Spring Intro

Започваме с нов проект и избирам Spring Initializr слагам име избирам приставките(Spring data JPA, MySQl Driver) за MySQL и ми се генерират пома, application.properties в Срц в майн в ресурси – който е празен и си го копирам от предния проект, в тестовете ми генерира и файл за писане на юнит тестове, и в Срц, в майн, в Джава има папка в която да си правя проекта и SpringBootApplication който стартира приложението

1). В application.properties променям само името и паролата и базата която да търси но може и да се изпише и така че да я създава ако я няма. Ето така: spring.datasource.url=

jdbc:mysql://localhost:3306/shampoo\_company?useSSL=false&createDatabaseIfNotExist=true

#Data Source Properties  
spring.datasource.driverClassName = com.mysql.cj.jdbc.Driver  
spring.datasource.url = jdbc:mysql://localhost:3306/new\_lab?useSSL=false

#new\_lab е името на базата  
spring.datasource.username = root  
spring.datasource.password = 1234  
  
#JPA Properties  
spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.MySQL8Dialect  
spring.jpa.properties.hibernate.format\_sql = TRUE  
spring.jpa.hibernate.ddl-auto = update  
  
###Logging Levels  
# Disable the default loggers  
logging.level.org = WARN  
logging.level.blog = WARN  
  
#Show SQL executed with parameter bindings  
logging.level.org.hibernate.SQL = DEBUG  
logging.level.org.hibernate.type.descriptor = TRACE

2). SpringBootApplication задейства спринга (стартира) – този файл(клас ни се генерира автоматично при създаване на проекта по горе описания начин) Тук не пипам нищо, при изпълнение на долното спринг търси и отива в ConsoleRunner

@SpringBootApplication  
public class SpringIntroExerciseApplication {  
  
 public static void main(String[] args) {  
 SpringApplication.*run*(SpringIntroExerciseApplication.class, args);  
 }

3). ConsoleRunner

Понеже нямам Майн и в спринг се използва долния клас за да комуникирам със спринг , да извикваме и използваме нещата в service

Това е аналога на майн в спринг

@Component -слага се за да знае спринг че трябва да изпълни кода в public void run(String... args)   
public class ConsoleRunner implements CommandLineRunner {  
  
 private final SeedService seedService;

Инициализираме финална стойност за Service който ще използваме в метода. И долу го вкарваме в конструктор с анотация @Autowired(задължително може и на полето и на конструктора като конструктора се прави само заради тази анотация) като така се осъществява връзка със нужните ни сървиси(прави Injection)   
  
 @Autowired  
 public ConsoleRunner(SeedService seedService) {  
 this.seedService = seedService;  
 }

@Override  
 public void run(String... args) throws Exception {  
  
 this.seedService.seedAuthors();

Казваме на този сървиз да извика дадения метод   
 }  
}

4). Service – отговаря за бизнес логиката, необходимия ни код за решаване на даден проблем (може да се прескочи и в рънъра да се свържем с репозиторито )

public interface SeedService {

Първо създаваме интерфейс в който дефинираме поведението на този сървиз (Спрямо името той отговаря за вкарването на обекти в нашите таблици “entity”) като в случая имаме методи за добавяне към трите ентита . За дефолтния метод трябва да прочета за да разбера защо се прави – прави се за да може да се пуснат едновременно всички методи.   
  
 void seedAuthors() throws IOException;  
  
 void seedCategories();  
  
 void seedBooks();  
  
 default void seedAll() throws IOException {  
 seedAuthors();  
 seedCategories();  
 seedBooks();  
 }  
  
}

Имплементацията на сървиса

@Service Задължително се слага за да покажем на спринг че това е сървис и да се вземе в предвид  
public class SeedServiceImpl implements SeedService {  
  
 private static final String *RESOURCE\_PATH* =  
 "src\\main\\resources\\files";  
 private static final String *AUTHORS\_FILE\_NAME* =  
 *RESOURCE\_PATH* + "\\authors.txt";  
 Горе двете са за да посочим пътя до файловете понеже данните които ще инпортнем са ни дадени в отделни файлове. Като си ги копираме и ги поставяме в новосъздадена папка в ресурсите в Интелиджей иначе може и да копираме директорията от папката където се намират

@Autowired правим задължителна препратка към репозиторито като трябва да се добавят и за другите ентита.Така се връзваме с базата и взимаме или вкарваме данни в нея.   
 private AuthorRepository authorRepository;

И тук написваме метода за въвеждане на данните в първото ентити   
 @Override  
 public void seedAuthors() throws IOException {  
 Files.*readAllLines*(Path.*of*(*AUTHORS\_FILE\_NAME*))  
 .stream()  
 .filter(s -> !s.isBlank())  
 .map(s -> s.split(" "))  
 .map(name -> new Author(name[0], name[1]))  
 .forEach(authorRepository::save);

На първия ред казваме на обекта Files да прочете всички редове от посочената директория(*readAllLines* – се използва само при малки файлове като този случай).На втория ред пускаме прочетеното в поток. На трети правим филтър(.isBlank – проверява дали стринга е празен или е съставен от интервали – връща true ако е така и false ако стринга е нормален)за да вземем само валидните стрингове. На четвърти ред сплитваме по празно място, защото са две имена – първо и второ. На пети след предния ред се получава масив на който даваме име name и оттук правим нов автор с имена от съответните индекси на този масив. На последния ред казваме на репозиторито всеки нов автор да се запази в базата.  
 }  
  
 @Override  
 public void seedCategories() {  
 Files.*readAllLines*(Path.*of*(*CATEGORIES\_FILE\_NAME*))  
 .stream()  
 .filter(s -> !s.isBlank())  
 .map(Category::new)  
 .forEach(categoryRepository::save);

}  
  
 @Override  
 public void seedBooks() {

Files.*readAllLines*(Path.*of*(*BOOKS\_FILE\_NAME*))  
 .stream()  
 .filter(s -> !s.isBlank())  
 .map(this::getBookObject)  
 .forEach(bookRepository::save);

}  
}

За да създадем обект от Книга използваме отделен метод понеже има много полета и ще направим стреама горе не четим.

private Book getBookObject(String line) {  
  
 String[] bookParts = line.split(" ");  
 Сплитваме прочетеното от файла

EditionType editionType = EditionType.*values*()[Integer.*parseInt*(bookParts[0])];  
 Взимаме числова стойност за енумерацията

LocalDate releaseDate =LocalDate.*parse*(bookParts[1], DateTimeFormatter.*ofPattern*("d/M/yyyy"));  
 Взимаме датата и посочваме в какъв формат да се запази

int copies = Integer.*parseInt*(bookParts[2]);  
 Взимаме броя на копията

BigDecimal price = new BigDecimal(bookParts[3]);  
 Взимаме цената

AgeRestriction ageRestriction =

AgeRestriction.*values*()[Integer.*parseInt*(bookParts[4])];  
 Взимаме числова стойност за енумерацията

String title = Arrays.*stream*(bookParts)  
 .skip(5)  
 .collect(Collectors.*joining*(" "));  
 Понеже всички останали индекси от масива който сме прочели и сплитнали от файла, са част от заглавието пускаме стреама така че да пропусне първите 5 индекса и другите ги взимаме с интервал м/у тях.

Author author = authorService.getRandomAuthor();

Правим още два сървиса, за автор и за категория (първо интерфейс и после имплементацията) където описваме методите(да вземем случайни автор и категория се изисква в заданието като за категория ни трябва сет защото една книга попада в много категории и със сет си гарантираме че няма да има повтарящи се категории)  
  
 Set<Category> categories = categoryService.getRandomCategories();  
  
 return new Book  
 (title, editionType, price, copies  
 , releaseDate, ageRestriction, author, categories);  
}

Имаме специфична бизнес логика за създаването на книгата, а именно при създаването и ни трябва случаен автор и няколко случайни категории. Тази логика е обвързана със конкретните класове затова правим отделни сървиси, така е и по-лесно за ъпдейт на кода(солид принципите)

Интерфейса

public interface AuthorService {  
  
 Author getRandomAuthor();  
}

и имплементацията

@Service – задължителната анотация  
public class AuthorServiceImpl implements AuthorService {  
  
 private final AuthorRepository authorRepository;

Правим връзка със базата  
  
 @Autowired задължителната анотация  
 public AuthorServiceImpl(AuthorRepository authorRepository) {  
 this.authorRepository = authorRepository;  
 }  
  
 @Override  
 public Author getRandomAuthor() {  
 long size = this.authorRepository.count();

Взима броя на авторите от базата  
  
 int authorId = new Random().nextInt((int) size) + 1;

Случайно ид на автор. Взимаме го като използваме

new Random() който е метод който взима нещо на случаен принцип в случая инт от броя автори, а даваме +1 защото nextInt взима число от 0 включително до посочената граница която не е включена(ако са 10 автора ще вземе от 0 до 9)и с +1 си гарантираме че няма да върне 0 и ще може да вземе 10 понеже в таблиците ид-тата почват от 1 не от 0.   
  
 return this.authorRepository.findById(authorId).get();

връщаме, казваме на репозиторито да намери по рандом ид-то автор, този метод иска опционален клас при който се прави проверка дали посоченото ид съществува, за да избегнем преобразуването на опционален клас му казваме .get(); което извършва метода без проверка, не е добра практика да се използва но в нашия случай сме сигурни че подаваме валидно ид.   
 }  
}

И другия интерфейс

public interface CategoryService {  
  
 Set<Category> getRandomCategories();  
}

имплементация

@Service задължителната анотация  
public class CategoryServiceImpl implements CategoryService {  
 private final CategoryRepository categoryRepository;  
  
 @Autowired задължителната анотация  
 public CategoryServiceImpl(CategoryRepository categoryRepository) {  
 this.categoryRepository = categoryRepository;  
 }

Разликата с предния сървиз е че тук ни трябва списък с уникални категории(техните ид-та) затова използваме Сет.  
 @Override  
 public Set<Category> getRandomCategories() {  
 long size = this.categoryRepository.count();  
 Пак взимаме броя на категориите   
 Random random = new Random();

Правим си отделно нов рандом клас  
 int categoriesCount = random.nextInt((int) size) + 1;  
 Взимаме като отделна променлива случаен брой категории  
 Set<Integer> categoriesIds = new HashSet<>();  
 Създаваме празен сет  
 for (int i = 0; i < categoriesCount; i++) {  
 int nextId = random.nextInt((int) size) + 1;  
 categoriesIds.add(nextId);  
 }  
 пускаме цикъл до взетия брой категории и на всяка итерация взимаме и слагаме случайно ид в сета

List<Category> allById =

this.categoryRepository.findAllById(categoriesIds);

Казваме на репозиторито да намери всички категории спрямо запазените ид-та и да ги запази в лист.  
 return new HashSet<>(allById);

Връщаме нов сет от листа.   
 }  
}

5).Repository – отговаря за комуникацията с базата,

@Repository  
public interface AuthorRepository extends JpaRepository<Author, Integer> {  
  
   
}

Тук е празно защото все още нямам нужда да дефинирам методи, в репозиторито си има вградени методи за общуване с базата (базови опашки за изпълнение), и на нашия сървиз му трябва репозитори с което да се изпълнят заявките към базата.

6). Entities – модели, обекти, субекти. Тук пазим нашите обекти които превръщаме в таблици в базата данни

Първото ентити

@Entity(name = "books") Казваме че е ентити и казваме какво име да приеме таблицата (може и да не се пише и тогава ще се генерира автоматично)  
public class Book {  
  
 @Id  
 @GeneratedValue(strategy = GenerationType.*IDENTITY*)

Първото поле даваме му анотация че е главен ключ и че е auto increment (двете анотации се пишат задължително за главен ключ)  
 private int id;  
  
 @Column(nullable = false,length = 50)

Анотация за колона , с първото в скобите казваме че не може да е нулл а с второто максималния брой символи(може и да не се пише анотацията но ако искаме да зададем някакви особености на колоната трябва)   
 private String title;  
  
 @Column(length = 1000)  
 private String description;  
  
 @Column(name = "edition\_type",nullable = false)

Тук даваме и име на колоната, правим го когато полето е от две думи за да може да го напишем по конвенцията SQL   
 private EditionType editionType;  
  
 @Column(nullable = false)  
 private BigDecimal price;  
  
 private int copies;  
  
 @Column(name = "release\_date")

Тук типа на данните е енумерация която дефинираме отделно   
 private LocalDate releaseDate;  
  
 @Column(name = "age\_restriction",nullable = false)  
 private AgeRestriction ageRestriction;  
  
 @ManyToOne

Създаваме поле с което да се създаде връзка с друго ентити. В случая гледаме от класа в който се намираме, книга и много книги могат да имат един автор и анотацията е тази  
 private Author author;  
  
 @ManyToMany

Тук много книги имат много категории  
 private Set<Category> categories;

Празен конструктор задължително, но трябва да има и още един с полетата но още не е написан   
 public Book() {}

Надолу са гетъри и сетъри на полетата  
  
 public String getTitle() {  
 return title;  
 }  
  
 public void setTitle(String title) {  
 this.title = title;  
 }  
  
 public String getDescription() {  
 return description;  
 }  
  
 public void setDescription(String description) {  
 this.description = description;  
 }  
  
 public EditionType getEditionType() {  
 return editionType;  
 }  
  
 public void setEditionType(EditionType editionType) {  
 this.editionType = editionType;  
 }  
  
 public BigDecimal getPrice() {  
 return price;  
 }  
  
 public void setPrice(BigDecimal price) {  
 this.price = price;  
 }  
  
 public int getCopies() {  
 return copies;  
 }  
  
 public void setCopies(int copies) {  
 this.copies = copies;  
 }  
  
 public LocalDate getReleaseDate() {  
 return releaseDate;  
 }  
  
 public void setReleaseDate(LocalDate releaseDate) {  
 this.releaseDate = releaseDate;  
 }  
  
 public AgeRestriction getAgeRestriction() {  
 return ageRestriction;  
 }  
  
 public void setAgeRestriction(AgeRestriction ageRestriction) {  
 this.ageRestriction = ageRestriction;  
 }  
  
 public Author getAuthor() {  
 return author;  
 }  
  
 public void setAuthor(Author author) {  
 this.author = author;  
 }  
  
 public Set<Category> getCategories() {  
 return categories;  
 }  
  
 public void setCategories(Set<Category> categories) {  
 this.categories = categories;  
 }  
}

Другите два класа

@Entity(name = "authors")  
public class Author {  
  
 @Id  
 @GeneratedValue(strategy = GenerationType.*IDENTITY*)  
 private int id;  
  
 @Column(name = "first\_name")  
 private String firstName;  
  
 @Column(name = "last\_name",nullable = false)  
 private String lastName;  
  
 @OneToMany(targetEntity = Book.class, mappedBy = "author")  
 private Set<Book> books;  
  
 public Author() {}  
  
 public Author(String firstName, String lastName) {  
 this.firstName = firstName;  
 this.lastName = lastName;  
 }  
  
 public int getId() {  
 return id;  
 }  
  
 public void setId(int id) {  
 this.id = id;  
 }  
  
 public String getFirstName() {  
 return firstName;  
 }  
  
 public void setFirstName(String firstName) {  
 this.firstName = firstName;  
 }  
  
 public String getLastName() {  
 return lastName;  
 }  
  
 public void setLastName(String lastName) {  
 this.lastName = lastName;  
 }  
  
 public Set<Book> getBooks() {  
 return books;  
 }  
  
 public void setBooks(Set<Book> books) {  
 this.books = books;  
 }  
}

@Entity(name = "categories")  
public class Category {  
  
 @Id  
 @GeneratedValue(strategy = GenerationType.*IDENTITY*)  
 private int id;  
  
 @Column(nullable = false)  
 private String name;  
  
 public Category() {}  
  
 public Category(String name) {  
 this.name = name;  
 }  
  
 public int getId() {  
 return id;  
 }  
  
 public void setId(int id) {  
 this.id = id;  
 }  
  
 public String getName() {  
 return name;  
 }  
  
 public void setName(String name) {  
 this.name = name;  
 }  
}

И енумерациите

public enum EditionType {  
 *NORMAL*, *PROMO*, *GOLD*}

public enum AgeRestriction {  
 *MINOR*, *TEEN*, *ADULT*}