della classe stessa, e che ha il compito di mappare (definire) la struttura della request tramite un path (percorso). L'annotation @RequestMapping sulla classe sarà valorizzato con il path radice, mentre tutte gli altri @RequestMapping sui metodi verranno valorizzati ognuno con il proprio path (che si aggiunge al path radice), per chiamare quel metodo specifico.

Una volta intercettata la request, il nostro Controller chiama il metodo del @Service attraverso l'@Autowired.

A sua volta il @Service chiama il metodo del @Repository, sempre tramite @Autowired. Da notare che nell'esempio non c'è logica all'interno del @Service, perché vuole essere solo una rappresentazione del concetto di MVC.

Una volta che ci troviamo nel @Repository, siamo nella parte in cui vengono effettuate le chiamate al database. Anche qui da notare il fatto che nell'esempio viene solo simulata una chiamata al database, creando manualmente un oggetto di tipo Animale, sempre per il fatto che questo esempio vuole essere solo una rappresentazione del concetto di MVC.

Una volta effettuata la chiamata al database, viene restituito un oggetto dal @Repository al @Service. Tale oggetto ripercorre tutto il percorso inverso, fino a che non viene restituito al client come risposta alla request effettuata.

Ultimo appunto da fare è che il metodo del @Controller nell'esempio non ha parametri in ingresso, quindi stiamo inviando una request vuota. Può capitare però che da client si possano inserire dei dati nella request e, in questo caso, tali dati saranno i parametri in ingresso del metodo richiamato nel @Controller.

ANNOTATION @RESTCONTROLLER PER ESPORRE SERVIZI REST

Una volta creato il giro dell'MVC, come nell'esempio precedente, vediamo come modificare il nostro Controller per far si che esponga dei microservizi REST.

Partiamo con il modificare l'annotation @Controller con @RestController sopra la dichiarazione della classe. Adesso il nostro Controller è impostato per esporre servizi REST, per cui quando il client farà una request, tale request utilizzerà quelli che si chiamano verbi.

I verbi sono i servizi che permettono di comunicare con il server, dicendogli cosa deve fare a seconda del tipo di request che stiamo inoltrando. I verbi sono:

- GET, per recuperare dati;
- POST, per inserire dati;
- PUT, per aggiornare i dati;
- DELETE, per cancellare i dati;

Quando nel @RestController si creano dei metodi, avranno delle annotation che identificano che tipo di verbo stiamo utilizzando. Per questo motivo, è possibile anche utilizzare lo stesso path per tutti i metodi del @RestController (purché si cambi il verbo) e il server sarà comunque in grado di capire cosa deve fare.

I metodi del @RestController che hanno parametri in ingresso devono avere l'annotation @RequestBody prima della dichiarazione dei parametri all'interno delle parentesi tonde del metodo.

Vediamo un esempio:

Animale2.java

```
package com.example.Corso_Spring_Hibernate.REST.entity;

public class Animale2 {
    String classificazione;
    String razza;

    public Animale2() {
    }

    public String getClassificazione() {
        return classificazione;
    }

    public void setClassificazione(String classificazione) {
        this.classificazione = classificazione;
    }

    public String getRazza() {
        return razza;
    }

    public void setRazza(String razza) {
        this.razza = razza;
    }
}
```

Repository2.java

```java
package com.example.Corso_Spring_Hibernate.REST;

import com.example.Corso_Spring_Hibernate.REST.entity.Animale2;

import java.util.ArrayList;
import java.util.List;

@org.springframework.stereotype.Repository
public class Repository2 {

    public List<Animale2> getListAnimali(){
        List<Animale2> listaAnimali= new ArrayList<>();
        Animale2 animale = new Animale2();
        animale.setClassificazione("Mammifero");
        animale.setRazza("Cane");
        listaAnimali.add(animale);

        return listaAnimali;
    }

    public Animale2 inserisciAnimale(Animale2 animale) {
        return animale;
    }

    public Animale2 modificaAnimale(Animale2 animale) {
        return animale;
    }

    public Animale2 cancellaAnimale(Animale2 animale) {
        return animale;
    }
}
```

Service2.java

```java
package com.example.Corso_Spring_Hibernate.REST;

import com.example.Corso_Spring_Hibernate.REST.entity.Animale2;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.GetMapping;

import java.util.List;

@org.springframework.stereotype.Service
public class Service2 {

    @Autowired
    Repository2 repository;

    public List<Animale2> getListAnimali() {
        return repository.getListAnimali();
    }

    public Animale2 inserisciAnimale(Animale2 animale) {
        return repository.inserisciAnimale(animale);
```

```
    }

    public Animale2 modificaAnimale(Animale2 animale) {
        return repository.modificaAnimale(animale);
    }

    public Animale2 cancellaAnimale(Animale2 animale) {
        return repository.cancellaAnimale(animale);
    }
}
```

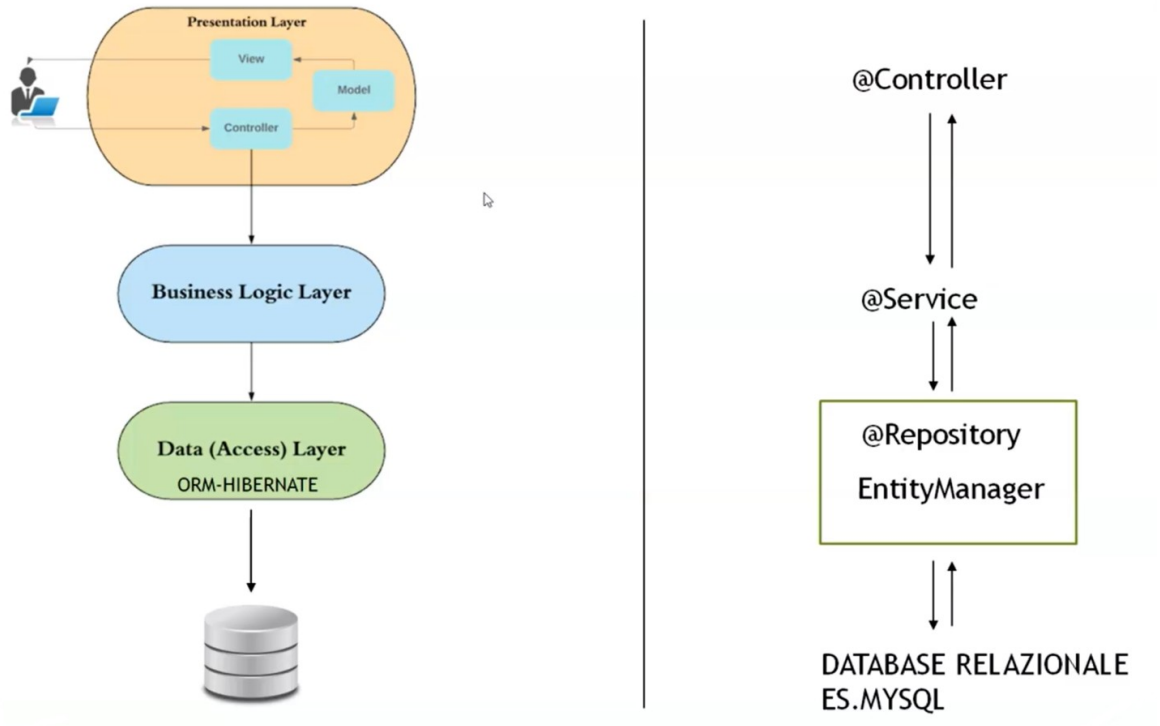
RestController.java

```
package com.example.Corso_Spring_Hibernate.REST;

import com.example.Corso_Spring_Hibernate.REST.entity.Animale2;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@org.springframework.web.bind.annotation.RestController
@RequestMapping("/")
public class RestController {

    @Autowired
    Service2 service;

    @GetMapping("/animali")
    public List<Animale2> getListAnimale() {
        return service.getListAnimali();
    }

    @PostMapping("/animali")
    public Animale2 inserisciAnimale(@RequestBody Animale2 animale) {
        return service.inserisciAnimale(animale);
    }

    @PutMapping("/animali")
    public Animale2 modificaAnimale(@RequestBody Animale2 animale) {
        return service.modificaAnimale(animale);
    }

    @DeleteMapping("/animali")
    public Animale2 cancellaAnimale(@RequestBody Animale2 animale) {
        return service.cancellaAnimale(animale);
    }
}
```

Ovviamente ogni metodo del @RestController nell'esempio restituirà un oggetto di tipo Animale2, senza effettuare alcuna operazione sul database, in quanto non siamo ancora collegati ad alcun database e non sono state implementate query. Questo esempio è solo un modello da seguire per poter definire un @RestController, seguendo il pattern MVC.

DATABASE

Finora abbiamo visto il funzionamento del container di Spring e un giro dell'MVC, fino però al Data Access Layer (che corrisponde al @Repository), senza integrarlo con Hibernate.

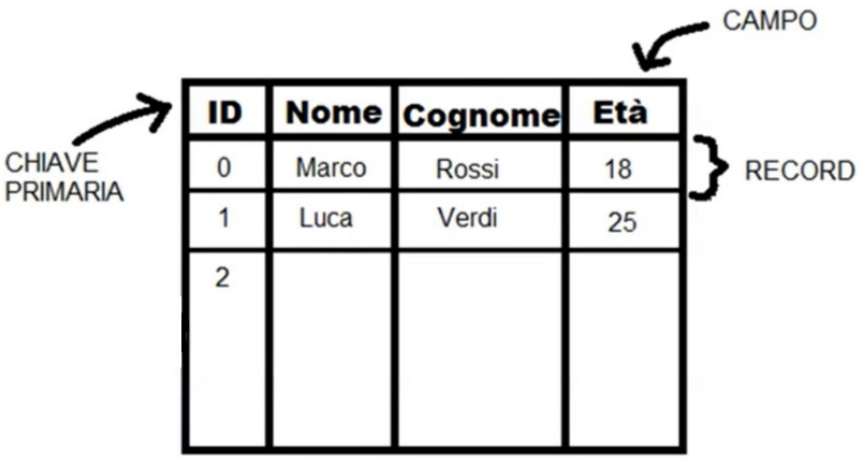


Per integrare il Data Access Layer con Hibernate, dobbiamo scaricare l'opportuno modulo.

Hibernate è una tecnologia che interagisce con il database (in particolare utilizzeremo il database relazionale Mysql).

DATABASE RELAZIONALI

I database relazionali sono una tecnologia di memorizzazione che ha avuto un grande successo a partire dagli anni 70. Essi contengono dei dati disposti in tabelle:



ID	Nome	Cognome	Età
0	Marco	Rossi	18
1	Luca	Verdi	25
2			

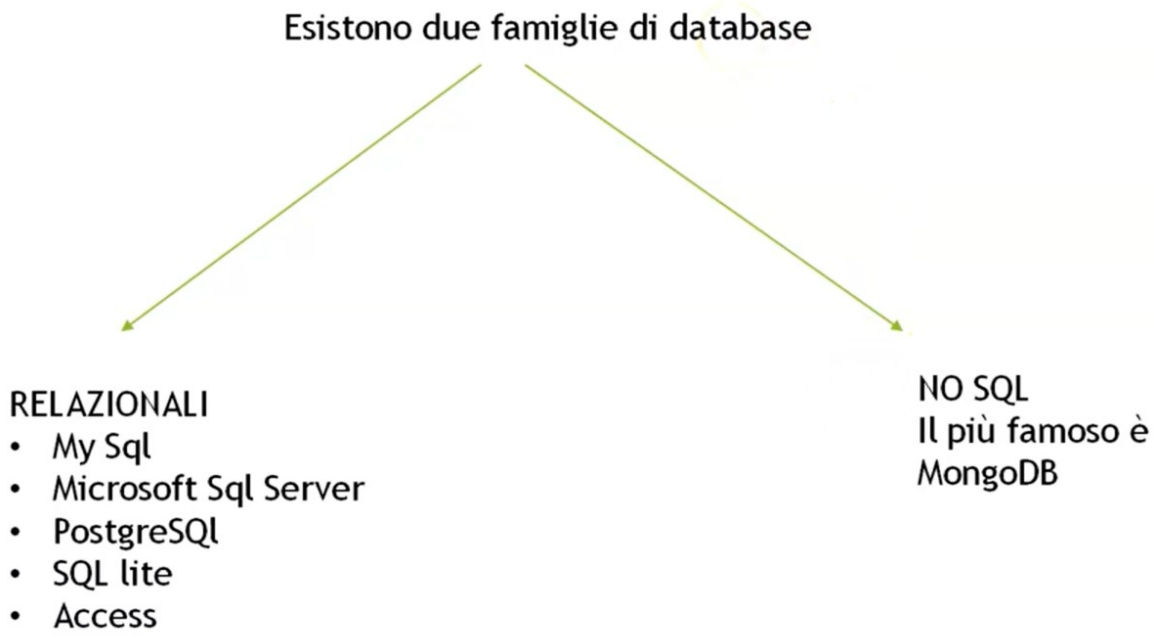
Quando osserviamo una tabella, dobbiamo familiarizzare con la terminologia e dobbiamo sapere che:

- un record è un insieme di campi;
- un campo è una singola cella della tabella;
- una tabella è un insieme di record;
- le tabelle sono relazionate tra loro grazie ad un riferimento alla chiave primaria, che è il campo della tabella contenente un codice univoco. Le relazioni tra tabelle possono essere di tipo:
  - OneToOne (uno ad uno);
  - ManyToOne (molti ad uno);
  - OneToMany (uno a molti);
  - ManyToMany (molti a molti).

Il principale linguaggio dei database relazionali è SQL, con il quale inviamo dei comandi ad un database.

Le operazioni fondamentali da svolgere prendono il nome di CRUD (Create Read Update Delete).

Quando ci si riferisce ad un database, si usa spesso l'acronimo DBMS (DataBase Management System), ma se ci riferisce ad un database relazionale, si utilizza il nome di RDBMS (Relational DataBase Management System).



Il JDBC è un API (libreria), definita da Sun Microsystem e composta da un insieme di classi ed interfacce scritte in JAVA, che ci consente di inviare comandi al database ed ottenere risultati.

Ogni DBMS avrà un proprio JDBC scaricabile dal sito del fornitore.

PROGETTO SPRING A MICROSERVIZI INTEGRATO CON HIBERNATE

TEORIA HIBERNATE E MICROSERVIZI

I microservizi sono un approccio architetturale alla realizzazione di applicazioni.

La differenza principale tra i microservizi e gli approcci monolitici è che nei microservizi ogni funzione è un servizio che può essere completato ed implementato in modo indipendente. Per questo motivo, i microservizi sono molto manutenibili, perché si può fare manutenzione ad un microservizio senza interrompere il funzionamento degli altri. Ciò invece non accade nella creazione di applicazioni con l'approccio monolitico, ovvero dove le funzioni fanno tutte parte di un'unica applicazione.

Hibernate è un framework di JAVA basato sull'ORM e utilizza il JDBC per comunicare con il database.

L'ORM (Object Relation Mapping) è una tecnica di programmazione che consiste nel mappare tabelle e record di un database attraverso oggetti software. Quando si parla di ORM siamo nel Data Access Layer (in Spring siamo all'interno del @Repository).

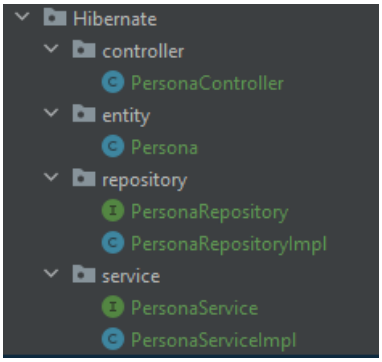
Quando mappiamo una tabella attraverso una classe JAVA, questa classe prende il nome di entità. Possiamo mappare una tabella attraverso una classe JAVA utilizzando le annotation o l'XML. Le annotation per fare ciò sono le seguenti:

- @Entity → Obbligatoria
- @Id → Obbligatoria
- @Table → Facoltativa
- @Column → Facoltativa
- @OneToOne → Facoltativa
- @ManyToOne → Facoltativa
- @OneToMany → Facoltativa
- @ManyToMany → Facoltativa
- @JoinColumn → Facoltativa
- @Lob → Facoltativa
- @Temporal → Facoltativa
- @Transient → Facoltativa
- @Transactional → Facoltativo

Per effettuare le operazioni di CRUD (Create Read Update Delete) si utilizza un'istanza dell'EntityManager.

L'EntityManager è un'interfaccia che ci permette di trasformare oggetti JAVA in record e viceversa. In pratica fa da ponte tra gli oggetti JAVA e i database.

PREDISPOSIZIONE DI UN PROGETTO SPRING A MICROSERVIZI DA INTEGRARE CON HIBERNATE



Persona.java

```java
package com.example.Corso_Spring_Hibernate.Hibernate.entity;

/**
 * Questa classe rappresenta la copia di una tabella su un database.
 * La sua struttura rappresenterà un record di tale tabella.
 * N.B.: Quando si lavora con oggetti che mappano tabelle, è
 * opportuno lavorare con le classi Wrapper, quindi non con i primitivi.
 */

public class Persona {
    Long id;
    String nome;
    String cognome;
    Integer eta;

    public Persona() {
    }

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getNome() {
        return nome;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }

    public String getCognome() {
        return cognome;
    }
```

```
    public void setCognome(String cognome) {
        this.cognome = cognome;
    }

    public Integer getEta() {
        return eta;
    }

    public void setEta(Integer eta) {
        this.eta = eta;
    }
}
```

PersonaRepository.java

```
package com.example.Corso_Spring_Hibernate.Hibernate.repository;

import com.example.Corso_Spring_Hibernate.Hibernate.entity.Persona;

import java.util.List;

public interface PersonaRepository {

    List<Persona> getPersonaList();

}
```

PersonaRepositoryImpl.java

```
package com.example.Corso_Spring_Hibernate.Hibernate.repository;

import com.example.Corso_Spring_Hibernate.Hibernate.entity.Persona;
import org.springframework.stereotype.Repository;

import java.util.List;

/**
 * Classe che provvede ad interrogare il database
 */

@Repository
public class PersonaRepositoryImpl implements PersonaRepository{
    @Override
    public List<Persona> getPersonaList() {
        return null;
    }
}
```

PersonaService.java

```java
package com.example.Corso_Spring_Hibernate.Hibernate.service;

import com.example.Corso_Spring_Hibernate.Hibernate.entity.Persona;
import java.util.List;

public interface PersonaService {

    List<Persona> getPersonaList();
}
```

PersonaServiceImpl.java

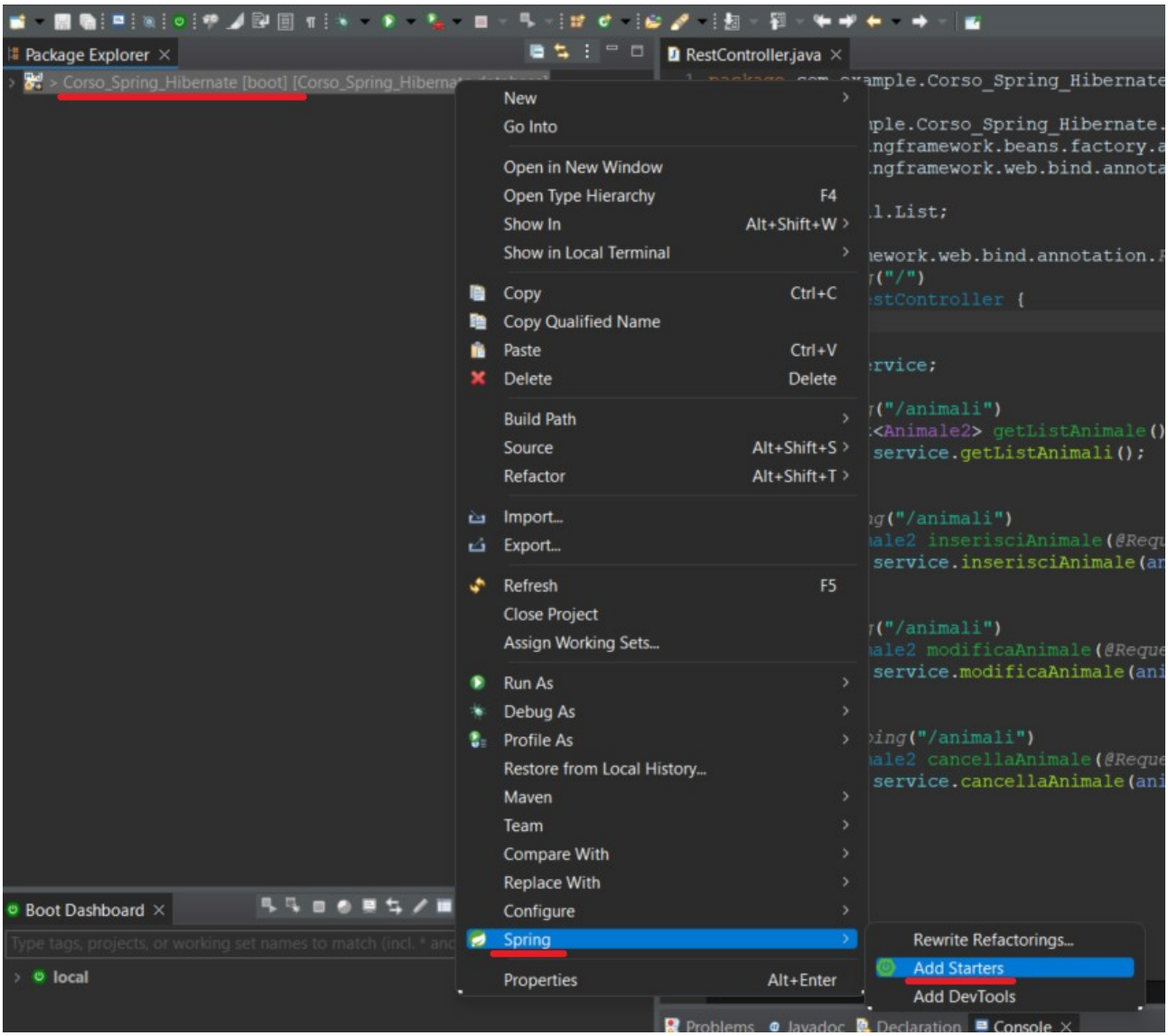
```java
package com.example.Corso_Spring_Hibernate.Hibernate.service;

import com.example.Corso_Spring_Hibernate.Hibernate.entity.Persona;
import com.example.Corso_Spring_Hibernate.Hibernate.repository.PersonaRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import java.util.List;

@Service
public class PersonaServiceImpl implements PersonaService{

    @Autowired
    PersonaRepository personaRepository;

    @Override
    public List<Persona> getPersonaList() {
        return personaRepository.getPersonaList();
    }
}
```

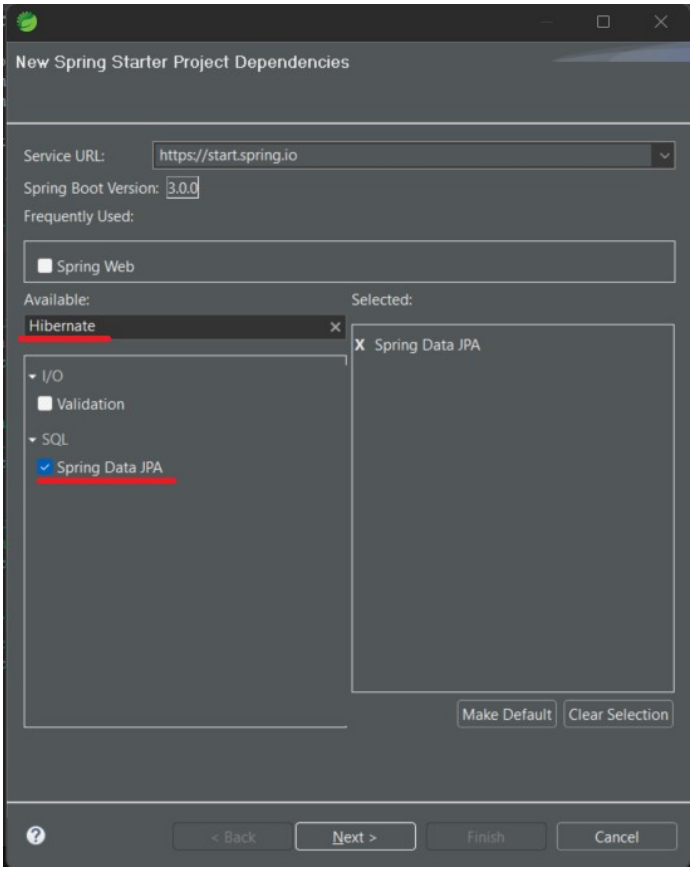
PersonaController.java

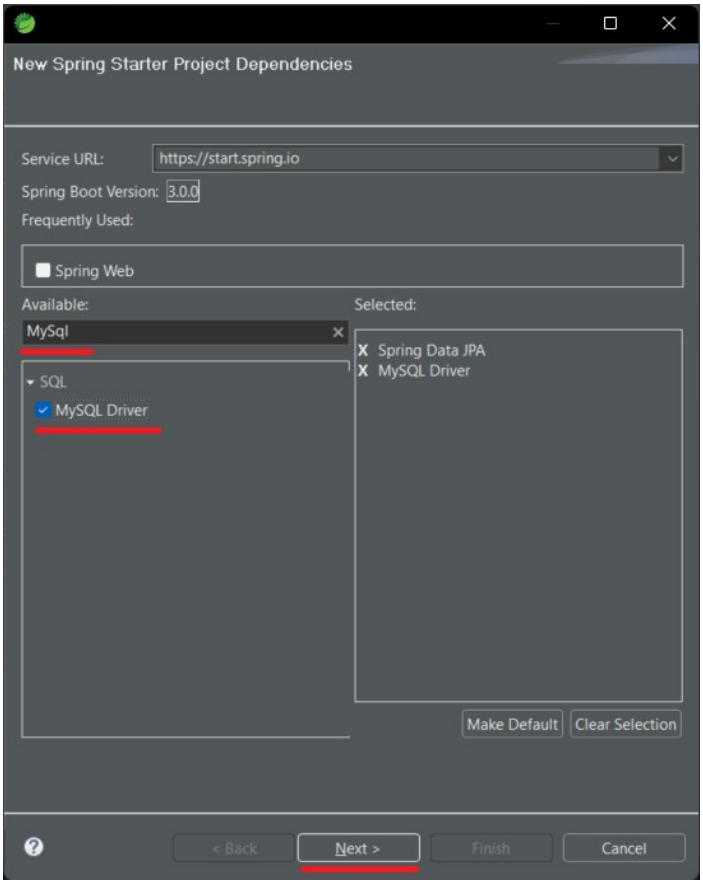
```java
package com.example.Corso_Spring_Hibernate.Hibernate.controller;

import com.example.Corso_Spring_Hibernate.Hibernate.entity.Persona;
import com.example.Corso_Spring_Hibernate.Hibernate.service.PersonaService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
import java.util.List;

/**
 * Classe che espone i servizi REST
 */
@RestController
@RequestMapping("/persona")
public class PersonaController {

    @Autowired
    PersonaService personaService;

    @GetMapping
    public List<Persona> getPersonaList(){
        return personaService.getPersonaList();
    }
}
```

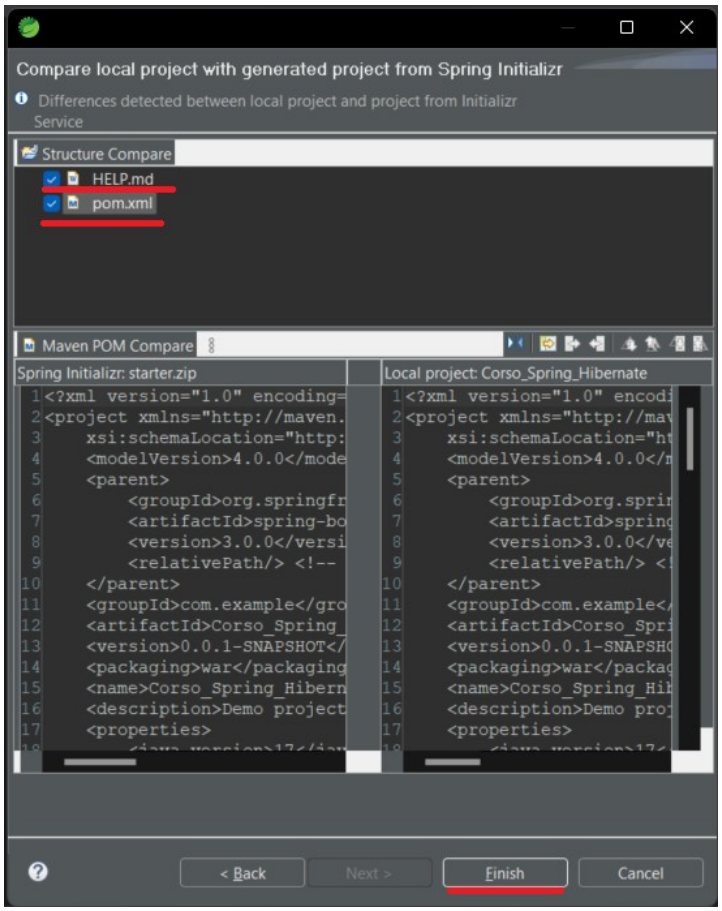
MAPPARE CLASSE SU TABELLA DI UN DATABASE

Prima di procedere con il mappare una classe su una tabella del database, bisogna verificare che sia presente nel progetto Spring il modulo che integra Hibernate. Se non ci dovesse essere, si può integrare anche nel progetto esistente tramite Spring Tool Suite nel seguente modo:




New Spring Starter Project Dependencies
Service URL:
https://start.spring.io
Spring Boot Version:
3.0.0
Frequently Used:
Spring Web
Available:
Hibernate
I/O
Validation
SQL
Spring Data JPA
Selected:
X Spring Data JPA
Make Default
Clear Selection
< Back
Next >
Finish
Cancel


New Spring Starter Project Dependencies
Service URL:
https://start.spring.io
Spring Boot Version:
3.0.0
Frequently Used:
Spring Web
Available:
MySql
SQL
MySQL Driver
Selected:
X Spring Data JPA
X MySQL Driver
Make Default
Clear Selection
< Back
Next >
Finish
Cancel



Ora dobbiamo mappare la nostra classe Persona dell'esempio precedente e per fare ciò ci servono delle annotazioni. La prima annotazione, che è quella più importante ed è obbligatoria, è l'annotation @Entity che indica che quella specifica classe è collegata ad una tabella del database.

Un'altra annotazione molto importante (anch'essa obbligatoria) è l'annotation @Id, che sta ad identificare la nostra chiave primaria.

Vediamo un esempio:

Persona.java

```java
package com.example.Corso_Spring_Hibernate.Hibernate.entity;

import jakarta.persistence.Entity;
import jakarta.persistence.Id;

/**
 * Questa classe rappresenta la copia di una tabella su un database.
 * La sua struttura rappresenterà un record di tale tabella.
 * N.B.: Quando si lavora con oggetti che mappano tabelle, è
 * opportuno lavorare con le classi Wrapper, quindi non con i primitivi.
 */

@Entity
public class Persona {

    @Id
    Long id;
    String nome;
    String cognome;
    Integer eta;

    public Persona() {
    }

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getNome() {
        return nome;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }

    public String getCognome() {
        return cognome;
    }

    public void setCognome(String cognome) {
        this.cognome = cognome;
    }

    public Integer getEta() {
        return eta;
    }

    public void setEta(Integer eta) {
        this.eta = eta;
    }
}
```

CONFIGURAZIONE DATASOURCE TRAMITE PROPERTIES

Dopo aver creato le classi che rappresentano le tabelle del database, possiamo configurare il nostro datasource per collegarci al database. Per fare ciò, possiamo utilizzare il file application.properties già creato da Spring nel percorso src→main→resources.

Il file application.properties permette di configurare Spring mediante delle combinazioni chiave-valore. Vediamo un esempio di configurazione dell'application.properties per collegarci al database:

```
#Dati per connessione al database MySQL
############################################################
spring.datasource.username = root
spring.datasource.password = root

#nome database
spring.datasource.url = jdbc:mysql://localhost:3306/palestra

#comando per visualizzare query in console
spring.jpa.show-sql=true

#comando che fa lo scan di tutte le classi annotate con @Entity
#e ne crea la rispettiva tabella nel database, senza doverlo fare noi.
#N.B: Se però, dopo aver creato la tabella ed aver inserito i dati, non
#commentiamo questo comando, la tabella verrà creata sempre da 0 ad
#ogni restart dell'applicazione, e ciò comporterà la perdita di tutti
#i dati inseriti. Per cui, subito dopo la creazione della tabella,
#è consigliato cambiare il valore di tale comando da create a update.
#spring.jpa.hibernate.ddl-auto = create

spring.jpa.hibernate.ddl-auto = update
############################################################
```

CONFIGURAZIONE DATASOURCE TRAMITE CLASSE @CONFIGURATION

È possibile configurare il datasource per il collegamento al database, in maniera alternativa al file application.properties, anche tramite una classe annotata con @Configuration.

Vediamo come impostare la classe @Configuration tramite il seguente esempio:

```
package com.example.Corso_Spring_Hibernate.Hibernate.datasource;

import org.springframework.boot.jdbc.DataSourceBuilder;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import javax.sql.DataSource;

@Configuration
public class DataSourcePerMySQL {

    @Bean
    public DataSource getDataSource() {
        DataSourceBuilder dataSourceBuilder = DataSourceBuilder.create();
        dataSourceBuilder.driverClassName("com.mysql.cj.jdbc.Driver");
        dataSourceBuilder.url("jdbc:mysql://localhost:3306/palestra");
        dataSourceBuilder.username("root");
        dataSourceBuilder.password("root");
        return dataSourceBuilder.build();
    }
}
```

GET, POST, PUT E DELETE CON HIBERNATE

Ora che abbiamo la connessione al database, possiamo eseguire delle interrogazioni. Tali interrogazioni verranno fatte nel @Repository, dove appunto implementeremo tutte le query che ci serviranno.

Prima di implementare le query, però, è necessario avere alcuni accorgimenti:

- nel @Repository, creare un'istanza dell'EntityManager mediante l'annotation @PersistenceContext. In questo modo, una volta che inizializzeremo la nostra applicazione, Spring ha all'interno del suo container anche dei bean dell'EntityManager. Quindi Spring andrà a pescare l'EntityManager grazie all'annotation @PersistenceContext e noi lo utilizzeremo per interrogare il database;
- nel @Repository, utilizzare l'annotation @Transactional sui metodi che implementano query che effettuano inserimenti, modifiche e cancellazioni sulle tabelle, escludendo quindi le query di sola lettura. Questo perché, se stiamo inserendo, modificando o cancellando più record dalle tabelle nella stessa transazione, nel caso in cui qualcosa non vada a buon fine, l'annotation @Transactional effettua un rollback e riporta lo stato della tabella a come era in origine.
- Nell' @Entity, utilizzare l'annotation @GeneratedValue(strategy = GenerationType.AUTO) per far gestire direttamente a Spring l'inserimento della chiave primaria all'interno della tabella.

Vediamo di seguito un esempio di interrogazioni al database:

Persona.java

```
package com.example.Corso_Spring_Hibernate.Hibernate.entity;

import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;

/**
 * Questa classe rappresenta la copia di una tabella su un database.
 * La sua struttura rappresenterà un record di tale tabella.
 * N.B.: Quando si lavora con oggetti che mappano tabelle, è
 * opportuno lavorare con le classi Wrapper, quindi non con i primitivi.
 */

@Entity
public class Persona {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    Long id;
    String nome;
    String cognome;
    Integer eta;

    public Persona() {
    }

    public Long getId() {
        return id;
    }
```

```
    public void setId(Long id) {
        this.id = id;
    }

    public String getNome() {
        return nome;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }

    public String getCognome() {
        return cognome;
    }

    public void setCognome(String cognome) {
        this.cognome = cognome;
    }

    public Integer getEta() {
        return eta;
    }

    public void setEta(Integer eta) {
        this.eta = eta;
    }
}
```

PersonaRepository.java

```
package com.example.Corso_Spring_Hibernate.Hibernate.repository;

import com.example.Corso_Spring_Hibernate.Hibernate.entity.Persona;

import java.util.List;

public interface PersonaRepository {

    List<Persona> getPersonaList();
    Persona insertPersona(Persona persona);
    Persona updatePersona(Persona persona);
    Persona deletePersona(Persona persona);
    List<Persona> findPersonaByName(String nome);
}
```

PersonaRepositoryImpl.java

```java
package com.example.Corso_Spring_Hibernate.Hibernate.repository;

import com.example.Corso_Spring_Hibernate.Hibernate.entity.Persona;
import jakarta.persistence.EntityManager;
import jakarta.persistence.PersistenceContext;
import jakarta.persistence.Query;
import jakarta.transaction.Transactional;
import org.springframework.stereotype.Repository;

import java.util.List;

/**
 * Classe che provvede ad interrogare il database
 */

@Repository
public class PersonaRepositoryImpl implements PersonaRepository{

    @PersistenceContext
    EntityManager entityManager;

    @Override
    public List<Persona> getPersonaList() {
        Query query = entityManager.createQuery("SELECT p FROM Persona p");
        return query.getResultList();
    }

    @Override
    @Transactional
    public Persona insertPersona(Persona persona) {
        entityManager.persist(persona);
        return persona;
    }

    @Override
    @Transactional
    public Persona updatePersona(Persona persona) {
        entityManager.merge(persona);
        return persona;
    }

    @Override
    @Transactional
    public Persona deletePersona(Persona persona) {
        Persona p = entityManager.find(Persona.class, persona.getId());
        entityManager.remove(p);
        return p;
    }

    @Override
    public List<Persona> findPersonaByName(String nome) {
        Query query = entityManager.createQuery("SELECT p FROM Persona p WHERE p.nome= :nome");
        return  query.setParameter("nome", nome). getResultList();
    }
}
```

PersonaService.java

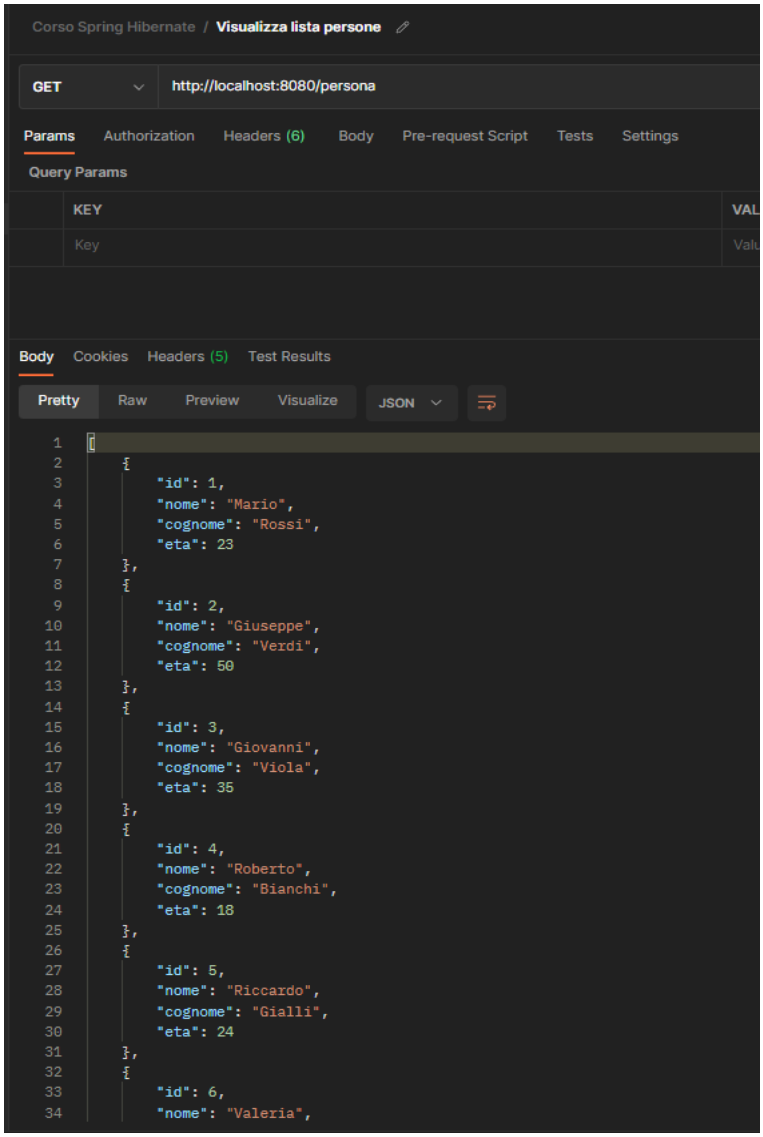
```java
package com.example.Corso_Spring_Hibernate.Hibernate.service;

import com.example.Corso_Spring_Hibernate.Hibernate.entity.Persona;

import java.util.List;

public interface PersonaService {

    List<Persona> getPersonaList();
    Persona insertPersona(Persona persona);
    Persona updatePersona(Persona persona);
    Persona deletePersona(Persona persona);
    List<Persona> findPersonaByName(String nome);
}
```

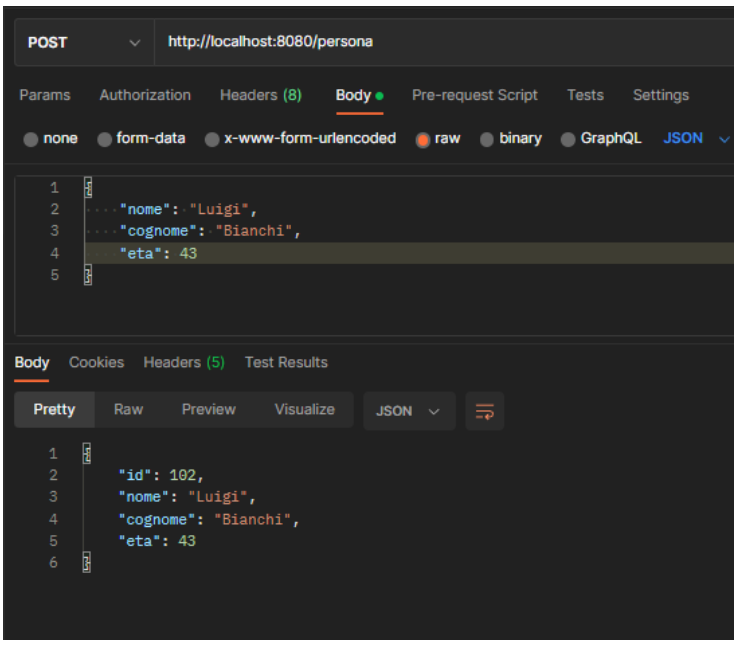
PersonaServiceImpl.java

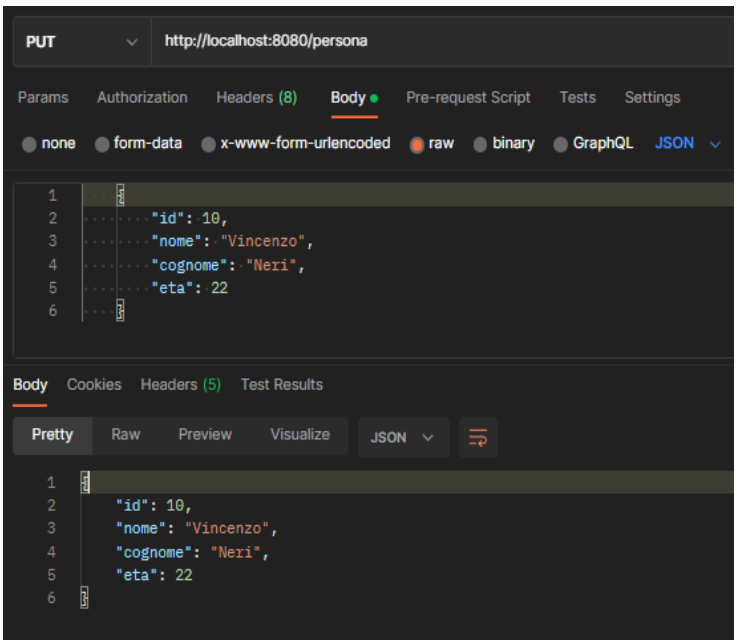
```java
package com.example.Corso_Spring_Hibernate.Hibernate.service;

import com.example.Corso_Spring_Hibernate.Hibernate.entity.Persona;
import com.example.Corso_Spring_Hibernate.Hibernate.repository.PersonaRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.util.List;

@Service
public class PersonaServiceImpl implements PersonaService{

    @Autowired
    PersonaRepository personaRepository;

    @Override
    public List<Persona> getPersonaList() {
        return personaRepository.getPersonaList();
    }

    @Override
    public Persona insertPersona(Persona persona) {
        return personaRepository.insertPersona(persona);
    }

    @Override
    public Persona updatePersona(Persona persona) {
        return personaRepository.updatePersona(persona);
    }

    @Override
    public Persona deletePersona(Persona persona) {
        return personaRepository.deletePersona(persona);
    }

    @Override
    public List<Persona> findPersonaByName(String nome) {
        return personaRepository.findPersonaByName(nome);
    }
}
```

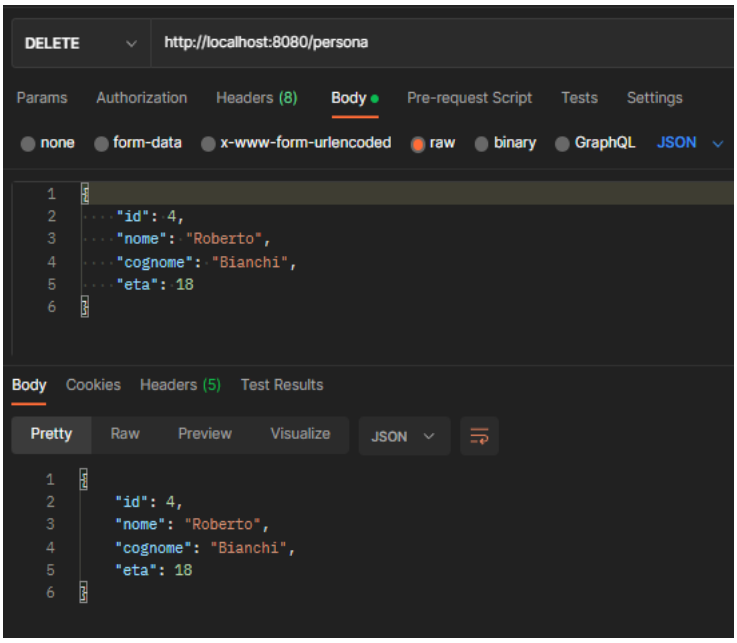
PersonaController.java

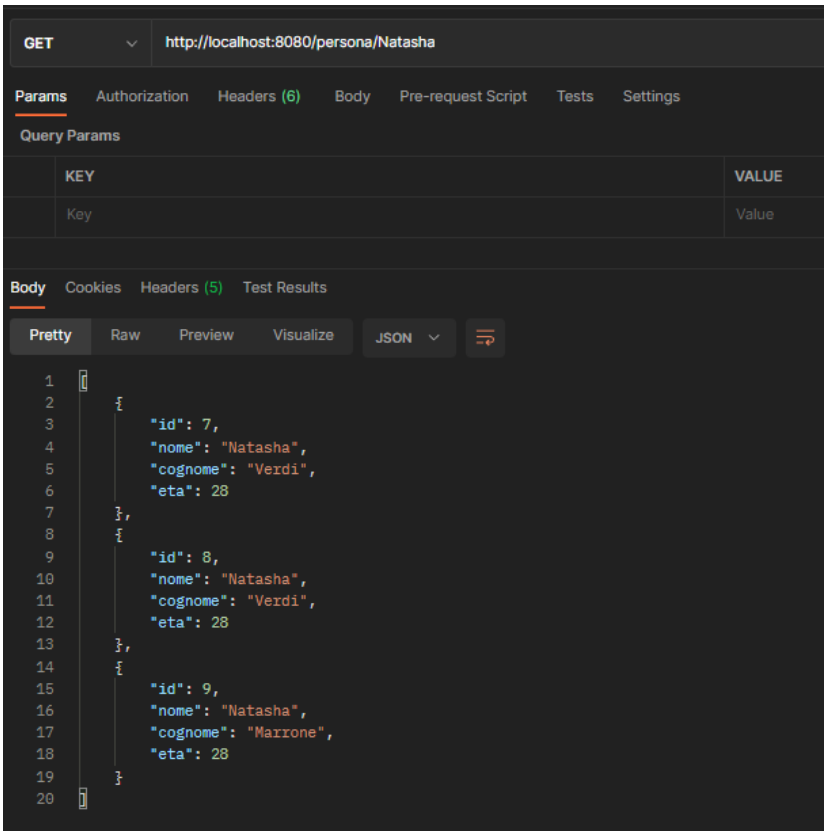
```java
package com.example.Corso_Spring_Hibernate.Hibernate.controller;

import com.example.Corso_Spring_Hibernate.Hibernate.entity.Persona;
import com.example.Corso_Spring_Hibernate.Hibernate.service.PersonaService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

import java.util.List;

/**
 * Classe che espone i servizi REST
 */

@RestController
@RequestMapping("/persona")
public class PersonaController {

    @Autowired
    PersonaService personaService;

    @GetMapping
    public ResponseEntity<List<Persona>> getPersonaList(){
        List<Persona> listaPersone = personaService.getPersonaList();
        return new ResponseEntity<List<Persona>>(listaPersone, HttpStatus.OK);
    }

    @PostMapping
    public ResponseEntity<Persona> insertPersona(@RequestBody Persona persona){
        Persona personaResult = personaService.insertPersona(persona);
        return new ResponseEntity<Persona>(personaResult, HttpStatus.CREATED);
    }

    @PutMapping
    public ResponseEntity<Persona> updatePersona(@RequestBody Persona persona){
        Persona personaResult = personaService.updatePersona(persona);
        return new ResponseEntity<Persona>(personaResult, HttpStatus.OK);
    }

    @DeleteMapping
    public ResponseEntity<Persona> deletePersona(@RequestBody Persona persona){
        Persona personaResult = personaService.deletePersona(persona);
        return new ResponseEntity<Persona>(personaResult, HttpStatus.OK);
    }

    @GetMapping("/{nome}")
    public ResponseEntity<List<Persona>> getPersonaByName(@PathVariable String nome){
        List<Persona> listaPersone = personaService.findPersonaByName(nome);
        return new ResponseEntity<List<Persona>>(listaPersone, HttpStatus.OK);
    }
}
```


Corso Spring Hibernate / Visualizza lista persone
GET
http://localhost:8080/persona
Params
Authorization
Headers (6)
Body
Pre-request Script
Tests
Settings
Query Params
KEY
Key
Body
Cookies
Headers (5)
Test Results
Pretty
Raw
Preview
Visualize
JSON
[
{
"id": 1,
"nome": "Mario",
"cognome": "Rossi",
"eta": 23
},
{
"id": 2,
"nome": "Giuseppe",
"cognome": "Verdi",
"eta": 50
},
{
"id": 3,
"nome": "Giovanni",
"cognome": "Viola",
"eta": 35
},
{
"id": 4,
"nome": "Roberto",
"cognome": "Bianchi",
"eta": 18
},
{
"id": 5,
"nome": "Riccardo",
"cognome": "Gialli",
"eta": 24
},
{
"id": 6,
"nome": "Valeria",


POST
http://localhost:8080/persona
Params
Authorization
Headers (8)
Body
Pre-request Script
Tests
Settings
none
form-data
x-www-form-urlencoded
raw
binary
GraphQL
JSON
{
"nome": "Luigi",
"cognome": "Bianchi",
"eta": 43
}
Body
Cookies
Headers (5)
Test Results
Pretty
Raw
Preview
Visualize
JSON
{
"id": 102,
"nome": "Luigi",
"cognome": "Bianchi",
"eta": 43
}


PUT
http://localhost:8080/persona
Params
Authorization
Headers (8)
Body
Pre-request Script
Tests
Settings
none
form-data
x-www-form-urlencoded
raw
binary
GraphQL
JSON
{
"id": 10,
"nome": "Vincenzo",
"cognome": "Neri",
"eta": 22
}
Body
Cookies
Headers (5)
Test Results
Pretty
Raw
Preview
Visualize
JSON
{
"id": 10,
"nome": "Vincenzo",
"cognome": "Neri",
"eta": 22
}

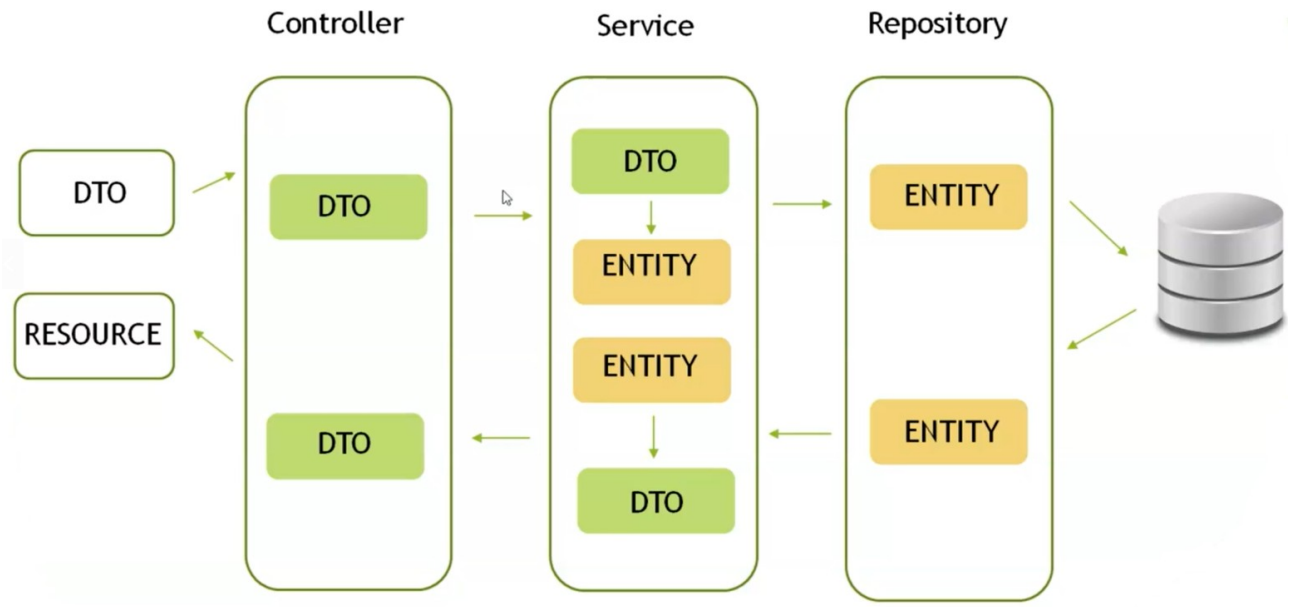

DELETE
http://localhost:8080/persona
Params
Authorization
Headers (8)
Body
Pre-request Script
Tests
Settings
none
form-data
x-www-form-urlencoded
raw
binary
GraphQL
JSON
{
"id": 4,
"nome": "Roberto",
"cognome": "Bianchi",
"eta": 18
}
Body
Cookies
Headers (5)
Test Results
Pretty
Raw
Preview
Visualize
JSON
{
"id": 4,
"nome": "Roberto",
"cognome": "Bianchi",
"eta": 18
}


GET
http://localhost:8080/persona/Natasha
Params
Authorization
Headers (6)
Body
Pre-request Script
Tests
Settings
Query Params
KEY
VALUE
Key
Value
Body
Cookies
Headers (5)
Test Results
Pretty
Raw
Preview
Visualize
JSON
[
{
"id": 7,
"nome": "Natasha",
"cognome": "Verdi",
"eta": 28
},
{
"id": 8,
"nome": "Natasha",
"cognome": "Verdi",
"eta": 28
},
{
"id": 9,
"nome": "Natasha",
"cognome": "Marrone",
"eta": 28
}
]

DIFFERENZA TRA DTO E ENTITY

L'applicazione Spring a microservizi con Hibernate integrato che abbiamo creato finora presenta solo Entity, per cui non abbiamo ancora inserito il concetto di DTO e la conversione da Entity a DTO. Questo significa che quando andiamo a richiamare ciò che abbiamo nel database (nel nostro caso Persona) attraverso una request, otteniamo all'interno della response tutto l'oggetto.

Ci sono casi in cui, però, che non è opportuno restituire tutto l'oggetto, ma solo una parte di esso, ed è per questo che entrano in gioco i DTO.



Per esempio, immaginiamo la classe Persona con i campi nome, cognome, età e data di creazione. La data di creazione ci serve per tener traccia di quando quell'oggetto viene inserito all'interno del database.

Quando, però, facciamo una GET, vogliamo il nome, cognome ed età della persona, ma poco ci interessa della data di inserimento dell'oggetto all'interno del database.

Per capire meglio il concetto di DTO, possiamo guardare i seguenti step:

- il client manda una request sotto forma di DTO al Controller;
- il DTO passa dal Controller al Service e viene convertito in Entity;
- l'Entity passa dal Service al Repository. Il Repository effettuerà le dovute operazioni sul database tramite l'Entity che gli è stato passato;
- Al Repository verrà ritornato l'Entity dal database;
- l'Entity passa dal Repository al Service e viene convertito in DTO. In questo modo il Service filtra i campi, facendo vedere all'utente finale solo le informazioni che gli servono;
- il DTO passa dal Repository al Controller e la risorsa viene finalmente esposta.