***Introduzione a Spring***

*Spring è un framework modulare che semplifica la creazione di applicazioni Java scalabili e manutenibili. La sua filosofia ruota attorno al* ***risolvere problemi comuni nello sviluppo software****, come la riduzione della complessità del codice e l'eliminazione di accoppiamenti rigidi tra le componenti.*

***Obiettivo***

*Spring offre soluzioni per:*

1. ***Gestione delle dipendenze*** *(Dependency Injection, DI).*
2. ***Decoupling*** *(riduzione delle dipendenze rigide tra le classi).*
3. ***Persistenza e gestione dei dati*** *con Spring Data.*
4. ***Creazione di applicazioni web*** *con Spring MVC.*

***Concetti Principali***

***Dependency Injection (DI)***

* *La DI si occupa di risolvere le dipendenze tra oggetti.*
* *Se un oggetto ha bisogno di un altro (dipendenza), è Spring che si occupa di* ***crearlo*** *e di* ***passarlo****, evitando di doverlo istanziare manualmente con new.*
* ***Esempio pratico****: invece di scrivere:*

*java*

*Copia codice*

*Service service = new Service();*

*Spring inizializza automaticamente l'oggetto:*

*java*

*Copia codice*

*@Autowired*

*private Service service;*

***Inversion of Control (IoC)***

* *Con IoC, è Spring a controllare il ciclo di vita degli oggetti, non il programmatore. Questo approccio decoupla le componenti e rende il codice più flessibile e testabile.*
* *Gli oggetti sono gestiti da un* ***contenitore IoC****, che li crea, li configura e li inietta dove servono.*

***Tecnologie Principali***

***1. Spring MVC***

* *Spring MVC è una tecnologia per creare applicazioni web basate sul pattern* ***Model-View-Controller****.*
* *I* ***controller*** *rispondono a richieste HTTP, elaborano i dati e restituiscono risposte sotto forma di:*
  + ***HTML*** *(tradizionale).*
  + ***JSON*** *(moderno, per API REST).*
* *Configurazione di base:*
  + *Tramite un file XML (vecchio stile) o annotazioni (@Controller, @RestController).*
  + *I controller "ibridi" possono generare risposte diverse (HTML o JSON) a seconda delle esigenze.*

***Configurazione (Web.xml)***

* *Spring MVC usa una servlet chiamata DispatcherServlet come punto di ingresso per tutte le richieste.*
* ***Esempio di configurazione:***

*xml*

*Copia codice*

*<servlet>*

*<servlet-name>dispatcher</servlet-name>*

*<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>*

*</servlet>*

*<servlet-mapping>*

*<servlet-name>dispatcher</servlet-name>*

*<url-pattern>/</url-pattern>*

*</servlet-mapping>*

***InternalResourceViewResolver***

* *Serve per collegare i controller alle viste (pagine HTML, JSP, etc.).*
* *Configurazione tipica:*

*java*

*Copia codice*

*@Bean*

*public InternalResourceViewResolver viewResolver() {*

*InternalResourceViewResolver resolver = new InternalResourceViewResolver();*

*resolver.setPrefix("/WEB-INF/views/");*

*resolver.setSuffix(".jsp");*

*return resolver;*

*}*

*Questo mapperà, ad esempio, una risposta dal controller alla vista /WEB-INF/views/nomeVista.jsp.*

***2. Spring Data e JPA***

* *Spring Data è un modulo per la gestione della persistenza. È un "involucro" attorno a tecnologie come JPA, Hibernate e JDBC.*
* ***JPA*** *(Java Persistence API):*
  + *Si occupa di mappare le classi Java su tabelle del database (ORM).*
  + *Usa annotazioni per gestire le entità e le relazioni tra di esse.*
  + ***Esempio di entità:***

*java*

*Copia codice*

*@Entity*

*public class User {*

*@Id*

*@GeneratedValue(strategy = GenerationType.IDENTITY)*

*private Long id;*

*private String name;*

*}*

***Repository***

* *Con Spring Data, non serve scrivere SQL per le operazioni CRUD. Basta estendere l'interfaccia JpaRepository:*

*java*

*Copia codice*

*public interface UserRepository extends JpaRepository<User, Long> {}*

***Configurazione (Spring Servlet)***

* *Il file di configurazione definisce:*
  + ***Component Scan****: dice a Spring dove trovare i controller.*

*xml*

*Copia codice*

*<context:component-scan base-package="com.example.controller" />*

* + ***Repository Scan****: individua le interfacce repository.*

*xml*

*Copia codice*

*<jpa:repositories base-package="com.example.repository" />*

***3. Spring Boot***

* *È un'estensione di Spring che semplifica la configurazione e l'avvio di applicazioni.*
* *Offre un approccio "opinionated": le configurazioni predefinite permettono di avviare un'applicazione con poco codice.*
* ***Esempio****:*

*java*

*Copia codice*

*@SpringBootApplication*

*public class Application {*

*public static void main(String[] args) {*

*SpringApplication.run(Application.class, args);*

*}*

*}*

* *Integra automaticamente moduli come Spring MVC, Spring Data e altro.*

***Filosofia di Spring***

*Spring non si limita a fornire strumenti, ma adotta una* ***filosofia*** *che semplifica lo sviluppo e la manutenzione del codice:*

1. *Minimizzare l'accoppiamento tra le componenti.*
2. *Automatizzare la gestione di cicli di vita complessi (connessioni, sessioni, transazioni).*
3. *Favorire la scrittura di codice leggibile, modulare e riutilizzabile.*

Nell’HTTP noi possiamo metterci anche una scritto, con il JSON non sa che farci, e creare una singol page che intercetta Javascript e da un significato.  
  
L'annotazione @ResponseBody in Spring è utilizzata per indicare che il **valore di ritorno di un metodo del controller** deve essere direttamente scritto nella **risposta HTTP** del client, anziché essere elaborato da una **vista** (come JSP o Thymeleaf).

In altre parole, trasforma il metodo in un **endpoint** che restituisce dati (ad esempio JSON o XML) anziché una pagina web.

**Quando si usa @ResponseBody**

1. **API REST**: Se il tuo controller deve fornire dati (come JSON o XML) anziché una pagina HTML.
2. **Risposta diretta**: Per inviare una stringa, un numero o un oggetto serializzabile al client senza passare per una vista.

**Esempio semplice**

Supponiamo di voler creare un'API che restituisca una lista di utenti in formato JSON:

**Controller con @ResponseBody**

java

Copia codice

@RestController

public class UserController {

@GetMapping("/users")

public List<User> getAllUsers() {

// Simuliamo una lista di utenti

return List.of(new User(1, "Simone"), new User(2, "Laura"));

}

}

**Cosa succede:**

1. Spring invoca il metodo getAllUsers().
2. Il valore ritornato (una lista di utenti) viene automaticamente convertito in formato JSON grazie a **Jackson** (libreria integrata in Spring Boot per la conversione).
3. Il risultato JSON viene inviato al client come risposta HTTP.

**Risultato della richiesta GET su /users**

json

Copia codice

[

{"id": 1, "name": "Simone"},

{"id": 2, "name": "Laura"}

]

**Differenza tra @Controller e @RestController**

* **@Controller**:
  + Usa @ResponseBody esplicitamente se vuoi restituire JSON/XML anziché una vista.
  + Esempio:

java

Copia codice

@Controller

public class ExampleController {

@GetMapping("/hello")

@ResponseBody

public String sayHello() {

return "Ciao Simone!";

}

}

La risposta sarà una stringa "Ciao Simone!" e non una vista HTML.

* **@RestController**:
  + È una combinazione di @Controller + @ResponseBody.
  + Ogni metodo restituisce automaticamente i dati (JSON, XML, etc.), senza bisogno di annotare ogni metodo con @ResponseBody.

**In breve:**

@ResponseBody serve per **dire a Spring di restituire direttamente il contenuto della risposta HTTP**, senza passare da una vista. Se stai creando un'API REST, puoi usare direttamente @RestController per semplificare il codice.

**Dependency Injection in Spring**

* In Spring, **qualsiasi oggetto gestito dal framework può avere una Dependency Injection (DI)**.
* **DI** permette al framework di occuparsi della creazione e inizializzazione degli oggetti, evitando al programmatore di scrivere codice manuale per farlo.
* Quando un oggetto viene iniettato, **Spring lo inizializza automaticamente, ma senza popolarlo (ad esempio con dati specifici)**.
* Se si passa un parametro (esempio: nome=Mario) tramite una richiesta HTTP (es. query string), **Spring utilizza la reflection per popolare automaticamente l'oggetto** a partire dai nomi degli attributi, che devono coincidere con i parametri passati nella richiesta.

**Spring MVC e le View**

* In Spring MVC, quando si restituisce una stringa come return "impiegato/index";, questa indica la **view JSP** da caricare.
  + Ad esempio: "impiegato/index" corrisponde al file index.jsp nella directory /WEB-INF/views/impiegato/.
* Nelle applicazioni DWP (Dynamic Web Pages), con le servlet tradizionali si usava getRequestDispatcher per passare dati alla JSP. In Spring, questo viene gestito più facilmente grazie ai **modelli e ai controller**.
  + Gli oggetti possono essere passati direttamente alla JSP tramite il modello (Model o ModelAndView), evitando di scrivere codice esplicito per il forwarding.
* **Nella stessa JSP è possibile avere più form**, purché separati logicamente, e ogni form può invocare un endpoint diverso del controller.

**REST vs. Sessioni**

* **REST** e **sessioni** sono concetti diversi:
  + Le sessioni sono utilizzate nelle applicazioni web tradizionali per mantenere uno stato tra client e server (esempio: autenticazione utente).
  + In REST, invece, si tende a evitare l'uso delle sessioni, poiché REST è **stateless** (senza stato).
    - Con React o altre tecnologie front-end, si usa spesso un token (ad esempio JWT) per gestire l'autenticazione al posto delle sessioni di server.
  + **Nota**: Non è impossibile usare sessioni con REST, ma è meno comune e non aderisce ai principi RESTful.

**Spring Data JPA**

* **Spring Data JPA** fornisce un'astrazione per lavorare con database relazionali.
* Il **repository** è un'interfaccia che rappresenta il livello di accesso ai dati.
  + Ogni entità avrà il suo repository, che estende l'interfaccia CrudRepository o JpaRepository.
  + Ad esempio:

java

Copia codice

public interface ImpiegatoRepository extends JpaRepository<Impiegato, String> {}

* + - Impiegato è l'entità mappata al database.
    - String è il tipo della chiave primaria (es. matricola).
* Spring JPA **genera automaticamente i metodi CRUD** (Create, Read, Update, Delete) e i metodi di ricerca per i repository:
  + Metodi personalizzati si possono definire usando la convenzione dei nomi:

java

Copia codice

List<Impiegato> findDistinctByNomeAndCognome(String nome, String cognome);

* + - Questo metodo genererà automaticamente una query per trovare impiegati con un determinato nome e cognome.
    - Si possono aggiungere più condizioni concatenando And:

java

Copia codice

List<Impiegato> findDistinctByNomeAndCognomeAndIndirizzo(String nome, String cognome, String indirizzo);

* Per query personalizzate più avanzate:
  + **Named Query**: Definita direttamente nell'entità con annotazioni come @NamedQuery.
  + **@Query**: Scrive una query direttamente nel repository.
  + **Query nativa**: Utilizza SQL puro. È meno portabile e rischiosa se si cambia il database.

**Configurazione di Spring JPA**

* In spring-servlet.xml (file di configurazione di Spring MVC), dobbiamo dichiarare:
  + Il **JPA repository**, specificando il package che contiene le interfacce dei repository.
  + Bean per configurare la **connessione al database**, il **transaction manager**, e le altre impostazioni necessarie.

**Come funzionano i repository in Spring**

* In un progetto tipico, i repository sono utilizzati nei **servizi**:
  + Nel servizio (ad esempio, ImpiegatoService), si inietta il repository con l'annotazione @Autowired:

java

Copia codice

@Service

public class ImpiegatoService {

@Autowired

private ImpiegatoRepository impiegatoRepository;

}

* + Quando l'applicazione parte, Spring verifica se ci sono annotazioni come @Autowired e collega automaticamente i bean definiti nel contesto.
  + Se il repository è definito correttamente, Spring usa un **dynamic proxy** ($Proxy) per fornire un'implementazione automatica dell'interfaccia.

**Dynamic Proxy**

* Spring utilizza il **Dynamic Proxy** per automatizzare le operazioni sui repository.
  + In pratica, Spring genera dinamicamente un'implementazione dell'interfaccia del repository e si occupa di eseguire le query o i metodi CRUD richiesti senza che il programmatore debba scrivere codice esplicito.

**Esempio concreto**

**Controller per gli Impiegati:**

java

Copia codice

@Controller

@RequestMapping("/impiegato")

public class ImpiegatoController {

@Autowired

private ImpiegatoService impiegatoService;

@GetMapping("/add")

public String showAddForm(Model model) {

model.addAttribute("impiegato", new Impiegato());

return "impiegato/add"; // Carica la view add.jsp

}

@PostMapping("/add")

public String addImpiegato(@ModelAttribute Impiegato impiegato) {

impiegatoService.save(impiegato);

return "redirect:/impiegato/index";

}

}

**Service per gli Impiegati:**

java

Copia codice

@Service

public class ImpiegatoService {

@Autowired

private ImpiegatoRepository impiegatoRepository;

public void save(Impiegato impiegato) {

impiegatoRepository.save(impiegato);

}

}

**Repository:**

java

Copia codice

public interface ImpiegatoRepository extends JpaRepository<Impiegato, String> {}.  
  
Un ulteriore concetto è la annotazione @Transactional, lo utilizzo dove devo andare a fare modifiche.  
  
In hibernate per esempio si può mettere l’autocommit in true, se andiamo nel sito JPA data c’è tutta la documentazione che ci spiega per filo e per segno tutti i gradi di elasticità.  
  
Domani vediamo qualcosa di spring boot, ha in pancia un application server, però non ha il tomcat, bisogna capire come deployare wildfly, perché wildfly è anche ejb container, tomcat solo servlet container.