02_Entity, repository & controller

September 23, 2023

1 Entity

Generamos un nuevo `package` llamado `models`. En este vamos a guardar guardar las clases de datos con las que vamos a trabajar. Estas clases a su vez van a ser entidades en la base de datos. Dentro del `package models` creamos una nueva clase llamada `Book` y la configuramos de la siguiente manera.

- `@Entity`: la clase va a tener las `annotation @Entity` que indica que la clase va a ser una entidad. A partir de está se va a generar una tabla en la base de datos con el nombre de la clase, en el ejemplo una tabla con el nombre `book`.
- `@Id`: la `annotation @Id` indica que el atributo `private Long id` va a ser la clave primaria de la tabla en la base de datos.
- Mediante `@GeneratedValue(strategy = GenerationType.AUTO, generator = "native")` y `@GenericGenerator(name = "native", strategy = "native")` indicamos la forma en que se van a generar los valores para el atributo `id`. Básicamente indicamos que los valores se generen de forma automática con funcionalidades de la base de datos utilizadas para esto.
- Atributos de la clase: los atributos de la clase van a representar columnas en la tabla que se genera en la base de datos.
- `Constructor, Getters y Setters`: tienen que estar declarados el constructor vacío y el que permite generar objetos del tipo de la clase. Nótese que el constructor no asigna valores al atributo `id` por que los valores de mismo se generan en forma automática en la base de datos. Tienen que estar declarados todos los `getters` y `setters` a excepción del `setter` de id.

[ ]:
```java
@Entity
public class Book {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO, generator = "native")
    @GenericGenerator(name = "native", strategy = "native")
    private Long id;
    private String isbn;
    private String title;
    private LocalDate date;
    private String synopsis;
```

```
    public Book() {}
    public Book(String isbn, String title, LocalDate date ,String synopsis) {
        this.isbn = isbn;
        this.title = title;
        this.date = date;
        this.synopsis = synopsis;
    }

    // AQUÍ CORRESPONDEN LOS GETTERS Y SETTERS
}
```

2 Respostory & service

2.1 Repository

Generamos un nuevo `package` llamado `repositories`. En este vamos a guardar `interfaces` con las que vamos a extender a la clase `JpaRepository`. Mediante la clase `JpaRepository` vamos a acceder a métodos que nos permiten crear, leer, actualizar y borrar datos de la base de datos (CRUD). Dentro de la `package repositories` creamos la interface llamada `BookRepository`. Configuramos la interface de la siguiente manera.

- `@RepositoryRestResource`: mediante la `annotation @RepositoryRestResource` indicamos que la interface va a ser utilizada para construir una arquitectura `REST`, que conecta sistemas mediante el protocolo `HTTP`.
- `extends JpaRepository`: mediante `extends` indicamos que la interface `BookRepository` hereda o es hija de la clase `JpaRepository`. Esto quiere decir que hereda y puede sobrescribir sus atributos y métodos.
- `<Book, Long>`: entre las `<>` indicamos primero el tipo de objeto para el que creamos la interface, en este caso son objetos de la clase `Book` y segundo indicamos el tipo de datos del atributo con la `annotation @id` de la clase `Book`, que en este caso es `Long`.
- De momento no necesitamos declarar métodos en la interface, podemos trabajar directamente con los que hereda de `JpaRepository`.

[ ]:
```
@RepositoryRestResource
public interface BookRepository extends JpaRepository <Book, Long>{
}
```

2.1.1 DTO

Generamos un nuevo `package` llamado `dtos`. En este vamos a guardar clases con las que vamos manipular los datos entran y salen de la API. DTO significa Data Transfer Objects, u objetos de transferencia de datos, y se utilizan para esto mismo, para el envío y recepción de datos. Dentro de la `package dtos` creamos la clase llamada `BookDTO`. Configuramos la clase de la siguiente manera.

- Los atributos son los mismos que los de la clase para la cual creamos el DTO, es decir, los mismos atributos de la clase `Book`.
- Tiene que tener el constructor vacío declarado.

- El constructor con parámetros solo recibe un objeto de clase `Book` llamado `book`. De ese objeto vamos a sacar los valores de los atributos para construir un objeto de tipo `BookDTO`.
- Solo tenemos que generar los métodos `getters` ya que no vamos a hacer `setters` de los atributos del DTO, los mismos toman sus valores en el constructor a partir de un objeto de tipo `Book`.

[ ]:
```java
public class BookDTO {

    private Long id;
    private String isbn;
    private String title;
    private LocalDate date;
    private String synopsis;

    public BookDTO() {
    }
    public BookDTO(Book book) {
        this.id = book.getId();
        this.isbn = book.getIsbn();
        this.title = book.getTitle();
        this.date = book.getDate();
        this.synopsis = book.getSynopsis();
    }

    // SOLO GETTERS
}
```

2.2 Service

Generamos un nuevo `package` llamado `services`. En este vamos a guardar `interfaces` que van a ser implementadas por clases que van a sobrescribir sus métodos. Dentro de la `package services` creamos la interface llamada `BookService`. Configuramos la interface de la siguiente manera.

- `void createBook(Book book)`: utilizamos el método `createBook`, que no devuelve nada, para crear un nuevo registro en la tabla `Book` de la base de datos, con los atributos del objeto `book` que le pasamos por parámetro. El nombre `createBook` del método es una elección personal. El método puede tener el nombre que queramos.
- `List<BookDTO> readAll()`: utilizamos el método `readAll`, que devuelve una lista de objetos de tipo `BookDTO`, para traer todos los registros de la tabla `Book`. El nombre `readAllBooks` del método es una elección personal. El método puede tener el nombre que queramos.
- `BookService` es una interface, por lo que solo tenemos que declarar los métodos, su implementación la vamos a hacer en una clase creada para esto.

[ ]:
```java
public interface BookService {

    // CREATE
    void createBook(Book book);
```

```
    // READ
    List<BookDTO> readAllBooks();
}
```

2.2.1 Implement

Generamos un nuevo `package` llamado `implement` dentro del `package services`. En este vamos a guardar clases que van a implementar los `services` que generamos en el `package services`. Llamamos al `package implement` y no `implements` por esta es una palabra reservada que se utilizar para la herencia. Dentro del `package implement` creamos la clase llamada `BookServiceImplement`. Configuramos la clase de la siguiente manera.

- `@Service`: mediante `@Service` indicamos que estamos creando una clase de servicio con la que vamos a acceder a funcionalidades de los repositorios.
- `implements BookService`: mediante `implements` indicamos que la clase implementa la interface `BookService`. Las clases pueden heredar solo de otra clase pero puede implementar de varias interfaces. Cuando implementamos una interface tenemos que, si o si, sobrescribir los métodos de la interface.
- `@Autowired`: mediante `@Autowired` instanciamos un objeto de la clase que indicamos inmediatamente abajo. En este caso, un objeto de la clase `BookRepository` al que llamamos `bookRepository`. Mediante este objeto vamos a acceder a los métodos que la clase BookRepository hereda de `JpaRepository`.
- `@Override`: mediante la `annotation @Override` indicamos que la linea a continuación no solo va a sobrescribir el método de la interface que implementa la clase sino que además va a anular este método. De esa forma, al llamar al método desde la interface `BookService` vamos a estar llamando al método sobrescrito en la clase `BookServiceImplement`.
- `bookRepository.save(book)`: mediante el método `save`, heredado por `BookRepository` de `JpaRepository`, guardamos un objeto de la clase `Book` en la base de datos.
- `List<BookDTO> readAll()`: mediante `List<BookDTO>` indicamos que el método va a devolver un listado de objetos de la clase BookDTO.
- `bookRepository.findAll().stream()`: mediante el método `findAll`, heredado por `BookRepository` de `JpaRepository`, traemos todos los registros guardados en la tabla Book de la base de datos. El método `.stream()` que acompaña a `findAll` nos permite tratar a los datos devueltos por `findAll` como una colección.
- `.map(book -> new BookDTO(book))`: el método `.map` nos permite manipular uno a uno los elementos de la colección que generamos mediante `.stream()`. Utilizamos la función flecha `book -> new BookDTO(book)` para tratar a cada elemento de la colección, al que llamamos `book` solo por coherencia, ya que podría llamarse `element`, y con cada uno de estos creamos un nuevo objeto de la clase `BookDTO`.
- `.collect(Collectors.toList())`: mediante el método `.collect` indicamos que hacer con los objetos de la clase `BookDTO` que devuelve `map`. El método `Collectors.toList()`, que pasamos como parámetro de `collect`, indica a `collect` que devuelva un listado de los objetos

devueltos por `map`, es decir, un listado de objetos de tipo `BookDTO`, que es lo que el método `readALl` tiene que devolver según la definición del mismo.

[ ]:
```java
@Service
public class BookServiceImplement implements BookService {

    @Autowired
    private BookRepository bookRepository;

    @Override
    public void createBook(Book book) {
        bookRepository.save(book);
    }

    @Override
    public List<BookDTO> readAll() {
        return bookRepository.findAll().stream()
                .map(book -> new BookDTO(book))
                .collect(Collectors.toList());
    }
}
```

3 Controller