

APPUNTI JAVA SPRING E HIBERNATE

Indice generale

COSA SONO SPRING E SPRING BOOT.....	3
DESIGN PATTERN.....	4
SPRING BOOT.....	5
CONFIGURAZIONE DI SPRING.....	5
IOC (INVERSION OF CONTROL) E DI (DEPENDENCY INJECTION).....	6
ALTRO ESEMPIO DI DEPENDENCY INJECTION.....	9
ANNOTATION @QUALIFIER.....	11
ANNOTATION @COMPONENTSCAN.....	14
ANNOTATION @SCOPE.....	16
ANNOTATION @PATHVARIABLE.....	18
ANNOTATION @REQUESTPARAM.....	19
ANNOTATION @CONTROLLER, @SERVICE, @REPOSITORY (MVC).....	20
SPIEGAZIONE ESEMPIO PRECEDENTE MVC TRAMITE @CONTROLLER, @SERVICE E @REPOSITORY....	23
ANNOTATION @RESTCONTROLLER PER ESPORRE SERVIZI REST.....	24

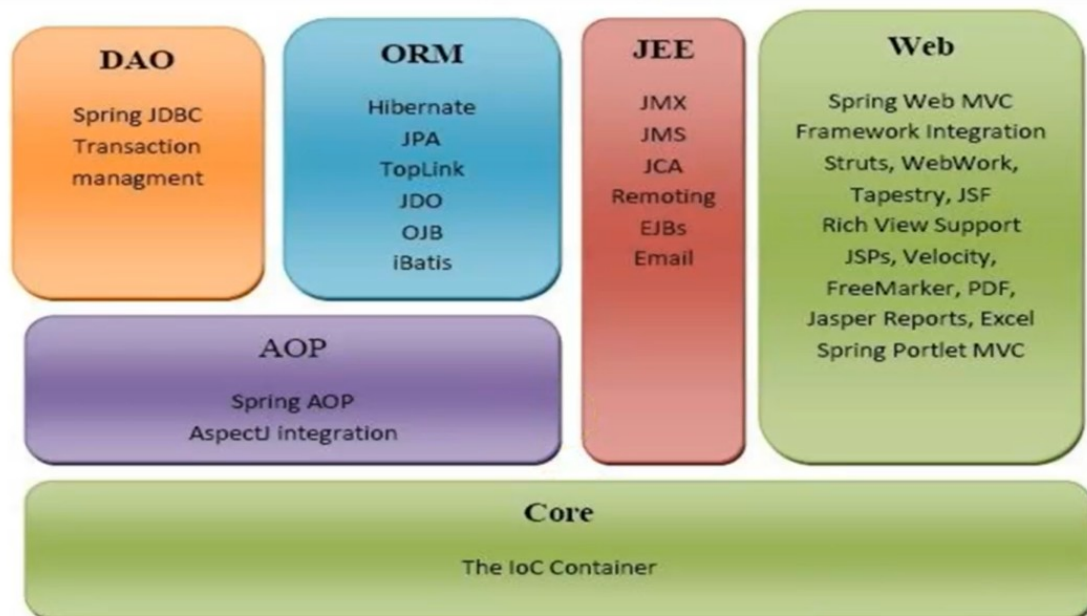
COSA SONO SPRING E SPRING BOOT

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:

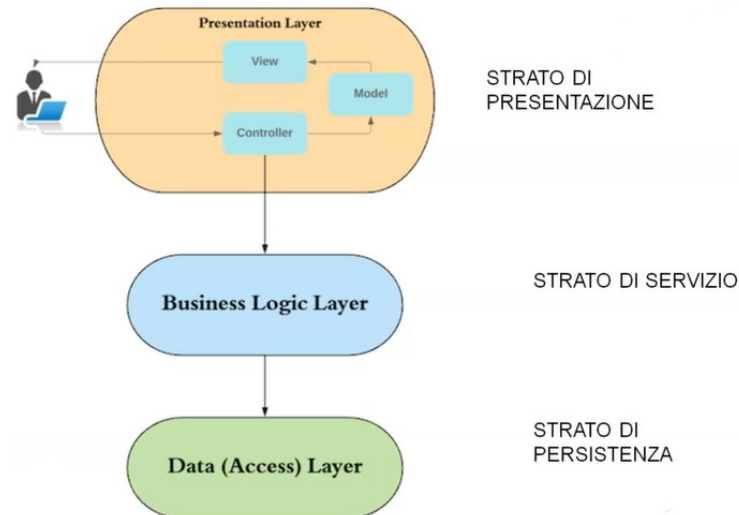


Il modulo obbligatorio di cui parlavamo è il Core, che è il motore del framework. Gli altri moduli possono essere utilizzati facoltativamente.

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, dopodiché 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
- @Size
- @Valid
- @EnableAutoConfiguration
- @SpringBootApplication

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

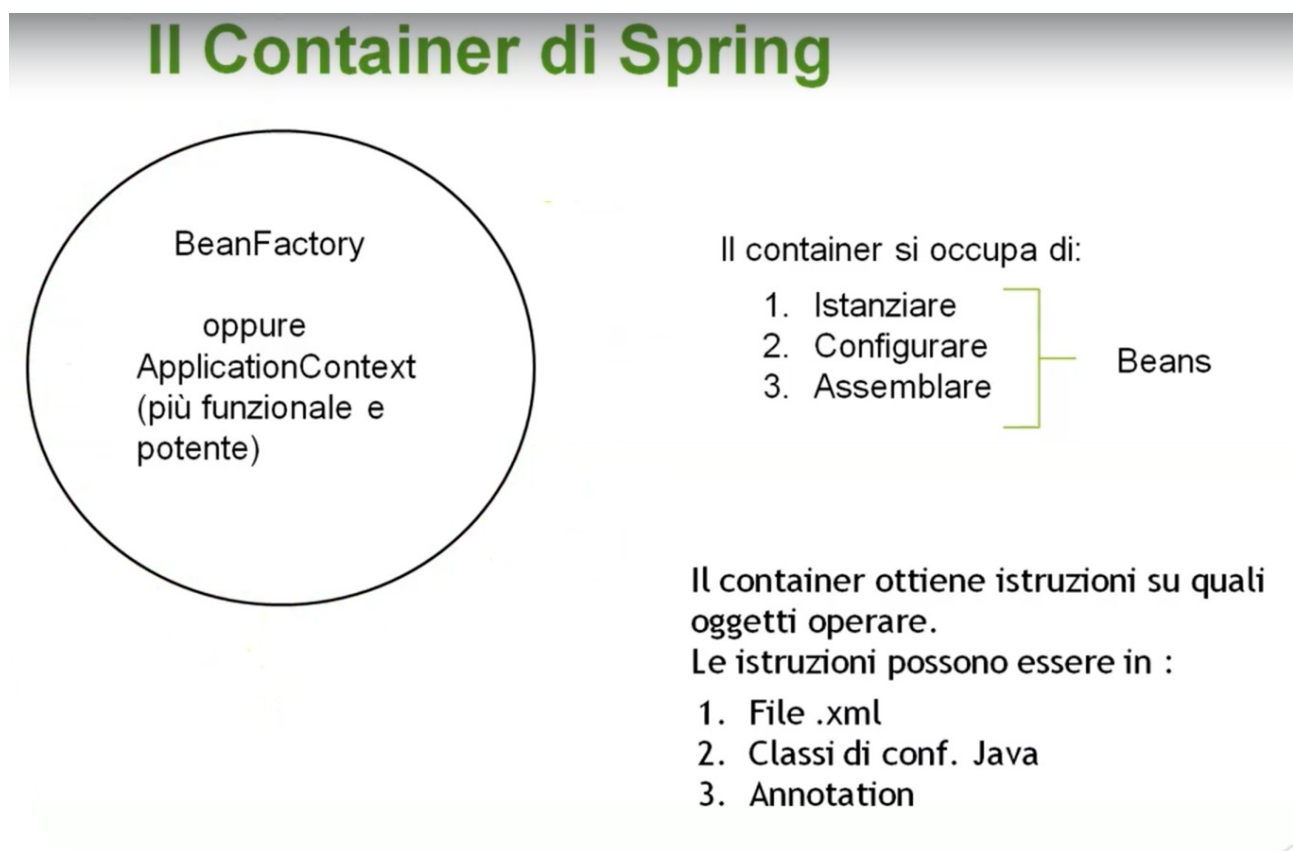
I due concetti che andremo a vedere di seguito sono fondamentali per il funzionamento di Spring.

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.

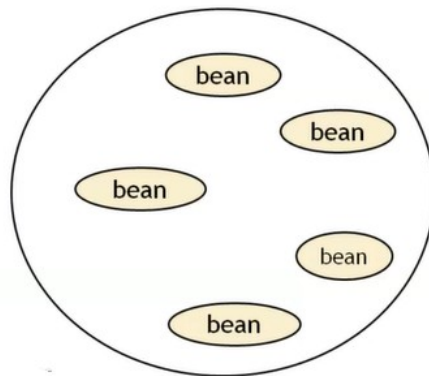
La stessa operazione può essere fatta mettendo l'annotation `@Bean` sopra alcuni metodi che si trovano nelle classi di configurazione, opportunamente annotate con `@Configuration`.

Sulle classi:

`@Component` `@Controller(@RestController)` `@Service` `@Repository`

Oppure sui metodi:

`@Bean` (su metodi all'interno di classi di configurazione, opportunamente annotate con `@Configuration`)



Dove vogliamo richiamare l'istanza:
`@Autowired`

Vediamo dal punto di vista pratico questo concetto:

Persona.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
 * container
 */
@Component
public class Persona {
    String nome;
    String cognome;

    @Autowired
    Indirizzo indirizzo;
}
```

Indirizzo.java

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

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

MainDependencyInjection.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

```
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());
    }
}
```



```
2022-11-30T18:57:51.771+01:00 INFO 7896 --- [nio-
2022-11-30T18:57:51.771+01:00 INFO 7896 --- [nio-
2022-11-30T18:57:51.772+01:00 INFO 7896 --- [nio-
nome marca = Citroen
nome modello = C1
2022-11-30T18:57:51.815+01:00 ERROR 7896 --- [nio-

jakarta.servlet.ServletException Create breakpoint : Ci
at org.springframework.web.servlet.view.Intern
at org.springframework.web.servlet.view.Intern
at org.springframework.web.servlet.view.Abstra
```

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 dà 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

```
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

```
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;
}
```

```

/*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("\nome 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");
}
}

```



Whitelabel Error Page

This application has no explicit mapping for /error, so you are seeing this as a fallback.

Thu Dec 01 00:04:24 GMT+01:00 2022

There was an unexpected error (type=Internal Server Error, status=500).

```

2022-12-01T00:04:16.432+01:00 INFO 18812 --- [
nome marca veicolo 1 = Citroen
nome modello veicolo 1 = C1

nome marca veicolo 2 = Fiat
nome modello veicolo 2 = Panda

2022-12-01T00:04:24.587+01:00 ERROR 18812 --- [

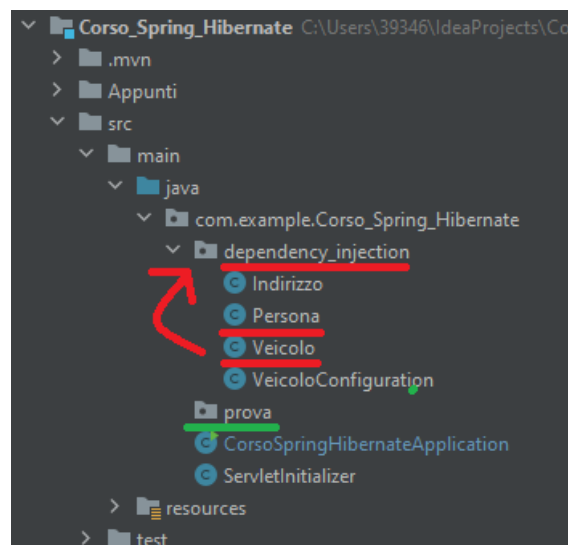
```

ANNOTATION @COMPONENTSCAN

Quando eseguiamo un'applicazione creata con 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 conterrà 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;

/* @Scope("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 container*/
@Component
public class Persona {
    public String nome;
    public String cognome;
    public static int contatore = 0;

    @Autowired
    Indirizzo indirizzo;

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


ScopeAnnotation.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"):

```
2022-12-05T13:28:20.394+01:00 INFO 189
2022-12-05T13:28:20.395+01:00 INFO 189
numero istanze di Persona : 3
```

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

```
2022-12-05T13:32:14.025+01:00
numero istanze di Persona : 1
```

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 {

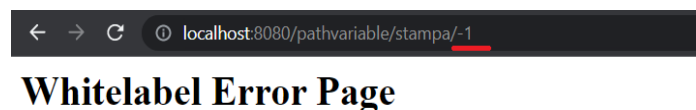
    @RequestMapping("/stampa/{variabile}")
    public void stampa(@PathVariable int variabile){
        if(variabile < 0){
            System.out.println("il valore della variabile è negativo");
        } else {
            System.out.println("il valore della variabile è positivo");
        }
    }
}
```

Il risultato di questo esempio è il seguente:



This application has no explicit mapping for /error, so you are seeing this as a fallback.
Mon Dec 05 14:43:54 GMT+01:00 2022
There was an unexpected error (type=Not Found, status=404).

```
2022-12-05T14:43:54.881+01:00 INFO 2
il valore della variabile è positivo
```



This application has no explicit mapping for /error, so you are seeing this as a fallback.
Mon Dec 05 14:44:06 GMT+01:00 2022
There was an unexpected error (type=Not Found, status=404).

```
2022-12-05T14:43:54.881+01:00 INFO 22732
il valore della variabile è positivo
il valore della variabile è negativo
```

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");
        }
    }
}
```

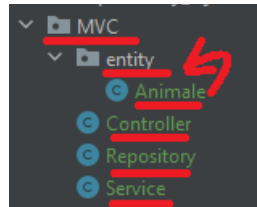
Il risultato di questo esempio è il seguente:



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

```
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

```
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

```
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 {

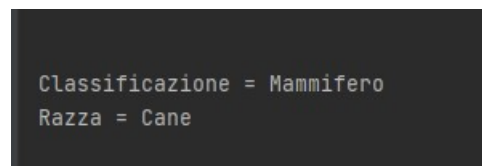
    @Autowired
    Service service;

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

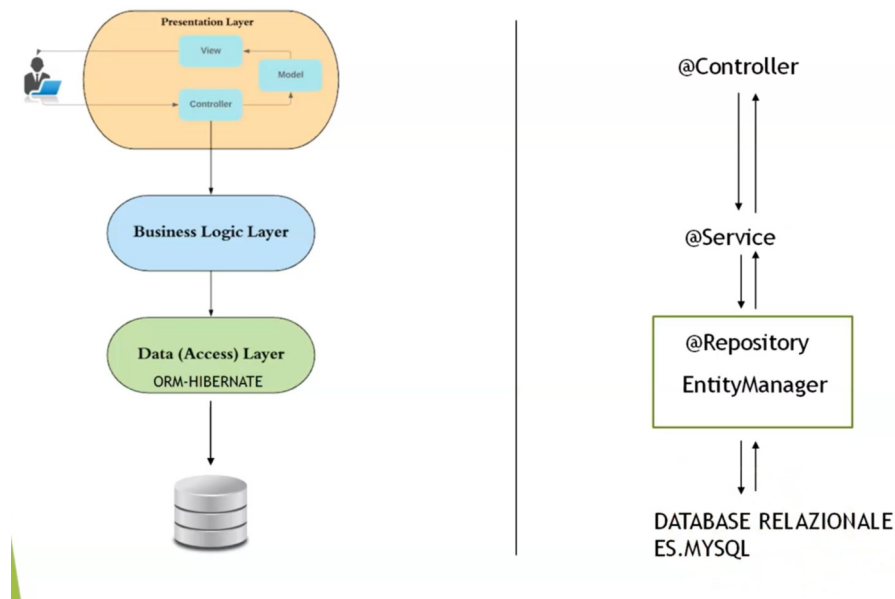
        Animale animale = service.getAnimaleService();
        System.out.println("\nClassificazione = " + animale.getClassificazione());
        System.out.println("Razza = " + animale.getRazza() + "\n");

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

Il risultato dell'esempio è il seguente:



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 sì 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

```
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

```
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.getListAnimale();
    }

    @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.