

|  |
| --- |
|  |

|  |
| --- |
|  |

# 

# 

# 

# 

# 

# 

# 

Corso Spring Boot: Jpa Hibernate Mapstruct

# 

Dispense:

Giuseppe ing. Grosso

c

. 1

Introduzione. 3

Sintesi argomenti trattati: 3

Requisiti ingresso. 3

Startup ambiente. 4

Utilizzo di un IDE. 5

Installazione tramite intellij comunity Edition. 5

Configurazione application.yaml. 7

Parametro spring.jpa.hibernate.ddl-auto: 7

Quale opzione usare?. 7

Installazione SQLiteStudio. 8

Installazione MySQL. 8

Schema Persistenza: JPA. 9

Mapping in java.. 10

Relazione One-To-One. 11

Relazione One-To-Many e Many-To-One. 12

Relazione Many-To-Many. 13

Spring data jpa. 15

Connessione ad un database fisico. 15

Mapping tabelle con classi java. 17

JPA Creazioni Query. 20

Inferred query. 20

Custom JPQL. 21

Native Queries con @Query. 21

Transazioni. 22

Data Transfert Object Pattern: DTO. 23

Perchè usare DTO.. 25

Svantaggi di DTO.. 26

MapStruct. 26

Casi di uso: publication. 26

Auth: esercitazione. 31

chinookjpa: esercitazione. 31

# Introduzione

L’obiettivo del corso è quello di dare allo studente elementi avanzati di programmazione nel linguaggio java ed in particolare con l’utilizzo di jpa, Hibernate, MapStruct

## Sintesi argomenti trattati:

- startup ambiente

- aggiunta librerie per lo sviluppo

- Spring JPA

- Connessione ad un database reale

- Mappare classi in tabelle

- Ciclo di vita delle entità di hibernate

- Queries

- Paginazione e sorting

- Relazioni 1:1, 1:M, M:M ed annotazioni per le tabelle

- transazioni

## Requisiti ingresso

è richiesta la conoscenza java avanzata e l’utilizzo dell’ambiente base di spring boot.

Verranno utilizzate le seguenti tecnologie:

- Spring boot 2.4.0

- JDK 8+ o openJDK 8+

- Maven 3+

# Startup ambiente.

Per lo start up dell’ambiente fare riferimento al sito ufficiale:<https://spring.io/projects/spring-boot>.

Oppure meglio da spring initializer

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

Dopo aver impostato i dati sopra riportati:

· progetto

· linguaggio

· versione spring

· package

· versione java

· dependencies: web, jpa, hibernate, lombok, mysql/sqlite

si effettua il download del jar e si importa es. in eclipse/intellij come existing maven project, oppure in intellij o visual studio code.

Tramite il comando :

./mvnw clean spring-boot:run

E’ possibile eseguire il progetto da linea di comando o nel terminale di visual studio code.

Oppure da eclipse dalla classe del main eseguire come se fosse un programma stand-alone.

# Utilizzo di un IDE.

E’ possibile utilizzare vari IDE per lo sviluppo dei servizi del catalogo API, tra i più importanti:

- eclipse

- visual Studio code

- intellij

Tutti gli ide sopra indicati sono utilizzabili per lo sviluppo sia frontend che backend. Tuttovia si raccomanda di utilizza indifferentemente sia visual studio code che intellij. In entrambi i casi è possibile scrivere codice sia in java che javascript che typescript.

Nel corso verranno utilizzati tutti e tre gli ide di sviluppo, si faccia riferimento ai siti dei produttori.

Es.<https://code.visualstudio.com/docs/java/> per vs code.

# Installazione tramite intellij comunity Edition.

Da inellij new Project, selezionare Spring assistant:

Impostati i parametri come immagine sopra, mentre per le librerie:

Web:

# Configurazione application.yaml.

*# configurazione del server*

server:

port: 8092

servlet:

contextPath: /jpademo

spring:

jpa:

database-platform: it.plansoft.demojpa.sql3.dialect.SQLiteDialect

hibernate.ddl-auto: update

show-sql: true

properties:

hibernate:

format\_sql: true

datasource:

url: jdbc:sqlite:src/main/resources/jpademo.db

driver-class-name: org.sqlite.JDBC

*# username: admin*

*# password: admin*

*# impostazioni per hibernate*

hibernate:

hbm2ddl:

auto: none

show-sql: true

## Parametro spring.jpa.hibernate.ddl-auto:

https://docs.spring.io/spring-boot/docs/1.1.0.M1/reference/html/howto-database-initialization.html

· none: hibernate non effettua nessuna operazione sullo schema di database (per produzione)

· create-only: creazione dello schema del database a partire dal modello

· drop: si droppa il database usando il modello entity come reference per le ddl di drop.

· create: si droppa e ricrea il database dopo aver usato il modello di riferimento

· create-drop : **default** per database in memoria e dice di creare e droppare quando si fa lo start e stop dell’applicazione

· validate: validazione dello schema in base al modello

· update: aggiornamento dello schema comparando lo schema con il database e preparando lo script di migrazione.

## Quale opzione usare?

None se il database viene generato manualmente o in produzione.

Se si vuole usare uno schema migrato e si vuole creare lo schema del database si può inserire create-only.

Per i test utilizzare create o create-drop.

Solitamente si utilizza **update** che permette di creare lo script dati migrazione.

# Installazione SQLiteStudio.

Per installare sqlite studio occorre andare sul sito ufficiale e scaricare la versione opportuna, es.

Dal sito<https://sqlitestudio.pl/> effettuare il download in maniera tale da poter utilizzare l’editor per poter generare la banca dati ed eventualmente inserire tabelle, chiavi relazioni etc.

Una esercitazioni sarà di mappare le tabelle di un database già fatto es.<https://www.sqlitetutorial.net/sqlite-sample-database/>

Esercitazione mapping.

# Installazione MySQL.

Procedere all’installazione dal sito ufficiale<https://www.mysql.com/it/downloads/>, la versione utilizzata nel corso sarà la seguente v 8.0.23 recuperare dal sito:<https://dev.mysql.com/downloads/installer/>

Tale installazione permette di installare il motore MySql ed il client MySQL workbench 8.0 al termine dell’installazione lanciare workbench e creare lo schema per ospitare i dati dell’esercitazione.

Es database chinook

# Schema Persistenza: JPA

# Mapping in java

Hibernate è un framework di persistenza per *plain java object* (o **pojo**) e quindi le tipologie di associazioni che sono prese in considerazione sono prevalentemente quelle unidirezionali, tipiche di java. Tali associazioni sono tipicamente espresse attraverso riferimenti o collezioni in funzione della molteplicità dell’associazione. La specifica JPA prevede le seguenti tipologie di associazioni:

|  |  |
| --- | --- |
| Relazione | Descrizione |
| 1:1 | Ogni istanza di una entità è legata ad una singola istanza di un’altra entità. Sono espresse attraverso l’annotazione javax.persistence.OneToOne posizionata in corrispondenza della proprietà o campo dell’oggetto persistito.  ESEMPIO: relazione tra una persona ed il proprio passaporto. |
| 1:M | Un’istanza di una entità può essere correlata a più istanze di altre entità. Sono espresse utilizzando l’annotazione javax.persistence.OneToMany posizionata in corrispondenza della proprietà o campo dell’oggetto persistito.  **ESEMPIO**: relazione esistente tra un ordine di acquisto e gli elementi che lo compongono. |
| M:1 | *Più istanze di una entità possono essere correlati a una singola istanza dell’altra entità.* Questa molteplicità è l’opposto di una relazione one-to-many. Sono espresse utilizzando l’annotazione javax.persistence.ManyToOne posizionata in corrispondenza della proprietà o campo dell’oggetto persistito.  **ESEMPIO**: relazione esistente tra una persona ed uno dei suoi genitori, che ovviamente può avere più figli. |
| M:M | Le istanze di una entità possono essere correlate a più istanze di ogni altro entità. Sono espresse attraverso l’annotazione javax.persistence.ManyToMany posizionata in corrispondenza della proprietà o campo dell’oggetto persistito. |

Le diverse associazioni sono caratterizzate da informazioni che sono caratteristiche della specifica relazione, ma tutte hanno in comune la modalità con cui sono recuperate dal database (fetching). L’enum [FetchType](http://docs.oracle.com/javaee/5/api/javax/persistence/FetchType.html) elenca le due strategie possibili di fetching, che sono EAGER o LAZY.

La prima indica al runtime di persistenza che la relazione deve essere recuperata immediatamente insieme all’oggetto che la utilizza.

La seconda è invece un suggerimento per cui la specifica implementazione può decidere di seguirlo, recuperando la relazione solamente quando viene acceduta, o meno, di fatto comportandosi come nel caso EAGER.

## Relazione One-To-One

Consideriamo la relazione esistente tra uno oggetto A ed un oggetto B, rappresentato dal modello dati mostrato in figura, dove ad A è associato univocamente B.

Utilizzando le annotation il mapping tra la classe A ed B sarà del tipo:

**@Entity**

**@Table( name = "A" )**

**public class A {**

**..**

**// Altri getter e setter**

**@OneToOne( fetch= FetchType.EAGER )**

**@JoinColumn( name="B" )**

**public b getB() {**

**return b;**

**}**

**public void setB(B b) {**

**this.b = b;**

**}**

**}**

Si noti che nella relazione è stato specificata EAGER come strategia di fetching ma, più importante, è stata utilizzata l’annotazione @JoinColumn per indicare la colonna della tabella A che contiene il valore della primary key referenziata nella tabella B.

Sebbene non necessario è anche possibile mappare la *relazione inversa* tra B e A cui è associato, senza necessità di modificare il database ma semplicemente utilizzando la stessa annotation sulla classe B nel modo seguente:

**@Entity**

**@Table( name = "B" )**

**public class B {**

**// Altri getter e setter**

**@OneToOne( mappedBy = "b" )**

**public A getA() {**

**return a;**

**}**

**public void setA(A a) {**

**this.a = a;**

**}**

**}**

In questo caso è sufficiente indicare la proprietà della classe A che mappa la relazione ed Hibernate provvederà ad invertirla per eseguire la join necessaria a recuperala.

## Relazione One-To-Many e Many-To-One

Consideriamo la relazione esistente tra le tracce e i media types e i generi, mostrata nel modello dati seguente, dove una traccia può appartenere ad uno ed un solo tipo di media e un solo genere.

Mentre una un tipo di media ed un genere può corrispondere più di una traccia audio.

Vediamo le annotazioni jpa per il mapping ManyToOne:

@Data

@AllArgsConstructor

@ToString

@Entity

@Table(name = "tracks")

public class Tracks extends BaseId<Long> {

public Tracks() {

}

…

@ManyToOne

@JoinColumn( name="media\_type\_id" )

private MediaTypes mediaTypes;

@ManyToOne

@JoinColumn( name="genre\_id" )

private Genres genres;

e l’annotazione oneToMany rispettive tabelle genres:

@Table(name = "genres")

public class Genres extends BaseId<Long> {

@Id

@Column(name = "genre\_id")

@GeneratedValue(strategy = GenerationType.*IDENTITY*)

private Long id;

@Column(name = "name")

private String name;

@OneToMany(mappedBy = "genres")

@JsonIgnore

private Set<Tracks> tracks;

tabella mediaTypes

@Data

@AllArgsConstructor

@Entity

@Table(name = "media\_types")

public class MediaTypes extends BaseId<Long> {

@Id

@Column(name = "media\_type\_id")

@GeneratedValue(strategy = GenerationType.*IDENTITY*)

private Long id;

@Column(name = "name")

private String name;

@OneToMany(mappedBy = "mediaTypes")

@JsonIgnore

private Set<Tracks> tracks;

Anche in questo caso è necessario, quindi, specificare la colonna di join nella relazione @ManyToOne presente nella classe Track e la proprietà di mapping nella @OneToMany nella classe Genres e MediaTypes.

## Relazione Many-To-Many

Questa volta prendiamo in consideriamo la relazione esistente tra la playlist e le tracce audio. In questo caso una playlist può avere più tracce audio ed analogamente ad una traccia audio può essere inclusa in più playlist. Una relazione di questo tipo può essere catturata esclusivamente utilizzando una tabella di relazione tra le due entità, come mostrato nel diagramma seguente, dove la tabella playlist\_track permette il tipo di relazione M:M:

Supponiamo ora di voler mappare nella classe playlist tutte le tracce audio. Per farlo utilizziamo l’annotazione @ManyToMany nel modo seguente:

Si noti che la tabella playlist\_track non ha un corrispondente model nel progetto ma è semplicemente referenziata dall’annotazione @JoinTable che serve a dichiarare il mapping tra le entità playlist e track.

@Id

@Column(name = "playlist\_id")

@GeneratedValue(strategy = GenerationType.*IDENTITY*)

private Long id;

@Column(name = "name")

private String name;

@ManyToMany

@JoinTable(

name="playlist\_track",

joinColumns=@JoinColumn(name="playlist\_id", referencedColumnName="playlist\_id"),

inverseJoinColumns=@JoinColumn(name="track\_id", referencedColumnName="track\_id"))

private Set<Tracks> tracks;

Vedi esercitazione chinookJpa.

# Spring data jpa.

Il mapping delle tabelle all’interno degli oggetti java viene effettuato nelle classi model, queste hanno una struttura così fatta:

@Entity

@Table(name = "albums")

public class Albums extends BaseId<Long>

{

@Id

@Column(name = "album\_id")

@GeneratedValue(strategy = GenerationType.*IDENTITY*)

private Long id;

@Column(name = "title")

private String title;

@ManyToOne

@JoinColumn(name = "artist\_id")

private Artists artist;

Come vediamo la classe viene annotata con due attributi:

@Entity che dice al framework che l’oggetto mappa una tabella

@Table dice che al framework che il nome della tabella è chiamata albums

@uniqueConstraints è possibile definire una constraint (vincolo) su una colonna specifica es. title inserendo il nome della constraint e il nome della colonna:

@Table(name = "albums",

uniqueConstraints = {

@UniqueConstraint(name = "albums\_email\_unique", columnNames = "title")

})

# Connessione ad un database fisico.

La connessione al database fisico viene effettuata tramite i file di configurazione nominato application.properties.

Per maggior chiarezza vengono attivati dei profili diversi es:

· postgres

· mysql

· sqlite

In maniera tale che si vada a vedere come effettuare le configurazioni specifiche per i database utilizzati.

Si rimanda alla documentazione dei siti ufficiali per i tre database:

<https://www.postgresql.org/docs/12/server-start.html>

<https://dev.mysql.com/downloads/mysql/>

<https://www.sqlite.org/download.html>

Una volta installati occorre es. per postgres far partire il server

es.

bin\initdb.exe -D C:\Users\ggrosso\GiuseppeGrossoFolder\download\temp\pgdata

Effettua l’inizializzazione della cartella con i dati che conterrà la banca dati successivamente.

bin\pg\_ctl.exe -D “C:\Users\ggrosso\GiuseppeGrossoFolder\download\temp\pgdata” start

Successivamente da pgAdmin4, che permette di effettuare la connessione al database.

Per quanto riguarda invece MySQL questo una volta installato permette tramite workbench di poter creare i vari schema e controllare lo stato di tabelle, sequence etc.

# Mapping tabelle con classi java.

Di seguito andiamo nel dettaglio a vedere il mapping delle tabelle in classi java dettagliando nel model tutte le annotazioni tipiche da inserire.

Es. entity e mapping:

Album

@Entity

@Table(name = "albums",

uniqueConstraints = {

@UniqueConstraint(name = "albums\_title\_unique", columnNames = "title")

})

public class Albums extends BaseId<Long>

{

@Id

@Column(name = "album\_id")

@GeneratedValue(strategy = GenerationType.*IDENTITY*)

private Long id;

@Column(name = "title")

private String title;

@ManyToOne

@JoinColumn(name = "artist\_id")

private Artists artist;

public Long getId() {

return id;

}

public void setId(Long id) {

this.id = id;

}

public String getTitle() {

return title;

}

public void setTitle(String title) {

this.title = title;

}

public Artists getArtist() {

return artist;

}

public void setArtist(Artists artist) {

this.artist = artist;

}

public Albums()

{

}

public Albums(Long id, String title, Artists artist) {

this.id = id;

this.title = title;

this.artist = artist;

}

}

Artist:

@Data

@AllArgsConstructor

@ToString

@Entity(name = "artists")

@Table(name = "artists")

public class Artists extends BaseId<Long> {

@Id

@Column(name = "artist\_id")

@GeneratedValue(strategy = GenerationType.*IDENTITY*)

private Long id;

public Long getId() {

return id;

}

public void setId(Long id) {

this.id = id;

}

@Column(name = "Name")

private String name;

@OneToMany(mappedBy = "artist")

@JsonIgnore

private Set<Albums> albums;

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public Set<Albums> getAlbums() {

return albums;

}

public void setAlbums(Set<Albums> albums) {

this.albums = albums;

}

public Artists() {

}

public Artists(Long id, String name) {

this.id = id;

this.name = name;

}

}

@Data

@AllArgsConstructor

@ToString

@Entity

@Table(name = "playlists")

public class Playlist extends BaseId<Long> {

@Id

@Column(name = "playlist\_id")

@GeneratedValue(strategy = GenerationType.*IDENTITY*)

private Long id;

@Column(name = "name")

private String name;

@ManyToMany

@JoinTable(

name="playlist\_track",

joinColumns=@JoinColumn(name="playlist\_id", referencedColumnName="playlist\_id"),

inverseJoinColumns=@JoinColumn(name="track\_id", referencedColumnName="track\_id"))

private Set<Tracks> tracks;

public Playlist() {

}

public Long getId() {

return id;

}

public void setId(Long id) {

this.id = id;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public Set<Tracks> getTracks() {

return tracks;

}

public void setTracks(Set<Tracks> tracks) {

this.tracks = tracks;

}

}

## JPA Creazioni Query.

Abbiamo vari tipi di opzioni su come creare le query:

· inferred Queries

· Custom JPQL con @Query es.

· Native Queries con @Query

Set<Albums> findByTitle(String title);

@Query("select a from Album a where a.title = :title")

Set<Albums> findByTitleCustomQuery(@Param("title") String title);

@Query(

value = "select \* from album as a where a.title = :title",

nativeQuery = true)

Set<Albums> findByTitleNativeQuery(@Param("title") String title);

### Inferred query.

Non abbiamo bisogno di dire a spring data cosa fare, automaticamente tramite il nome del metodo.

Quello che è conveniente è che spring data automaticamente effettua il check delle query, quindi della validità delle query allo startup dell’ambiente.

Pertanto se esempio chiamiamo il metodo findByTitle e nell’entity non vi è il campo title avremo una eccezione di questo tipo:

Invocation of init method failed; nested exception is java.lang.IllegalArgumentException: Failed to create query for method public abstract java.util.Set it.plansoft.publication.repository.PubblicationRepository.findByAuthorsCognomes(java.lang.String)! No property cognomes found for type Author! Did you mean 'cognome'? Traversed path: Pubblication.authors.

E’ possibile pertanto effettuare una annotazione del tipo @Query e provvedere una custom JPQL query.

## Custom JPQL.

Se la query è complessa è conveniente utilizzare una custom JPQL query:

es.

@Query("select p from Publication p join Author a where a.cognome = :name and p.titolo = :titolo")

Set<Pubblication> findByAuthorsCustomCognome(@Param("cognome") String cognome, @Param("titolo") String titolo);

Similarmente alle query inferred dobbiamo fare un check di validita per le JPQL. In caso di errore manda un errore di invalid query:

es

org.hibernate.hql.internal.ast.QuerySyntaxException:

unexpected token: xxx near line 1, column 64 [select p from ...]

Le custom query, tuttovia, possono essere complicate se si vanno a cercare più parametri.

Possiamo introdurre join su altre tabelle e può ritornare oggetti dto complessi.

## Native Queries con @Query.

Un’altra alternativa sono le query native:

**@Query(**

**value = "select \* from album as a where a.title = :title",**

**nativeQuery = true)**

**Set<Albums> findByTitleNativeQuery(@Param("title") String title);**

Invece di specificare un JPQL query, che è un’astrazione andiamo a scrivere esattamente la query in maniera nativa. Attenzione questa query potrebbe utilizzare nella query un dialetto particolare, è importante notare che jpa valida le query in fase di startup dell’applicazione. Pertanto se si usano specifici dialetti SQL, non è possibile successivamente effettuare migrazioni su altri database.

Pertanto le query native sono i primi candidati per le query di integrazione.

## Transazioni.

<https://docs.spring.io/spring-framework/docs/4.2.x/spring-framework-reference/html/transaction.html>

Es.

@Transactional

public Albums insertAlbumAndArtist(Albums albums) {

if (albums != null) {

if (albums.getArtist() != null) {

Artists a = albums.getArtist();

// lettura dell'artista se non esiste lo inserisco

artistService.findByName(a.getName()).map(item -> {

return null;

}).orElseGet(() -> {

return artistService.save(a);

});

}

if (true) throw new RuntimeException("Errore Finto");

albumService.save(albums);

}

return albums;

}

Es. no transaction

public Albums insertAlbumAndArtist(Albums albums) {

if (albums != null) {

if (albums.getArtist() != null) {

Artists a = albums.getArtist();

// lettura dell'artista se non esiste lo inserisco

artistService.findByName(a.getName()).map(item -> {

return null;

}).orElseGet(() -> {

return artistService.save(a);

});

}

if (true) throw new RuntimeException("Errore Finto");

albumService.save(albums);

}

return albums;

}

Effettuando i due esempi di codice si vede nel primo caso che i dati non vengono inseriti all’interno del database, mentre nel secondo caso abbiamo dei dati sporchi ovvero viene inserito il primo record mentre il secondo record non viene inserito.

[Request processing failed; nested exception is java.lang.RuntimeException: Errore Finto] with root cause

java.lang.RuntimeException: Errore Finto

# Data Transfert Object Pattern: DTO.

DTO è un altro design pattern ben noto e molto utilizzato. Il concetto è quello di introdurre una o più classi per modellare una struttura dati adatta per un caso d’uso specifico al fine di trasferire dati tra gli strati di un’applicazione, in particolare per lo strato di presentation.

Transfer Object è una semplice classe POJO con metodi getter / setter e non ha nessun comportamento ma ha lo scopo di contenitore di dati. E’ poi lo strato di business che prelevando i dati dal DB si occupa di popolare questo tipo di classi. Per i client, un DTO è un oggetto in sola lettura, può comunque essere usato per trasferire dati dal client verso il server, nel qual caso il client creerà un proprio oggetto DTO da passare al server per, ad esempio, aggiornare dei dati nel DB.

Il pattern DTO viene utilizzato es. all’interno di una applicazione complessa, che molto spesso è su larga scala, multi azienda e che richiede di essere scalabile.

E’ necessario che vi sia una buona architettura di base, per questo tipo di applicazioni vi sono dei pattern che nel tempo hanno risolto alcune criticità, in particolare il data transfer object design pattern, permette di risolvere il problema di aggregare ed incapsulare i dati per il trasferimento al layer di presentazione.

La seguente immagine rappresenta ad alto livello quello che viene effettuato:

Es. immaginiamo di avere una entity User e una Location con un mapping oneToMany, ovvero in una location possono essere ospitati molti utenti ed un mapping inverso sull’user manyToOne

Es.

User:

@Entity

Public Class User {

@Id

@GeneratedValue…

Private Long id;

private String username;

..

@ManyToOne(fetch = FetchType.LAZY, optional = false)

@JoinColumn(name = “location\_id”)

Private Location location;

// getter e setter

Location:

@Entity

Public Class Location{

@Id

@GeneratedValue…

Private Long id;

private Double lat;

private Double lng;

..

@OneToMany(mappedBy = “location”)

Private Set<User> users;

// getter e setter

Per il trasferimento dei dati DTO vogliamo fare un oggetto UserLocationDto che abbia le properties di User e di Location.

Es.

UserLocationDTO {

// properties user

// properties location

Come abbiamo visto il pattern permette quindi di trasferire i dati dal controller verso lo strato di presentazione, una cosa importante sarebbe avere un mapping automatico che ci permetta di portare i dati dal model nel dto e viceversa.

## Perchè usare DTO

Se si sviluppa lo strato di business e i client in maniera indipendente (ad esempio, un microservizio Java con un’API REST e un front-end JavaScript), l’uso di DTO permette di creare API stabili non soggette ad eventuali cambiamenti di requisiti o ottimizzazioni che avvengono nello strato di business.

Sempre nel contesto di una API REST i DTO consentono di escludere alcuni attributi delle entità, ad esempio attributi ad uso interno che non devono essere visibili a nessun utente, oppure escludere lunghi elenchi di entità associate (relazioni one-to-many o many-to-many) che invece è meglio reperire con specifici endpoint REST ai fini di una maggiore efficienza, selezionando solo i dati utili al tuo caso d’uso.

· Disaccoppiamento BL con PL

· Ottimizzazione degli oggetti di PL

## Svantaggi di DTO

Il principale svantaggio del pattern DTO è l’introduzione di ridondanza dei dati in quei casi in cui il DTO è molto simile se non uguale all’entity. Oltre ad avere due classi praticamente uguali bisogna fare un doppio lavoro quando l’entity dovesse cambiare per qualche motivo.

Inoltre, come insegna l’esperienza, è spesso difficile disaccoppiare le operazioni di CRUD di una API

dal livello di persistenza. Anche se è vero che l’uso di DTO consente di cambiare le tue entità senza cambiare la tua API, i progetti del mondo reale dimostrano che se cambi le tue entità, spesso è necessario modificare anche i DTO. Se ti trovi in una situazione in cui il tuo DTO è identico alla tua entità e sono previsti spesso cambiamenti, dovresti prendere in considerazione la rimozione del DTO ed esporre direttamente l’entità.

· Ridondanza di oggetti

· Manutenzione del codice.

# MapStruct.

MapStruct è un framework che ci permette di introdurre il pattern DTO sfruttando il vantaggio di avere disaccoppiamento tra BL e PL e non dover riscrivere oggetti simili e soprattutto rifare il binding dei dati.

<https://mapstruct.org/>

MapStruct ci mette a disposizione un generatore di codice che ci permette di mappare le entity con i DTO.

In generale è uno strumento per poter effettuare il mapping tra oggetti diversi es. entity e DTOs, questo viene fatto tramite annotazioni e andando a generare in automatico parti di codice implementativo a partire da una interfaccia.

# Casi di uso: publication

Facciamo la seguente struttura per permettere all’applicazione di poter avere la struttura voluta:

src -----

|-------- dto

|-------- mapper

All’interno del package dto andiamo ad inserire gli oggetti di mapping della PL (presentation logic) mentre all’interno del package mapper andiamo ad inserire gli oggetti di conversione tra MODEL e DTO.

Inoltre all’interno del pom.xml dobbiamo inserire le librerie di mapStruct come evidenziato di seguito:

<!-- mapstruct -->

<dependency>

<groupId>org.mapstruct</groupId>

<artifactId>mapstruct</artifactId>

<version>${mapstruct.version}</version>

</dependency>

<!-- mapstruct processor -->

<dependency>

<groupId>org.mapstruct</groupId>

<artifactId>mapstruct-processor</artifactId>

<version>${mapstruct.version}</version>

</dependency>

<!-- lombok interaction mapstruct -->

<!-- https://mvnrepository.com/artifact/org.projectlombok/lombok-mapstruct-binding -->

<dependency>

<groupId>org.projectlombok</groupId>

<artifactId>lombok-mapstruct-binding</artifactId>

<version>0.1.0</version>

</dependency>

Mapper:

public interface IMapper<DTO, MODEL> {

DTO toDto(MODEL model);

DTO toDto(Optional<MODEL> model);

List<DTO> toDtos(List<MODEL> models);

//Page<DTO> toDtos(Page<MODEL> models);

MODEL dtoToModel(DTO dto);

List<MODEL> DtosToModels(List<DTO> dtos);

}

AuthorDTO

@Data

@AllArgsConstructor

@NoArgsConstructor

@Builder

public class AuthorDto extends BaseDto<Long> {

private Long id;

private String nome;

private String cognome;

private Pubblication pubblication;

}

Introduciamo per generalizzare il processo di CRUD con le classi Service di traduzione da MODEL a DTO e nelle classi di Controller.

@NoArgsConstructor

public class BaseCrudDtoService

<

REPOSITORY extends JpaRepository<MODEL, ID>,

MAPPER extends IMapper<DTO, MODEL>,

DTO extends BaseDto<ID>,

MODEL extends BaseId<ID>,

ID> implements ICrudDtoService<DTO, ID

> {

protected REPOSITORY repository;

protected MAPPER mapper;

public BaseCrudDtoService(REPOSITORY repository, MAPPER mapper) {

this.repository = repository;

this.mapper = mapper;

}

@Override

public List<DTO> findAll() {

List<MODEL> models = repository.findAll();

return this.convertModelsToDtos(models);

}

@Override

public Page<DTO> findAll(Pageable page) {

Page<MODEL> models = repository.findAll(page);

return this.convertPagetoDtos(models);

}

@Override

public DTO findById(ID id) {

return this.convertModelToDto(repository.findById(id));

}

@Override

public DTO save(DTO dto) {

MODEL model = this.mapper.dtoToModel(dto);

MODEL m = repository.save(model);

return convertModelToDto(m);

}

@Override

public List<DTO> saveAll(List<DTO> dtos) {

List<DTO> mout = new ArrayList<>();

for (DTO dto : dtos) {

MODEL mm = repository.save(this.mapper.dtoToModel(dto));

mout.add(this.convertModelToDto(mm));

}

return mout;

}

@Override

public void delete(DTO dto) {

deleteById(dto.getId());

}

@Override

public void deleteById(ID id) {

// read

Optional<MODEL> m = repository.findById(id);

if (m.isEmpty()) {

throw new RuntimeException(String.*format*("Entity with id %s not found", id));

}

repository.delete(m.get());

}

@Override

public DTO update(DTO dto) {

// read

MODEL min = this.mapper.dtoToModel(dto);

Optional<MODEL> m = repository.findById(min.getId());

if (m.isEmpty())

throw new RuntimeException(String.*format*("Entity with id %s not found", min.getId()));

MODEL mm = repository.save(m.get());

return this.mapper.toDto(mm);

}

private List<DTO> convertModelsToDtos(List<MODEL> models) {

List<DTO> dtoToBeReturned = new ArrayList<>();

models.forEach(model -> {

dtoToBeReturned.add(this.convertModelToDto(model));

}

);

return dtoToBeReturned;

}

private DTO convertModelToDto(MODEL model) {

return this.mapper.toDto(model);

}

private DTO convertModelToDto(Optional<MODEL> model) {

if (model != null && !model.isEmpty() && model.get() != null)

return this.mapper.toDto(model.get());

Optional<DTO> dto = Optional.*empty*();

return dto.get();

}

private Page<DTO> convertPagetoDtos(Page<MODEL> models) {

List<Page<DTO>> dtoToBeReturned = new ArrayList<>();

Page<DTO> p = new PageImpl(models.getContent(), models.getPageable(), models.getTotalPages());

return p;

}

}

public class BaseCrudDtoController<

SERVICE extends BaseCrudDtoService,

REPOSITORY extends JpaRepository<MODEL, ID>,

MAPPER extends IMapper<DTO, MODEL>,

DTO extends BaseDto<ID>,

MODEL extends BaseId<ID>,

ID>

implements ICrudDtoController<DTO, ID> {

protected SERVICE service;

public BaseCrudDtoController(SERVICE service) {

this.service = service;

}

// ?page=0&size=2&sort=createdAt,desc

@GetMapping("/paged")

public ResponseEntity<Page<DTO>> findAll(Pageable pageable) {

Page<DTO> page = service.findAll(pageable);

return ResponseEntity.*ok*(page);

}

@Override

@GetMapping("/")

public ResponseEntity<List<DTO>> findAll() {

return ResponseEntity.*ok*(service.findAll());

}

@Override

@GetMapping("/{id}")

public ResponseEntity<DTO> findById(@PathVariable ID id) {

return ResponseEntity.*ok*((DTO) service.findById(id));

}

@Override

@PostMapping("/")

public ResponseEntity<DTO> save(@RequestBody DTO model) {

return ResponseEntity.*ok*((DTO) service.save(model));

}

@Override

@PostMapping("/addAll")

public ResponseEntity<List<DTO>> saveAll(@RequestBody List<DTO> model) {

return ResponseEntity.*ok*(service.saveAll(model));

}

@Override

@DeleteMapping("/")

public ResponseEntity<DTO> delete(DTO model) {

DTO m = (DTO) service.findById(model.getId());

if (m != null)

service.delete(model);

return ResponseEntity.*ok*(m);

}

@Override

@PostMapping("/{id}")

public ResponseEntity<DTO> deleteById(@PathVariable ID id) {

DTO m = (DTO) service.findById(id);

if (m != null)

service.deleteById(id);

return ResponseEntity.*ok*(m);

}

@Override

@PutMapping("/")

public ResponseEntity<DTO> update(@RequestBody DTO model) {

return ResponseEntity.*ok*((DTO) service.save(model));

}

}

I metodi di conversione possono essere messi a comune su una classe di interfaccia che permetta in maniera del tutto generica di convertire model e dto e viceversa.

## Auth: esercitazione.

Prendiamo e convertiamo il progetto Auth dal repository di github e convertiamolo utilizzando le librerie introdotte:

· Mapstruct

· Lombok

· Datasource

o Sqlite

o mySQL

o postgres

## chinookjpa: esercitazione.

Prendiamo e convertiamo il progetto chinookjpa dal repository di github e convertiamolo utilizzando le librerie introdotte:

· Mapstruct

· Lombok