

Terzo Sprint

Introduzione a Spring: L'inizio con un Framework

Nel mondo dello sviluppo software, affrontare complessi problemi aziendali richiede strumenti e approcci adeguati. Mentre le librerie standard offrono funzionalità di base, spesso ci troviamo a desiderare qualcosa di più alto livello che gestisca problemi comuni, ma complessi, come la connessione ai database, la gestione delle transazioni o la costruzione di interfacce web. Qui entra in gioco l'idea di un **framework**.

Un framework, a differenza di una libreria, fornisce una struttura su cui gli sviluppatori possono costruire le loro applicazioni. Mentre una libreria è una raccolta di funzioni che puoi chiamare, un framework **chiama** il tuo codice. Questo principio è spesso descritto come "**Inversione del Controllo**" (**IoC**) e rappresenta il cuore di molti framework, tra cui Spring.

Spring è uno dei framework Java più popolari. Nasce con l'obiettivo di semplificare lo sviluppo di applicazioni Java enterprise, riducendo la necessità di codice ripetitivo e aumentando la modularità e la flessibilità.

Ecco alcune ragioni per cui gli sviluppatori scelgono Spring:

- **Iniezione delle Dipendenze (DI)**: Uno dei principali vantaggi di Spring è l'implementazione dell' IOC tramite la **Dependency Injection**. Questo significa che invece di creare manualmente le dipendenze, Spring le fornisce all'oggetto al momento della sua creazione.
- **Astrazione**: Spring fornisce un livello di astrazione per molte operazioni comuni (ad es. accesso ai dati), permettendoti di concentrarti sulla logica di business piuttosto che sulle specifiche tecniche.
- **Modularità**: Spring è composto da una serie di **moduli** che si occupano di diverse aree, come l'accesso ai dati, la sicurezza, la programmazione orientata agli aspetti (AOP) e altro. Quindi possiamo scegliere quali moduli utilizzare a seconda dell'applicazione che andremo a sviluppare.
- **Integrazione**: Spring offre una facile integrazione con una vasta gamma di tecnologie, sia quelle più antiche che le più moderne.
- **Sviluppo Orientato ai Test**: Con Spring, scrivere test unitari e d'integrazione è semplificato grazie all'iniezione delle dipendenze e alle altre caratteristiche del framework.

Iniziare con Spring significa abbracciare l'idea del lavoro con un framework. Si passa da un approccio in cui si controlla tutto manualmente a uno in cui si permette al framework di gestire molte delle operazioni di routine. L'obiettivo è scrivere applicazioni più pulite, modulari e mantenibili, riducendo al contempo il tempo di sviluppo e la complessità.

Dependency Injection

La Dependency Injection (DI) è un principio di design utilizzato nella programmazione orientata agli oggetti, che permette di **ridurre l'alto accoppiamento** tra i componenti software.

La Dependency Injection è una tecnica che coinvolge tre elementi chiave: la dipendenza, il contenitore e l'iniezione.

Dipendenza: In OOP, quando un oggetto A fa riferimento o usa un oggetto B, si dice che A ha una "dipendenza" da B. Questo è spesso fatto attraverso composizione o ereditarietà.

Contenitore: Si tratta di un sistema o framework (come l' IOC container su Spring) che sa come costruire e configurare classi e le loro dipendenze.

Iniezione: È il processo di fornire la dipendenza a un oggetto. Questo può avvenire in diversi modi: tramite un costruttore, tramite un metodo setter, o tramite proprietà o campi direttamente.

Come funziona?

Senza la DI, un oggetto è responsabile della creazione e gestione delle proprie dipendenze. Questo può portare a un alto grado di accoppiamento, rendendo difficile la manutenzione, la modularità e il testing.

Con la DI, invece, l'oggetto non si preoccupa di come ottenere le sue dipendenze. Sono fornite (o "iniettate") dall'esterno, spesso da un contenitore DI.

Esempio senza Dependency Injection:

```
public class Car {  
    private Engine engine;  
  
    public Car() {  
        this.engine = new PetrolEngine();  
    }  
}
```

```

    }

    public void start() {
        engine.start();
    }
}

```

In questo esempio, la classe Car è **strettamente accoppiata** con PetrolEngine. Se vogliamo utilizzare un DieselEngine o un ElectricEngine, dovremmo modificare la classe Car direttamente.

Esempio con Dependency Injection (tramite costruttore):

```

public class Car {
    private Engine engine;

    public Car(Engine engine) {
        this.engine = engine;
    }

    public void start() {
        engine.start();
    }
}

public class Application {
    public static void main(String[] args) {
        Engine petrolEngine = new PetrolEngine();
        Car car = new Car(petrolEngine);
        car.start();
    }
}

```

In questo esempio, l'engine viene iniettato attraverso il costruttore di Car. Questo significa che possiamo facilmente **sostituire** l'engine senza cambiare il codice all'interno della classe Car.

Benefici di DI:

Decoupling: La DI aiuta a separare la logica di creazione e configurazione delle dipendenze dalla logica di business, rendendo il codice più modulare.

Testabilità: L'alto decoupling rende più semplice "mockare" o simulare dipendenze durante i test, rendendo il testing unitario più semplice e robusto.

Manutenibilità: La gestione centralizzata delle dipendenze rende il codice più pulito e facile da mantenere.

Flessibilità: Poiché le dipendenze sono iniettate e non create internamente, è più facile cambiarle, configurarle o estenderle in futuro.

Spring e Dependency Injection:

Spring è uno dei framework più popolari che supportano la Dependency Injection (DI) in Java. L'implementazione di Spring della DI è guidata principalmente attraverso il suo **IoC Container**. Ecco una panoramica dettagliata di come Spring implementa la Dependency Injection:

1. Il Container IoC di Spring

Il cuore della capacità di gestire le dipendenze di Spring risiede nel suo container IoC. Questo container è responsabile della creazione, configurazione e assemblaggio dei bean - oggetti gestiti dal container stesso. La configurazione può essere effettuata tramite XML, annotazioni Java o codice Java.

2. Bean

In Spring, gli oggetti che formano la spina dorsale dell'applicazione e che sono gestiti dal container Spring IoC sono chiamati bean. Un bean è un oggetto che viene istanziato, assemblato e gestito da un container IoC di Spring.

3. Configurazione

Prima di poter utilizzare la DI, devi configurare come e quali oggetti dovrebbero essere iniettati. Questa configurazione può avvenire in diversi modi:

- **XML:** Inizialmente, la configurazione XML era il modo standard per definire i bean e le loro dipendenze in Spring.
- **Annotazioni:** Spring 2.5 ha introdotto il supporto per l'annotazione, rendendo la configurazione più snella. Annotazioni come **@Component**, **@Service**, **@Repository**, e **@Controller** indicano che una classe è un bean. **@Autowired** è usato per indicare dove iniettare una dipendenza.
- **Java Config:** A partire da Spring 3.0, puoi configurare il container Spring usando Java. Utilizza **@Configuration** per indicare una classe di configurazione e **@Bean** per definire un bean.

4. Tipi di Iniezione

Spring supporta diversi tipi di iniezione:

- Iniezione tramite **costruttore**: La dipendenza viene iniettata tramite il costruttore dell'oggetto. Utilizza l'annotazione **@Autowired** sopra un costruttore per questo tipo di iniezione.

```

public interface Service {
    void execute();
}

@Component
public class MyService implements Service {
    public void execute() {
        System.out.println("Esecuzione di MyService");
    }
}

@Component
public class Consumer {
    private final Service service;

    @Autowired
    public Consumer(Service service) {
        this.service = service;
    }

    public void doSomething() {
        service.execute();
    }
}

```

In questo esempio, Consumer ha una dipendenza su Service, e MyService è l'implementazione di Service. Spring inietterà automaticamente un'istanza di MyService nel costruttore di Consumer.

- **Iniezione tramite **setter**:** Qui, Spring inietta le dipendenze tramite metodi setter della classe. Ancora una volta, @Autowired è l'annotazione chiave.

```

@Component
public class Consumer {
    private Service service;

    @Autowired
    public void setService(Service service) {
        this.service = service;
    }

    public void doSomething() {
        service.execute();
    }
}

```

- Iniezione basata su **campo**: Spring può iniettare direttamente nel campo, senza bisogno di setter. Anche in questo caso, @Autowired viene utilizzato.

```

@Component
public class Consumer {

    @Autowired
    private Service service;

    public void doSomething() {
        service.execute();
    }
}

```

5. Risoluzione delle Dipendenze

Quando inietti una dipendenza, Spring cerca nel suo container un bean che corrisponda al tipo richiesto. Se ce ne sono molti, puoi utilizzare l'annotazione @Qualifier per specificare quale bean utilizzare.

6. Ambito dei Bean

Spring non crea sempre un nuovo bean ogni volta che una dipendenza viene iniettata. L'ambito del bean (@Scope) determina come e quando vengono creati i bean. Gli ambiti comuni includono "singleton" (un'istanza per container) e "prototype" (una nuova istanza ogni volta).

Differenza tra Spring Bean e Java Bean

JavaBean:

Un JavaBean è una classe Java standard che segue alcune convenzioni:

- **Costruttore senza argomenti:** Un JavaBean deve avere un costruttore senza argomenti, il che rende possibile istanziare l'oggetto senza fornire informazioni iniziali.
- **Proprietà private:** Gli attributi o le variabili membro in un JavaBean sono solitamente private.
- **Accesso tramite getter e setter:** Un JavaBean fornisce metodi getter e setter pubblici per ottenere e impostare i valori delle sue proprietà. Questo encapsulamento garantisce che l'oggetto mantenga il suo stato in modo coerente.

- **Serializzabile:** Idealmente, un JavaBean dovrebbe essere serializzabile, il che significa che dovrebbe implementare l'interfaccia `java.io.Serializable`.

L'intenzione principale di un JavaBean è di rappresentare dati, come un DTO (Data Transfer Object).

Spring Bean:

Un Spring Bean si riferisce a un oggetto gestito dal **Inversion of Control Container** di Spring.

- **Gestione del ciclo di vita:** Il ciclo di vita di un Spring Bean è gestito dal container Spring, dall'istanziazione all'inizializzazione fino alla sua distruzione.
- **Iniezione delle dipendenze:** Gli oggetti che un bean potrebbe necessitare (dipendenze) possono essere iniettati nel bean dal container Spring. Questo può essere fatto tramite costruttori, setter o campi diretti.
- **Ambito:** Puoi definire l'ambito di un Spring Bean, determinando come e quando viene creato e condiviso o ricreato.
- **Configurazione:** La creazione e la configurazione dei Spring Beans possono avvenire tramite file XML, annotazioni Java o configurazione Java pura.

Creazione dei bean

Vediamo come si possono creare Spring Beans utilizzando sia `@Component` che `@Bean` e discutiamo le differenze principali tra queste due annotazioni.

1. Creazione di un bean con `@Component`:

Supponiamo di avere una semplice classe `Car` che vogliamo registrare come un bean di Spring:

```
package com.example.demo;

import org.springframework.stereotype.Component;

@Component
public class Car {
    public String getCarDetails() {
        return "A standard car";
    }
}
```

Quando si utilizza `@Component` sopra una classe, si indica a Spring di trattare quella classe come un componente e di gestirla come un bean. Inoltre, la scansione dei

componenti deve essere abilitata (ad esempio, con `@ComponentScan` o con configurazione XML) affinché Spring possa rilevare e registrare automaticamente tali componenti. In caso stessimo utilizzando Spring Boot non è necessario definire manualmente questa configurazione.

2. Creazione di un bean con `@Bean`:

Supponiamo di avere una configurazione di Spring come questa:

```
package com.example.demo;

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

@Configuration
public class AppConfig {

    @Bean
    public Car car() {
        return new Car();
    }
}
```

Con `@Bean`, stai dicendo a Spring di creare un bean a partire dal metodo annotato. Questo metodo deve restituire un'istanza dell'oggetto desiderato. Questo metodo di configurazione è tipicamente utilizzato in una classe annotata con `@Configuration`, che indica a Spring che la classe contiene definizioni di bean.

Differenze principali tra `@Component` e `@Bean`:

- **Punto di utilizzo:** `@Component` (e le sue varianti specifiche come `@Service`, `@Repository`, `@Controller`) è utilizzato direttamente sulla classe che si desidera registrare come bean. `@Bean` è utilizzato all'interno di una classe di configurazione e si riferisce a un metodo che crea e restituisce l'istanza del bean.
- **Scansione dei componenti:** Le classi annotate con `@Component` vengono automaticamente rilevate e registrate come bean se la scansione dei componenti è abilitata. I metodi annotati con `@Bean` sono utilizzati per creare esplicitamente e registrare un bean.
- **Flessibilità:** Con `@Bean`, hai una maggiore flessibilità nella creazione dell'oggetto. Puoi eseguire una logica personalizzata nel metodo prima di restituire l'oggetto. Questo può essere utile quando si configurano bean che provengono da librerie esterne o quando si desidera condizionare la creazione del bean in base a vari criteri.

- **Scopo:** @Component è generalmente utilizzato per auto-detection e per registrare bean che rappresentano componenti dell'applicazione (come servizi, repository, ecc.). @Bean è utilizzato per registrare bean quando hai bisogno di una logica personalizzata nella creazione dell'oggetto o quando lavori con oggetti che non sono sotto il tuo controllo diretto (ad esempio, oggetti da librerie esterne).
-

Persistenza dati

Spring Data costituisce il modulo di Spring che si occupa della **persistenza dati**, fornisce un approccio semplificato e standardizzato per accedere ai dati in varie fonti di dati, sia che si tratti di database relazionali, database NoSQL o persino servizi REST. Il suo obiettivo principale è di ridurre al minimo la quantità di codice boilerplate che uno sviluppatore deve scrivere per eseguire operazioni CRUD e query complesse.

Durante la formazione vedremo un modulo specifico di Spring Data, ovvero **Spring Data JPA**. Preciso che questo non è l'unico modo di lavorare con le persistenze dati su Spring ma esistono altri sotto moduli, tra cui:

- **Spring Data JDBC:** Questa sotto-proposta si concentra sulla persistenza relazionale utilizzando JDBC. È più leggera rispetto a JPA e non offre una mappatura oggetto-relazionale completa, ma può essere preferibile per chi cerca una soluzione più semplice senza il sovraccarico di JPA.
- **Spring Data MongoDB:** Questa sotto-proposta fornisce supporto per MongoDB, un popolare database NoSQL basato su documenti. Con questa libreria, puoi sfruttare le funzionalità di Spring Data mentre lavori con MongoDB.
- **Spring Data Redis:** Questa sotto-proposta supporta Redis, un database in memoria utilizzato spesso come cache o come broker di messaggi.
- **Spring Data Cassandra:** Questa sotto-proposta fornisce supporto per Cassandra, un sistema di database distribuito NoSQL.
- **Spring Data Elasticsearch:** Offre integrazione con Elasticsearch, una piattaforma di ricerca e analisi.

Ogni sotto-proposta di Spring Data è progettata per semplificare l'interazione con una particolare fonte di dati o tecnologia di persistenza. La forza di Spring Data è che, indipendentemente dalla sotto-proposta che stai utilizzando, l'approccio generale e l'interfaccia sono abbastanza simili, rendendo più facile per gli sviluppatori adattarsi a diverse tecnologie di persistenza.

Spring Data JPA

Spring Data JPA è uno dei sottoprogetti del progetto Spring Data. È stato creato per semplificare l'interazione con i database relazionali attraverso l'uso di **JPA (Java Persistence API)**. Spring Data JPA non è una implementazione JPA in sé; piuttosto, agisce come un layer tra l'applicazione e l'implementazione JPA effettiva ovvero **Hibernate** (lo approfondiremo successivamente).

Vediamo i punti di forza di Spring Data JPA:

1. Obiettivo:

L'obiettivo principale di Spring Data JPA è quello di ridurre il codice **boilerplate** associato all'implementazione dei Data Access Object (DAO) o dei repository. Ciò viene fatto attraverso la fornitura di un set di API di alto livello e l'uso di convenzioni.

2. Repository:

Il concetto chiave in Spring Data JPA è il **Repository**. Un repository è essenzialmente un **DAO** ma viene gestito da Spring e sfrutta la potenza delle convenzioni per ridurre la quantità di codice che uno sviluppatore deve scrivere. Esistono diverse interfacce che possono essere estese:

- **Repository<T, ID>**: L'interfaccia base. Non ha metodi specifici.
- **CrudRepository<T, ID>**: Estende Repository e fornisce metodi CRUD.
- **PagingAndSortingRepository<T, ID>**: Estende CrudRepository e aggiunge supporto per la paginazione e l'ordinamento.
- **JpaRepository<T, ID>**: Estende PagingAndSortingRepository e aggiunge altre funzionalità JPA, come la capacità di effettuare flush e delete in batch.

3. Query Methods:

Uno dei punti di forza di Spring Data JPA è la capacità di definire query semplicemente **nominando** i metodi in un modo specifico. Per esempio:

```
public interface UserRepository extends JpaRepository<User, Long> {  
    List<User> findByLastName(String lastName);  
}
```

In questo esempio, Spring Data JPA genererà automaticamente un'implementazione del metodo `findByLastName` basata sul nome del metodo stesso. **Importante** notare che i nomi dei campi utilizzati nei Query Methods devono corrispondere ai nomi degli attributi del Model relativo alla repository.

4. **@Query Annotation:**

Per le query più complesse o quelle che non possono essere descritte utilizzando la convenzione dei nomi dei metodi, è possibile utilizzare l'annotazione `@Query`:

```
public interface UserRepository extends JpaRepository<User, Long> {
    @Query("SELECT u FROM User u WHERE u.email = ?1")
    User findByEmail(String email);
}
```

Spring Data JPA semplifica in modo significativo la creazione di applicazioni basate su dati con Spring. Elimina gran parte del codice boilerplate associato all'interazione con i database relazionali, permettendo agli sviluppatori di concentrarsi sulla logica di business. Con il supporto per le query generate automaticamente, le custom query e un'ampia serie di operazioni CRUD out-of-the-box, gli sviluppatori possono realizzare applicazioni robuste e scalabili con meno sforzo e in meno tempo.

Hibernate

Hibernate è un framework per la **mappatura oggetto-relazionale (ORM)** in Java, che fornisce un modo per mappare oggetti Java a tabelle di database e viceversa. È stato creato per risolvere il problema dell'impedenza tra il mondo degli oggetti e il mondo relazionale dei database. Ecco una panoramica dettagliata di Hibernate:

Cos'è un ORM (Object-Relational Mapping)

ORM, che sta per "Object-Relational Mapping", è una tecnica di programmazione per **convertire** i dati tra il sistema di database relazionale e gli oggetti in OOP (Object-Oriented Programming). La necessità dell'ORM nasce dal fatto che le tecniche di persistenza tradizionali basate su database relazionali e i linguaggi di programmazione orientati agli oggetti hanno modi differenti di **rappresentare i dati e le relazioni**.

Le applicazioni orientate agli oggetti rappresentano i dati sotto forma di **oggetti** con proprietà (campi) e comportamenti (metodi). I database relazionali, d'altro canto, rappresentano i dati sotto forma di **tabelle**, righe e colonne. Questa discrepanza, nota

come "**problema dell'impedenza**", può rendere complesso il trasferimento di dati tra codice orientato agli oggetti e database relazionali.

L'ORM funge da "**ponte**" tra queste due rappresentazioni, consentendo agli sviluppatori di interagire con il database in un modo che sia naturale dal punto di vista del linguaggio di programmazione, ma che allo stesso tempo mantenga l'integrità e le prestazioni del database. Piuttosto che scrivere SQL "a mano" o utilizzare query string per operare su un database, l'ORM permette agli sviluppatori di utilizzare il proprio linguaggio di programmazione nativo.

Vantaggi dell'ORM:

- **Produttività:** Si riduce la quantità di codice da scrivere, dato che non è necessario scrivere query SQL per ogni operazione di CRUD (Create, Read, Update, Delete).
- **Manutenibilità:** Se la struttura del database cambia, gli aggiustamenti possono essere fatti in un unico posto (il mappatore ORM) invece che in tutte le query nel codice.
- **Indipendenza dal database:** Molte librerie ORM supportano diversi database. Ciò significa che si potrebbe, in teoria, cambiare il database sottostante con poche o nessuna modifica al codice dell'applicazione.
- **Cache:** Alcuni sistemi ORM includono una cache di primo livello che può migliorare le prestazioni riducendo la necessità di query al database per oggetti già recuperati.

Architettura principale

L'architettura principale di Hibernate è composta dai seguenti elementi:

SessionFactory: Una volta configurata e creata, fornisce le sessioni di Hibernate. Di solito c'è una sola SessionFactory per l'intera applicazione e si crea all'avvio dell'applicazione.

Session: Rappresenta una **singola unità di lavoro con il database**. È utilizzata per ottenere una connessione fisica con il database. La sessione fornisce metodi CRUD per salvare, aggiornare, cancellare o caricare oggetti.

Transaction: Rappresenta l'unità di lavoro con il database e garantisce che una serie di operazioni venga eseguita in modo **atomico**, il che significa che o tutte le operazioni vengono eseguite con successo, o nessuna viene eseguita. Se una qualsiasi delle operazioni fallisce, tutte le modifiche vengono annullate (rollback).

Query e Criteria: Permette di fare delle query. Mentre la Query utilizza stringhe HQL (Hibernate Query Language), Criteria utilizza metodi orientati agli oggetti per le query.

Mappatura

La mappatura tra classi Java e tabelle di database avviene attraverso file XML o, come vedremo noi, attraverso annotazioni Java.

Ad esempio, se si dispone di una classe Student e si desidera mapparla a una tabella STUDENT, si può fare come segue usando annotazioni:

```
@Entity
@Table(name="STUDENT")
public class Student {
    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    @Column(name="id")
    private int id;

    @Column(name="first_name")
    private String firstName;

    // ... altri campi e metodi
}
```

- **@Entity:** Questa annotazione segnala che la classe Student è una classe di **entità**, che rappresenta una **tabella** nel database. In pratica, indica al framework JPA/Hibernate che questa classe dovrebbe essere mappata su una tabella nel database.
- **@Table(name="STUDENT"):** Questa annotazione è utilizzata per specificare il **nome della tabella** a cui la classe di entità deve essere mappata. Se questa annotazione non viene fornita, il nome della tabella verrà dedotto dal nome della classe di entità. In questo caso, la classe Student sarà mappata sulla tabella "STUDENT" nel database.
- **@Id:** Questa annotazione segnala che il campo successivo (id in questo caso) rappresenta la **chiave primaria** della tabella. Ogni classe di entità **deve avere** una chiave primaria, e questa annotazione identifica quale campo o proprietà della classe rappresenta tale chiave primaria.
- **@GeneratedValue(strategy=GenerationType.IDENTITY):** Questa annotazione indica che il valore della chiave primaria (in questo caso, id) sarà **generato automaticamente**. La strategia GenerationType.IDENTITY significa che il valore sarà generato dalla colonna stessa, spesso attraverso

un'auto-incremento. Ciò dipende dal tipo di database in uso, ma molti database RDBMS, tra cui MYSQL, supportano colonne auto-incremento per le chiavi primarie.

- **@Column(name="...")**: Questa annotazione specifica che il campo Java mappato verrà associato a una **colonna** con un nome specifico nella tabella del database. Ad esempio, l'annotazione @Column(name="first_name") sul campo firstName indica che il campo firstName della classe Student sarà mappato sulla colonna "first_name" nella tabella "STUDENT". Se l'annotazione @Column non è presente, il nome della colonna verrà dedotto dal nome del campo o della proprietà Java.

Queste annotazioni sono una parte fondamentale del funzionamento di JPA e Hibernate e permettono di stabilire un mappatura tra le classi Java e le tabelle del database, senza dover scrivere codice SQL esplicito.

Relazioni tra entità

Le relazioni tra le entità sono fondamentali nella modellazione dei dati, e Hibernate supporta tutte le tipologie di relazioni comuni nei database relazionali:

- **ManyToOne/OneToMany**
- **OneToOne**
- **ManyToMany**

ManyToOne - OneToMany:

Immaginiamo di avere due entità: Post e Comment. Un Post può avere molti Comment, ma ogni Comment appartiene a un solo Post.

Abbiamo 2 modi per definire questa relazione:

- **Unidirezionale**
- **Bidirezionale**

Unidirezionale

- **OneToMany**: In una relazione unidirezionale OneToMany, la parte "uno" della relazione conosce la parte "molti", ma non viceversa. Consideriamo il seguente esempio che utilizza un Post e i suoi Comment:

```
@Entity  
public class Post {  
  
    @Id  
    @GeneratedValue(strategy=GenerationType.IDENTITY)  
    private Long id;
```

```

    private String title;

    @OneToMany(cascade = CascadeType.ALL, orphanRemoval =
true)
    @JoinColumn(name = "post_id")
    private List<Comment> comments = new ArrayList<>();

}

@Entity
public class Comment {

    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Long id;

    private String text;

    // NOTA: Non c'è nessun riferimento a Post qui!

}

```

In questa configurazione:

La relazione viene gestita solo dal lato Post attraverso il campo comments. L'annotazione @JoinColumn(name = "post_id") nel Post indica che la colonna post_id nella tabella Comment è la chiave esterna che stabilisce la relazione.

- **ManyToOne:** In una relazione unidirezionale ManyToOne, la parte "molti" della relazione conosce la parte "uno", ma non viceversa. Consideriamo l'opposto dell'esempio precedente:

```

@Entity
public class Comment {

    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Long id;

    private String text;

    @ManyToOne
    @JoinColumn(name = "post_id")
    private Post post;

    // getter, setter, ecc.
}

```

```

@Entity
public class Post {

    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Long id;

    private String title;

    // NOTA: Non c'è nessun riferimento a Comment qui!

    // getter, setter, ecc.
}

```

In questa configurazione:

La relazione viene gestita solo dal lato Comment attraverso il campo post.

L'annotazione `@JoinColumn(name = "post_id")` in Comment indica che la colonna `post_id` nella tabella Comment è la chiave esterna che stabilisce la relazione con Post.

In entrambi i casi, poiché la relazione è unidirezionale, una delle due entità non ha un riferimento all'altra, rendendo la relazione "a senso unico".

Ma quale delle due soluzioni è la migliore?

Generalmente, una relazione unidirezionale ManyToOne è preferita per le seguenti ragioni:

- **Performance:** Hibernate può avere problemi di performance con le relazioni OneToMany unidirezionali, specialmente quando si tratta di inserimenti e aggiornamenti.
- **Naturalità del Modello Relazionale:** In un database relazionale, è naturale avere chiavi esterne nella tabella "molti" che puntano alla tabella "uno". Una relazione ManyToOne si adatta meglio a questo modello.
- **Semplicità:** Le relazioni ManyToOne sono spesso più semplici da gestire, dato che ci si concentra sulla gestione delle entità "molti" e non si ha bisogno di preoccuparsi di mantenere una lista nel lato "uno".

Tuttavia, la decisione finale dovrebbe essere basata sulle specifiche esigenze dell'applicazione. Se, ad esempio, si prevede di accedere spesso alle entità figlie attraverso l'entità genitore, una relazione OneToMany potrebbe avere senso.

Bidirezionale

Una relazione bidirezionale, in particolare nel contesto di una relazione uno-a-molti, può introdurre complessità aggiuntiva nella gestione delle entità. Questo può comportare, in alcune circostanze, una riduzione delle performance dell'applicazione. Tuttavia, quando si presenta la necessità di navigare e accedere all'entità associata da entrambi i lati della relazione, la bidirezionalità diventa utile e può essere implementata. Andiamo a vedere come:

- L'entità Post avrà una lista di Comment. Questa lista sarà mappata alla proprietà post dell'entità Comment.

```
@Entity
public class Post {

    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Long id;

    private String title;

    @OneToMany(mappedBy = "post", cascade = CascadeType.ALL,
    orphanRemoval = true)
    private List<Comment> comments = new ArrayList<>();

    // getter, setter, ecc.

    public void addComment(Comment comment) {
        comments.add(comment);
        comment.setPost(this);
    }

    public void removeComment(Comment comment) {
        comments.remove(comment);
        comment.setPost(null);
    }
}
```

- L'entità Comment avrà un riferimento al Post. La chiave esterna nel database sarà gestita dall'entità Comment.

```
@Entity
public class Comment {

    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Long id;

    private String text;
```

```

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "post_id")
    private Post post;

    // getter, setter, ecc.
}

```

Dettagli importanti:

- **mappedBy:** L'attributo mappedBy indica il lato "**proprietario**" della relazione. In questo caso, la proprietà post nell'entità Comment è il lato proprietario. Ciò significa che quando ci sono aggiornamenti da fare sulla chiave esterna, Hibernate guarderà il valore di post in Comment.
- **Metodi di comodità:** Nell'entità Post, abbiamo aggiunto metodi **addComment** e **removeComment** come metodi di comodità per gestire sia il lato Post che Comment della relazione. È buona pratica usare questi metodi ogni volta che si desidera aggiungere o rimuovere un commento, per mantenere la coerenza nell'oggetto.
- **FetchType.LAZY:** Usiamo il caricamento **lazy** (FetchType.LAZY) per evitare il caricamento immediato di tutti i commenti ogni volta che recuperiamo un Post. Questo può migliorare notevolmente le prestazioni, specialmente quando ci sono molti commenti.
- **Cascade e orphanRemoval:** L'attributo cascade consente di propagare determinate operazioni Hibernate sulle entità correlate. Ad esempio, se si elimina un Post, tutti i suoi Comment saranno eliminati. L'attributo orphanRemoval indica ad Hibernate di eliminare le entità figlie che non sono più associate all'entità genitore.

OneToOne

Una relazione One-to-One rappresenta una connessione tra due entità, dove un'entità può avere al massimo una singola istanza dell'altra e viceversa. Ad esempio, pensiamo alla relazione tra una persona e il suo passaporto: ogni persona può avere al massimo un passaporto e ogni passaporto appartiene a una sola persona.

Unidirezionale

Per rappresentare una relazione One-to-One con Hibernate, si possono utilizzare le annotazioni **@OneToOne**, **@JoinColumn** e, optionalmente, **@PrimaryKeyJoinColumn**.

Ecco un esempio pratico utilizzando due entità: Persona e Passaporto.

1. Creare l'entità Passaporto:

```
@Entity
public class Passaporto {

    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Long id;

    private String numeroPassaporto;

    // Getters, setters e altri metodi
}
```

2. Creare l'entità Persona e stabilire una relazione One-to-One con Passaporto:

```
@Entity
public class Persona {

    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Long id;

    private String nome;

    @OneToOne(cascade = CascadeType.ALL)
    @JoinColumn(name = "passaporto_id")
    private Passaporto passaporto;

    // Getters, setters e altri metodi
}
```

In questo esempio:

`@OneToOne`: Indica che c'è una relazione One-to-One tra Persona e Passaporto.

`@JoinColumn(name = "passaporto_id")`: Specifica che la colonna `passaporto_id` nella tabella Persona rappresenta la chiave esterna (foreign key) alla tabella Passaporto.

Dettagli importanti:

- **Bidirezionalità:** Nell'esempio fornito, abbiamo definito una relazione unidirezionale, cioè possiamo accedere al Passaporto da una Persona, ma non il contrario. Se abbiamo bisogno di accedere alla Persona da un Passaporto, possiamo rendere la relazione bidirezionale aggiungendo un campo `persona`

nell'entità Passaporto e annotandolo con **@OneToOne(mappedBy = "passaporto")**.

- **Chiavi Primarie Condivise:** Un altro approccio per implementare una relazione One-to-One è utilizzare chiavi primarie condivise. Ciò significa che l'entità Passaporto avrebbe la stessa chiave primaria dell'entità Persona. Per implementare questo, si potrebbe usare l'annotazione `@PrimaryKeyJoinColumn`.
- **Lazy Loading vs Eager Loading:** Hibernate, di default, utilizza il lazy loading per le relazioni One-to-One. Questo significa che l'entità collegata (in questo caso Passaporto) non viene caricata immediatamente quando si carica un'entità Persona, ma solo quando si accede al campo passaporto. Questo comportamento può essere modificato usando l'attributo fetch dell'annotazione `@OneToOne`.

ManyToMany

Una relazione Many-to-Many si verifica quando molti record in una tabella sono associati a molti record in un'altra tabella. Ad esempio, considera la relazione tra studenti e corsi: uno studente può iscriversi a molti corsi e un corso può avere molti studenti iscritti.

Implementare una relazione Many-to-Many con Hibernate richiede l'uso delle annotazioni **@ManyToMany**, **@JoinTable** e, potenzialmente, **@JoinColumn**.

Ecco un esempio pratico utilizzando due entità: Studente e Corso.

1. Creare l'entità Corso:

```
@Entity
public class Corso {

    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Long id;

    private String nome;

    @ManyToMany(mappedBy = "corsi")
    private Set<Studente> studenti = new HashSet<>();

    // Getters, setters e altri metodi
}
```

2. Creare l'entità Studente e stabilire una relazione Many-to-Many con Corso:

```

@Entity
public class Studente {

    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Long id;

    private String nome;

    @ManyToMany
    @JoinTable(
        name = "studente_corso",
        joinColumns = @JoinColumn(name = "studente_id"),
        inverseJoinColumns = @JoinColumn(name = "corso_id"))
    private Set<Corso> corsi = new HashSet<>();

    // Getters, setters e altri metodi
}

```

Nell'esempio:

- **@ManyToMany:** Indica che c'è una relazione Many-to-Many tra Studente e Corso.
- **@JoinTable:** Specifica la tabella di collegamento (chiamata spesso "tabella ponte" o "tabella di associazione") che viene utilizzata per mappare la relazione Many-to-Many. La tabella di collegamento avrà colonne di chiave esterna per entrambe le entità coinvolte.
- **name:** Il nome della tabella di collegamento.
- **joinColumns:** La chiave esterna per l'entità corrente (in questo caso Studente).
- **inverseJoinColumns:** La chiave esterna per l'entità opposta (in questo caso Corso).

Tabella di associazione

La tabella di associazione, spesso chiamata "**tabella ponte**" o "**tabella di giunzione**", viene utilizzata per mappare le relazioni Many-to-Many tra due entità. Mentre in molti casi questa tabella può semplicemente contenere chiavi esterne per le due entità coinvolte, ci sono situazioni in cui potresti voler **immagazzinare informazioni aggiuntive** relative all'associazione stessa. In tali scenari, conviene considerare la tabella di associazione come una vera e propria entità.

Implementazione

Supponiamo di avere due entità: **Studente** e **Corso**. Oltre a voler mappare quali studenti sono iscritti a quali corsi, vogliamo anche registrare la **data di iscrizione** di uno studente a un corso.

In questo caso, creeremo una **terza entità**, ad esempio **Iscrizione**, che rappresenta la relazione tra uno Studente e un Corso e include anche la data di iscrizione.

1. Definizione della tabella Iscrizione. Andremo ad inserire il riferimento alle due tabelle Studente e Corso utilizzando la relazione ManyToOne.

```
@Entity
public class Iscrizione {

    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Long id;

    @ManyToOne
    private Studente studente;

    @ManyToOne
    private Corso corso;

    private LocalDate dataIscrizione;

    // Getters, setters e altri metodi
}
```

2. Modifica delle entità Studente e Corso. Possiamo decidere anche in questo caso se rendere la relazione tra Studente/Corso ed Iscrizione Unidirezionale o Bidirezionale, inserendo o meno il riferimento alla tabella Iscrizione. In questo caso tratteremo il caso unidirezionale:

```
@Entity
public class Studente {

    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Long id;
    private String nome;

    // Getters, setters e altri metodi
}

@Entity
public class Corso {
```

```

@Id
@GeneratedValue(strategy=GenerationType.IDENTITY)
private Long id;
private String nome;

// Getters, setters e altri metodi
}

```

In conclusione, trattare la tabella di associazione come una vera entità è una pratica consigliata quando si hanno informazioni aggiuntive da memorizzare sulla relazione stessa. Se non ci sono tali informazioni, un approccio Many-to-Many standard potrebbe essere sufficiente.

Query derivate

Le query derivate sono una delle caratteristiche più potenti e convenienti di Spring Data JPA. Esse consentono di definire metodi CRUD **senza dover scrivere effettivamente una query SQL**. Sarà Spring Data JPA a generare automaticamente la query in base al nome del metodo.

Le query derivate si basano su un linguaggio di naming molto specifico e seguono una certa convenzione:

- **Nome dell'attributo:** Le query derivate si basano in gran parte **sul nome degli attributi dell'entità**. Ad esempio, se la tua entità User ha un attributo chiamato username, puoi creare un metodo `findByUsername(String username)` nel tuo repository. Spring Data JPA sarà in grado di interpretarlo e generare la query SQL/JPQL appropriata per cercare gli utenti in base al loro username.
- **Condizioni composte:** Puoi combinare più attributi per formare una condizione composta. Ad esempio, se la tua entità User ha attributi username e password, puoi avere un metodo come `findByUsernameAndPassword(String username, String password)`. Questo genererà una query che cercherà un utente in base a entrambi questi attributi.
- **Operatori condizionali:** Spring Data JPA supporta anche una varietà di operatori condizionali come GreaterThan, LessThan, Like, etc., che possono essere utilizzati con gli attributi. Ad esempio, se la tua entità User ha un attributo age, puoi avere metodi come `findByAgeGreater Than(int age)` o `findByAgeLessThanEqual(int age)`.
- **Attributi annidati:** Se un'entità ha relazioni con altre entità, puoi anche riferirti agli attributi di quelle entità annidate. Supponiamo che l'entità User abbia un attributo address di tipo Address, e Address ha un attributo city. Puoi creare un

metodo come `findByAddressCity(String city)` per cercare tutti gli utenti di una particolare città.

- **Ignorare la sensibilità delle maiuscole/minuscole:** In alcuni database, la comparazione delle stringhe è sensibile alle maiuscole/minuscole. Con Spring Data JPA, puoi utilizzare parole chiave come `IgnoreCase` per ignorare la sensibilità delle maiuscole/minuscole durante la comparazione. Ad esempio, `findByUsernameIgnoreCase(String username)`.

Importante: La corrispondenza tra il nome del metodo e gli attributi dell'entità è fondamentale per il funzionamento delle query derivate. Se commetti un errore nel nome, come un errore di battitura nell'attributo, Spring Data JPA solleverà un'eccezione al momento dell'avvio dell'applicazione, indicando che non può creare una query per quel metodo.

Alcuni esempi di query derivate:

- **findByUsername(String username):** Questo metodo recupererà tutti gli utenti con un dato username.
- **findByAgeGreaterThanOrEqual(int age):** Questo metodo recupererà tutti gli utenti con un'età superiore al valore fornito.
- **findByLastNameAndFirstName(String lastName, String firstName):** Questo metodo recupererà tutti gli utenti con il cognome e il nome forniti.

Altre funzionalità delle query derivate:

- **Ordinamento:** Puoi anche specificare l'ordinamento dei risultati utilizzando `OrderBy` seguito dal campo su cui si desidera ordinare e la direzione (Asc o Desc). Per esempio, `findByAgeOrderByNameAsc(int age)` recupererà gli utenti di una certa età e li ordinerà in base al cognome in ordine ascendente.
- **Limite nei risultati:** Puoi limitare il numero di risultati utilizzando `Top` o `First`. Per esempio, `findFirst3ByAge(int age)` recupererà i primi tre utenti di una determinata età.

Spring Boot

Spring Boot è un progetto all'interno dell'ecosistema Spring che mira a semplificare il processo di **configurazione** e **esecuzione** di applicazioni basate su Spring. È stato

creato per affrontare la crescente complessità e il boilerplate associati alla configurazione delle applicazioni Spring. Offre una serie di convenzioni predefinite e un ambiente runtime **autonomo** che comprende il **web Server Tomcat**, permettendo agli sviluppatori di eseguire e runnare le loro applicazioni Spring.

Caratteristiche principali di Spring Boot:

- **Auto-configurazione:** Spring Boot tenta di individuare automaticamente la configurazione richiesta basandosi sulle dipendenze del progetto.
- **Standalone:** Le applicazioni Spring Boot sono applicazioni web standalone che possono essere eseguite da una riga di comando senza necessità di un server esterno.
- **Nessuna generazione di codice:** Spring Boot non genera codice e non c'è assolutamente nessun requisito per XML.

Differenze tra Spring e Spring Boot:

Configurazione vs Convenzione: Mentre Spring offre un potente set di configurazioni, richiede una certa quantità di setup per una nuova applicazione. Spring Boot, al contrario, segue l'approccio "convenzione sopra la configurazione", fornendo configurazioni di default e affinando le impostazioni solo quando necessario.

Runtime: Spring necessita di un server web o di un'applicazione per essere eseguito, come Tomcat. Spring Boot, d'altro canto, viene fornito con Tomcat, Jetty o Undertow incorporati, eliminando la necessità di un server esterno.

Dipendenze: Spring Boot introduce un concetto chiamato "Starter Dependencies", che sono un set di dipendenze predefinite che facilitano la gestione delle librerie. Questo elimina la necessità di specificare versioni di librerie compatibili, poiché tutto ciò è gestito dagli "starter".

Proprietà esternalizzate: Spring Boot introduce un modo per esternalizzare la configurazione tramite proprietà e file YAML, rendendo più facile la configurazione di vari ambienti. Queste proprietà verranno definite nel file "**application.properties**" che troviamo nella cartella **resources**. Qui possiamo ad esempio trovare le configurazioni per il database.

In conclusione, mentre Spring offre un framework potente e flessibile per lo sviluppo di applicazioni Java, Spring Boot rende questo processo molto più semplice e veloce. Permette agli sviluppatori di concentrarsi sulla scrittura del codice di business, riducendo il boilerplate e la complessità della configurazione.

Lombok

Lombok è una libreria Java che fornisce un modo per **eliminare il codice boilerplate** nei progetti Java, in particolare per le classi POJO (Plain Old Java Objects). Esso genera **automaticamente** metodi come getter, setter, costruttori, hashCode(), equals(), toString() e altro attraverso l'uso di annotazioni. Lombok fa ciò durante la fase di **compilazione**, il che significa che non ha alcun overhead a runtime.

Annotation Principali

1. **@Getter e @Setter**: Queste annotazioni possono essere applicate a livello di classe o di campo. Quando applicate a una classe, Lombok genera automaticamente getter e setter per ogni campo nella classe. Se applicate a un singolo campo, genereranno getter e setter solo per quel campo.
2. **@NoArgsConstructor, @AllArgsConstructor, @RequiredArgsConstructor**: Queste annotazioni servono a generare costruttori:
 - @NoArgsConstructor genera un costruttore senza argomenti.
 - @AllArgsConstructor genera un costruttore con argomenti per tutti i campi della classe.
 - @RequiredArgsConstructor genera un costruttore solo per quei campi **final** o con l'annotazione **@NonNull**.
3. **@EqualsAndHashCode**: Genera implementazioni dei metodi **hashCode()** e **equals()**. Puoi personalizzare il comportamento escludendo determinati campi o utilizzando solo specifici campi.
4. **@ToString**: Genera un metodo **toString()**. Come @EqualsAndHashCode, ha opzioni per escludere campi o per utilizzare solo specifici campi.
5. **@Data**: È una combinazione delle annotazioni sopra menzionate. Applicando @Data a una classe, otterrai getter, setter, equals(), hashCode(), toString() e un costruttore per tutti i campi final.
6. **@Builder**: Fornisce un modo per implementare il pattern Builder. Genera un costruttore, un setter per ogni campo e un metodo build() per creare l'oggetto. Vediamo un esempio di come si utilizza:

Inseriamo l'annotation sopra la classe.

```
@Builder  
public class UserDTO {  
  
    private Long id;  
  
    private String username;  
  
    private String password;
```

```
    private Usertype usertype;  
  
}
```

Richiamiamo il metodo builder valorizzando i campi che ci interessano. Questo metodo ci restituirà un oggetto UserDTO con i campi che abbiamo deciso di valorizzare. Ci evita di creare costruttori diversi a seconda delle esigenze.

```
UserDTO.builder()  
    .id(user.getId())  
    .username(user.getUsername())  
    .password(user.getPassword())  
    .usertype(user.getUsertype())  
    .build()
```

7. **@NotNull**: Questa annotazione può essere utilizzata su un campo o un parametro. Se il campo o il parametro viene impostato su null, Lombok genererà un NullPointerException.
8. **@Value**: Simile a @Data, ma rende tutto immutabile. Quindi rende tutti i campi final, genera getter ma non setter, e un costruttore per tutti i campi.

Java Stream

Le Java Streams rappresentano una potente aggiunta a Java 8 e sono diventate uno degli strumenti più utilizzati quando si tratta di **manipolare collezioni** di dati in modo **funzionale** e **dichiarativo**. Introducono un nuovo modo di lavorare con i dati che differisce dalle classiche iterazioni imperativi come for o while.

Cos'è uno Stream?

Uno Stream in Java è una sequenza di elementi (ad esempio, una lista o un set) su cui è possibile eseguire operazioni in **sequenza o in parallelo**. Gli Stream sono progettati per supportare operazioni funzionali su sequenze di elementi, come **map, filter, reduce, etc.**

Caratteristiche principali:

- **No Storage:** Uno Stream non è una struttura dati, ma una vista su una sorgente di dati, come una collezione, un array o un generatore di input/output. Non memorizza i dati.
- **Functional in nature:** Le operazioni sugli Stream non modificano la sorgente; restituiscono un nuovo Stream con i risultati delle operazioni.
- **Laziness-seeking:** Molte operazioni sugli Stream, come filter o map, sono "lazy" e vengono eseguite solo quando è strettamente necessario.
- **Possibility of parallel execution:** Alcune operazioni sugli Stream possono essere eseguite in parallelo, migliorando le prestazioni.

Metodi intermedi vs. Metodi terminali

Gli Stream hanno due tipi di operazioni: **metodi intermedi** e **metodi terminali**.

- **Metodi intermedi:** Sono operazioni che trasformano uno Stream in un altro Stream. Sono "lazy", il che significa che non vengono eseguiti fino a quando non si invoca un'operazione terminale sullo Stream. Possono essere collegati in catena, ovvero eseguiti uno dopo l'altro. Esempi comuni sono map, filter, flatMap, sorted, distinct, ecc.

```
List<String> list = Arrays.asList("a", "b", "c");
Stream<String> stream = list.stream().filter(element ->
element.contains("b"));
```

In questo esempio, filter è un'operazione intermedia. Non viene eseguita finché non si invoca un'operazione terminale sullo Stream.

- **Metodi terminali:** Sono operazioni che producono un risultato o un effetto collaterale. Una volta eseguita un'operazione terminale, lo Stream non può più essere utilizzato. Esempi comuni sono collect, forEach, reduce, findFirst, anyMatch, ecc.

```
List<String> list = Arrays.asList("a", "b", "c");
long count = list.stream().filter(element ->
element.contains("b")).count();
```

In questo esempio, count è un'operazione terminale che restituisce il numero di elementi nello Stream dopo l'applicazione del filtro.

In sintesi:

Le Java Streams forniscono un modo più dichiarativo e funzionale per lavorare con le collezioni. I metodi intermedi permettono di descrivere come i dati devono essere

trasformati, mentre i metodi terminali indicano come i dati trasformati devono essere consumati o restituiti.

Analizziamo 3 dei metodi più utilizzati ovvero map, filter e collect:

1. **map:** Il metodo map è un'operazione intermedia che trasforma ciascun elemento di uno Stream in un altro tipo, generando uno Stream di tipo diverso.

Esempio:

Supponiamo di avere una lista di numeri e vogliamo creare una nuova lista con i quadrati di questi numeri:

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
List<Integer> squaredNumbers = numbers.stream().map(n -> n *
n).collect(Collectors.toList());
System.out.println(squaredNumbers); // [1, 4, 9, 16, 25]
```

2. **filter:** Il metodo filter è un'operazione intermedia che seleziona gli elementi di uno Stream in base a una certa condizione, restituendo un nuovo Stream con solo gli elementi che soddisfano quella condizione.

Esempio:

Dalla stessa lista di numeri, supponiamo di voler selezionare solo i numeri pari:

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
List<Integer> evenNumbers = numbers.stream().filter(n -> n %
2 == 0).collect(Collectors.toList());
System.out.println(evenNumbers); // [2, 4]
```

3. **collect:** Il metodo collect è un'operazione terminale che trasforma un Stream in una struttura dati o un altro tipo di oggetto. Si utilizza spesso un Collector per specificare come i dati devono essere raccolti.

Esempio:

Dopo aver utilizzato map o filter (o altri metodi intermedi) su uno Stream, potremo voler raccogliere i risultati in una lista:

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
List<Integer> evenNumbers = numbers.stream()
.filter(n -> n % 2 == 0)
.collect(Collectors.toList());
System.out.println(evenNumbers); // [2, 4]
```

Nell'esempio, abbiamo utilizzato il Collector Collectors.toList() per raccogliere gli elementi del Stream filtrato in una List.

Esistono tantissimi altri metodi da utilizzare con le Stream che vi consiglio di andare ad approfondire.

Analisi del codice

Partiamo con l'analisi del file **pom.xml**. Come abbiamo già visto all'interno di questo file troveremo tutte le dipendenze del nostro progetto.

Cerchiamo di analizzare le parti più importanti:

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>2.4.0</version>
  <relativePath/>
</parent>
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>2.4.0</version>
  <relativePath/>
</parent>
```

La sezione `<parent>` nel pom.xml di un progetto Maven indica un "parent POM". Questo concetto è usato per creare una sorta di ereditarietà tra i POM. Il POM padre fornisce una configurazione di default che può essere ereditata dai suoi POM figli.

Nel nostro caso `spring-boot-starter-parent` rappresenta il nostro POM padre, vediamo perchè è così utile:

- **Gestione Centralizzata delle Versioni delle Dipendenze:** Il POM padre stabilisce delle versioni predefinite per le dipendenze, garantendo coerenza e compatibilità tra le varie sotto-dipendenze. Di conseguenza, quando si aggiungono dipendenze Spring Boot al progetto, non è necessario specificare manualmente la loro versione: queste verranno automaticamente allineate alle versioni indicate nel POM padre.
- **Configurazione dei Plugin:** `spring-boot-starter-parent` fornisce configurazioni di default per i plugin Maven, come il plugin per la creazione del pacchetto JAR. Questo semplifica il processo di building e packaging delle applicazioni Spring Boot.

- **Proprietà Comuni:** Il POM padre può definire proprietà che possono essere riutilizzate nel POM figlio. Questo può includere configurazioni come la versione Java target. Ad esempio:

```
<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
    <java.version>11</java.version>
    <lombok.version>1.18.24</lombok.version>
</properties>
```

Qua specifichiamo la versione di Java e di Lombok che verranno utilizzate nel resto del POM

- **Standardizzazione:** Usando un POM padre come spring-boot-starter-parent, ci assicuriamo che il nostro progetto aderisca alle best practices e alle convenzioni suggerite dal team di Spring Boot.
- **Facilità di Aggiornamento:** Quando si vuole aggiornare la versione di Spring Boot che si sta utilizzando, è possibile semplicemente cambiare la versione nel <parent>, e tutte le dipendenze e configurazioni collegate saranno aggiornate di conseguenza.

In sintesi, l'uso di spring-boot-starter-parent nel pom.xml fornisce un set di configurazioni predefinite che rendono più semplice l'inizializzazione, la configurazione e la gestione delle applicazioni Spring Boot.

Al di sotto del tag parent all'interno dei tag <**dependencies**> </**dependencies**> troviamo tutte le dipendenze che verranno utilizzate all'interno del nostro progetto. Tra le varie possiamo distinguere le **Starter Dependencies** di Spring Boot. Esse rappresentano uno strumento molto potente di Spring Boot che ci consente tramite un'unica dipendenza di andare ad importare un gruppo di dipendenze correlate che sono comuni per una particolare funzionalità o tecnologia.

Esempi comuni di Starter Dependencies:

- **Spring Boot Starter Web:** Include tutto ciò che ti serve per creare una web application con Spring MVC e Tomcat come server incorporato.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
```

```
<artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

- **Spring Boot Starter Data JPA:** Facilita l'uso di Spring Data JPA con Hibernate come provider JPA.

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

Dopo aver esaminato il pom.xml, continuiamo ad esplorare il resto del codice. Non tratterò nuovamente le JSP in dettaglio, poiché presentano poche variazioni rispetto allo sprint precedente. Tuttavia, è essenziale sottolineare le differenze nell'approccio alle chiamate.

Nei contesti basati su Servlet, i metodi principali utilizzati per gestire le richieste sono **doGet** e **doPost**. In contrasto, Spring offre una flessibilità maggiore, consentendo di mappare **specifici metodi a specifici URL** all'interno di un controller.

Consideriamo UserController come esemplificativo:

```
@Controller
@RequestMapping("/user")
public class UserController {

    @Autowired
    private UserService service;

    @PostMapping("/login")
    public String login(HttpServletRequest request,
    @RequestParam(value = "username") String username,
    @RequestParam(value = "password") String password) {
        return service.login(request, username, password);
    }

    @GetMapping("/getall")
    public String getAll(HttpServletRequest request) {
        setAll(request);
        return "users";
    }
}
```

Osserviamo le annotazioni principali:

- **@RequestMapping("/user")**: Specifica che tutti i metodi presenti in UserController avranno un percorso base di "/user".
- **@PostMapping("/login")**: Indica che il metodo login risponde all'URL "/login" e accetta richieste POST.
- **@GetMapping("/getall")**: Analogamente, questa annotazione indica che il metodo getAll risponde all'URL "/getall" e accetta richieste GET.

Pertanto, per invocare il metodo login da una JSP, il modulo apparirà come:

```
<form class="login" action="/user/login" method="post">
```

In sostanza, Spring semplifica e rende più intuitiva la definizione e la gestione delle rotte rispetto al modello tradizionale delle servlet.

La **Dependency Injection** (DI) è uno dei principi fondamentali dietro la progettazione e la struttura di Spring, e segna una chiara differenza rispetto alle tecniche di gestione delle dipendenze utilizzate in applicazioni basate su servlet semplici.

In un'applicazione non-Spring, come quella basata su servlet, quando un componente ha bisogno di accedere a un servizio o a un'altra dipendenza, solitamente lo crea direttamente utilizzando il costruttore della classe corrispondente, come mostrato nell'esempio del UserServlet.

```
@WebServlet("/UserServlet")
public class UserServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;

    private UserService service;

    @Override
    public void init() {
        this.service = new UserService();
    }
}
```

In questo esempio, ogni volta che UserServlet viene inizializzato, viene creata una **nuova istanza** di UserService. Questo approccio, anche se funzionale, ha delle limitazioni in termini di flessibilità e modularità.

Spring, con il suo meccanismo di Dependency Injection, introduce un cambiamento radicale a questo paradigma. Invece di creare le dipendenze direttamente, un componente Spring dichiara le sue dipendenze e **si affida al framework** per fornirle quando necessario.

```

@Controller
@RequestMapping("/user")
public class UserController {

    @Autowired
    private UserService service;
}

```

Nell'esempio di UserController, l'annotazione **@Autowired** indica a Spring di "iniettare" un'istanza di UserService in quella variabile. Spring si occupa di cercare un bean adatto (un'istanza di UserService gestita dal **IOC Container**) e di fornirlo al controller. In questo modo ogni classe che avrà bisogno di utilizzare UserService andrà a richiamare la stessa istanza senza la necessità di dover istanziare ogni volta un nuovo oggetto.

E' importante ricordare che per fare sì che una classe possa essere riconosciuta e gestita come un **bean** da Spring, essa deve essere annotata con specifiche annotazioni che ne indicano la funzione e il ruolo all'interno dell'applicazione. Una di queste è **@Service**.

```

@Service
public class UserService extends AbstractService<User, UserDTO> {}

```

L'annotazione **@Service** è una specializzazione di **@Component** (come anche **@Controller**, **@RestController** e **@Repository**), ed è tipicamente utilizzata per classi che rappresentano servizi nella logica di business dell'applicazione. Marcando una classe con **@Service**, stiamo comunicando a Spring di riconoscerla come un bean e di **registrarlo** nel suo IoC container. Da quel momento in poi, Spring sarà in grado di **iniettare** istanze di quella classe dove necessario, garantendo anche la corretta gestione del suo ciclo di vita.

Stesso discorso potremo fare per le classi UserConverter e UserRepository all'interno della classe UserService, rispettivamente annotate con **@Component** e **@Repository**.

```

public class UserService extends AbstractService<User, UserDTO> {

    @Autowired
    private UserConverter converter;
    @Autowired
    private UserRepository repository;

}

```

Andiamo proprio ad approfondire queste due classi per vedere le differenze rispetto allo sprint precedente.

Converter

All'interno del progetto, anziché soffermarci sulla classe UserConverter, che è stata leggermente rivista dall'ultimo sprint introducendo l'utilizzo del **pattern builder** per la creazione degli oggetti, è interessante analizzare in profondità la classe AbstractConverter. Quest'ultima adotta le **Java Stream** per effettuare conversioni tra liste di DTO e Entity.

```
public abstract class AbstractConverter<Entity, DTO> implements  
Converter<Entity, DTO> {  
  
    public List<Entity> toEntityList(List<DTO> listDTO) {  
        return listDTO.stream()  
            .map(this::toEntity)  
            .collect(Collectors.toList());  
    }  
  
    public List<DTO> toDTOList(List<Entity> listEntity) {  
        return listEntity.stream()  
            .map(this::toDTO)  
            .collect(Collectors.toList());  
    }  
}
```

Entriamo nel dettaglio dei metodi toEntityList e toDTOList:

- **Operatore map:** Questo operatore **trasforma** ogni elemento di uno Stream. Nella nostra implementazione, applichiamo map per convertire ciascun elemento della lista. Per esempio, se stiamo lavorando con una lista di DTO, ogni DTO viene trasformato in una Entity mediante il metodo toEntity. Analogamente, se abbiamo una lista di Entity, ciascuna Entity viene convertita in un DTO con il metodo toDTO. Di fatto, con l'operazione map, **otteniamo uno Stream di Entity o DTO**, a seconda del contesto.
- **Operatore collect:** Dopo aver usato map per trasformare gli elementi, dobbiamo **racchiudere** questi elementi trasformati in una struttura dati **concreta**. L'operatore collect ci consente di fare esattamente questo. Nel nostro caso, raccogliamo gli elementi trasformati in una List.

Un'altra cosa importante da segnalare è l'utilizzo dei **Method Reference** che sostanzialmente rappresentano un metodo più elegante e conciso per richiamare delle funzioni. In questo caso sono stati utilizzati all'interno dell'operatore map:

```
map(this::toEntity)
```

Sostanzialmente la sintassi è questa **ClassName::methodName**, ovvero nome della classe e metodo della classe da richiamare. In questo caso siccome i metodi fanno parte della classe stessa in cui stiamo utilizzando il method reference utilizzeremo la keyword **this**.

Se ad esempio avessi una lista di Stringhe scritte in minuscolo e volessi ottenere una lista delle stesse Stringhe ma convertite in maiuscolo dovrei richiamare il metodo `toUpperCase` della classe `String` in questo modo:

```
List<String> words = List.of("java", "stream", "method",
    "reference");

List<String> upperCaseWords = words.stream()
    .map(String::toUpperCase)
    .collect(Collectors.toList());
```

In questo modo la lista risultante **upperCaseWords** sarà una lista delle stringhe contenute nella lista **words** convertite in maiuscolo.

Repository

Come abbiamo analizzato nella parte teorica le repository rappresentano essenzialmente i DAO ma vengono gestite da Spring. Prendiamo come esempio l'interfaccia `UserRepository`:

```
@Repository
@Transactional
public interface UserRepository extends JpaRepository<User, Long> {
    User findByUsernameAndPassword(String username, String
password);
}
```

Andiamo ad analizzare i componenti principali di questa classe:

- **@Repository**: Questa è una delle annotazioni Spring Stereotype (le altre includono `@Component`, `@Service` e `@Controller`). L'annotazione `@Repository` serve a indicare che la classe annotata è una repository, una classe che fungerà da ponte tra l'applicazione e il database. Il suo uso facilita anche la gestione delle eccezioni, convertendo le eccezioni specifiche del database in eccezioni `Spring DataAccessException` non legate al database.
- **@Transactional**: Questa annotazione è utilizzata per indicare che qualsiasi metodo eseguito all'interno della classe (o sull'interfaccia, in questo caso)

dovrebbe essere eseguito all'interno di una **transazione**. Le transazioni assicurano che le operazioni sul database siano completate con successo o, in caso di errore, **che nessuna delle operazioni venga eseguita** (tutto o niente). Se questa annotazione viene utilizzata su un'interfaccia o su una classe (e non su un metodo specifico), allora tutti i metodi pubblici in quella classe o interfaccia saranno eseguiti all'interno di una transazione.

- **JpaRepository<User, Long>**: Spring Data JPA offre vari repository predefiniti, e JpaRepository è uno di questi. Estendendo JpaRepository, la nostra interfaccia UserRepository erediterà automaticamente metodi per operazioni CRUD (Create, Read, Update, Delete) sul database. I due parametri generici passati (User e Long) rappresentano rispettivamente la classe dell'entità e il tipo del suo ID. In questo caso, abbiamo un'entità User con un ID di tipo Long.
- **findByUsernameAndPassword(String username, String password)**: Questo è un metodo di **query derivata**. Spring Data JPA permette di definire query solo attraverso la firma dei metodi, senza scrivere l'implementazione o la query SQL/JPQL effettiva. Il nome del metodo descrive la sua funzione. In questo caso, il metodo sarà utilizzato per trovare un utente in base a username e password. Spring Data JPA genererà automaticamente l'implementazione di questo metodo basandosi sul suo nome. In pratica, eseguirà una query che selezionerà un utente dal database dove il campo username corrisponde al valore passato come username e il campo password corrisponde al valore passato come password.

In sintesi, UserRepository è un'interfaccia che funge da punto di accesso al database per le operazioni riguardanti gli utenti. Spring Data JPA consente una riduzione drastica della verbosità, evitando di dover scrivere l'implementazione delle operazioni CRUD e di altre query comuni, come quella definita nel metodo findByUsernameAndPassword.

Un'altra novità introdotta durante questo sprint riguarda la gestione delle eccezioni. In particolare andremo ad analizzare la classe **GeneralExceptionHandler**.

Questa classe rappresenta un meccanismo centrale di gestione delle eccezioni in un'applicazione Spring MVC. Analizziamola pezzo per pezzo:

- **@ControllerAdvice**: Questa annotazione indica che la classe è un consiglio globale per i controller all'interno dell'applicazione. Ciò significa che qualsiasi eccezione sollevata dai metodi dei controller può essere gestita in questa classe. @ControllerAdvice è particolarmente utile quando si vuole avere un singolo punto per gestire eccezioni specifiche o tutte le eccezioni che possono verificarsi nei tuoi controller.

- **@ExceptionHandler(RuntimeException.class)**: Questa annotazione indica che il metodo sottostante (in questo caso handleRuntimeException) è designato per gestire le eccezioni del tipo specificato, in questo caso RuntimeException e le sue sottoclassi. Se un'eccezione di tipo RuntimeException (o una sottoclasse di essa) viene sollevata da un qualsiasi controller, questo metodo sarà invocato.
- **handleRuntimeException(RuntimeException ex, HttpServletRequest request)**: Questo è il metodo che gestisce effettivamente l'eccezione. Accetta due parametri:
 - **RuntimeException ex**: L'oggetto eccezione che è stato sollevato. Questo permette di accedere a dettagli sull'eccezione come il suo messaggio (ex.getMessage()).
 - **HttpServletRequest request**: Questo rappresenta la richiesta HTTP corrente. Viene utilizzato nel metodo per impostare un attributo "**errorMessage**", che contiene il messaggio dell'eccezione. Questo attributo può quindi essere utilizzato nella vista per mostrare un messaggio di errore all'utente.
- **return "errorPage"**: Dopo aver gestito l'eccezione, il metodo reindirizza l'utente a una vista chiamata "errorPage". Questa rappresenta una pagina di errore generico in cui si può mostrare il messaggio di errore o fornire ulteriori dettagli sull'errore.

In sintesi, questa classe fornisce una **gestione centralizzata** delle RuntimeException all'interno di un'applicazione Spring MVC. Se si verificano eccezioni durante l'elaborazione di una richiesta nel controller, piuttosto che rompere brutalmente l'interazione con l'utente o mostrare una pagina di errore inaspettata, viene mostrata una pagina di errore controllata e potenzialmente più informativa all'utente.

L'ultima differenza, ma non la meno importante, che andremo ad analizzare riguarda la creazione del database. Mentre precedentemente eravamo abituati a creare le tabelle e le relazioni direttamente su MySQL, Spring e Hibernate ci permettono di andare a realizzare il nostro Database direttamente con delle classi Java.

Vediamo come fare prendendo come esempio il Model User:

```
@Entity
public class User {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(unique = true)
    private String username;
```

```
    private String password;  
  
    private Usertype usertype;  
  
}
```

Ecco una spiegazione dettagliata di ciascun componente:

- **@Entity**: Questa annotazione proviene da JPA e indica che la classe User è un'entità, ovvero una classe che può essere mappata su una tabella nel database.
- **@Id**: Questa annotazione JPA indica che il campo sottostante (id) è la chiave primaria dell'entità.
- **@GeneratedValue(strategy = GenerationType.IDENTITY)**: Anche questa è un'annotazione JPA che indica che il valore per il campo id verrà generato automaticamente. Il tipo di strategia IDENTITY significa che l'id sarà autoincrementale .
- **@Column(unique = true)**: Questa annotazione JPA indica che il campo username sarà mappato su una colonna nel database e che questa colonna avrà una **restrizione di unicità**, assicurando che non possano esserci due utenti con lo stesso nome utente.
- **id, username, password, usertype**: Questi sono i campi dell'entità User. Rappresentano le informazioni essenziali associate a un utente: un identificatore univoco, un nome utente, una password e un tipo di utente. Rappresentano le colonne che verranno create nella tabella User sul db.