DESARROLLO DE APLICACIONES JEE CON SPRING FRAMEWORK

Dirección Nacional de Educación Continua

13-11-2021 Módulo 3

INTRODUCCIÓN

- La presente tiene como objetivo explicar las capacidades de un entorno de trabajo, sus funciones y capacidades, además explicar en detalle algunos de los módulos del framework de spring, con la intención de preparar al lector para generar soluciones empresariales simples y efectivas, conociendo la arquitectura y con esto ser un aporte al momento de diseñar una aplicación, aplicando los conocimientos de este documento.

RESUMEN

- En el Módulo anterior, hemos revisado lo siguiente:
- Framework Spring MVC, y sus capacidades.
- Inyección de dependencias.
- Spring Boot
- Thymeleaf
- DataSource
- Capa Servicios
- Capa DAO

CONFIGURANDO UN DATASOURCE EN SPRING

- Spring no sólo es muy fácil de integrar con estos productos tal y hemos visto en él documento, sino que además proporciona con Spring Data JPA su propia capa de abstracción que nos facilita y simplifica aún más el trabajo
- Básicamente para la configuración de un datasource en Spring se debe importar la dependencia de la BD en el POM.xml y configurar en el Application.properties, en el siguiente tópico mostraremos un ejemplo de esto.

UTILIZANDO JDBC TEMPLATE EN SPRING

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

<!-- in-memory database -->
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
</dependency>

<!-- MySQL JDBC driver
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
</dependency>
-->

<!-- Postgresql
<dependency>
    <groupId>org.postgresql</groupId>
    <artifactId>postgresql</artifactId>
    <version>42.2.5</version>
</dependency>
-->
```

- JdbcTemplate es una de las formas de acceder a la base de datos proporcionada por Spring, y es la forma de implementación más básica y de nivel inferior para acceder a la base de datos en Spring.
- Lo primero es configurar el proyecto para lo cual se debe considerar los siguientes ficheros.
  - POM.xml
  - Application.properties
- Se deben incluir las dependencias de la BD a la que nos conectaremos y el spring-boot-starter-jdbc, para el ejemplo será H2.
- En el application.properties, se deben incluir las configuraciones de Hikari y la BD elegida.

UTILIZANDO JDBC TEMPLATE EN SPRING

```
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.ParameterizedPreparedStatementSetter;
import org.springframework.jdbc.core.RowMapper;
import org.springframework.jdbc.core.support.AbstractLobCreatingPreparedStatementCallb
import org.springframework.jdbc.support.lob.LobCreator;
import org.springframework.jdbc.support.lob.LobHandler;
import org.springframework.stereotype.Repository;
import org.springframework.transaction.annotation.Transactional;

import cl.aiep.Book;

import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStream;
import java.math.BigDecimal;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.Optional;

@Repository
public class JdbcBookRepository implements BookRepository {

    @Autowired
    private JdbcTemplate jdbcTemplate;

    @Autowired
    LobHandler lobHandler;

    @Override
    public int count() {
        return jdbcTemplate
                .queryForObject("select count(*) from books", Integer.class);
    }
```

- Ahora analizaremos la anotación @Repository que utiliza jdbcTemplate
- Método count(), método que se encarga de contar la cantidad de Libros, utiliza jdbcTemplate y un SQL nativo para H2.

CREANDO UN DAO QUE UTILIZA JDBCTEMPLATE

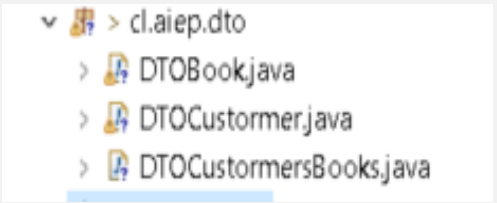
- Como vimos en el ejemplo anterior, una vez configurada la BD en nuestro proyecto podemos generar implementar el patrón DAO.
- ¿Qué es el patrón DAO?
- El patrón Data Access Object (DAO) propone separar por completo la lógica de negocio de la lógica para acceder a los datos, de esta forma, el DAO proporcionará los métodos necesarios para insertar, actualizar, borrar y consultar la información; por otra parte, la capa de negocio solo se preocupa por lógica de negocio y utiliza el DAO para interactuar con la fuente de datos.

REALIZANDO QUERYS QUE RECIBEN PARAMETROS

```java
// jdbcTemplate.queryForObject, populates a single object

public Optional<Book> findById(Long id) {
    return jdbcTemplate.queryForObject(
            "select * from books where id = ?",
            new Object[]{id},
            (rs, rowNum) ->
                    Optional.of(new Book(
                            rs.getLong("id"),
                            rs.getString("name"),
                            rs.getBigDecimal("price")
                    ))
    );
}
```

- Como vimos en el ejemplo anterior, las query's generadas con JdbcTemplate permiten consultar directamente las tablas asociadas a una Base de datos, además nos permite pasar parámetros de consulta como en la imagen.
- Como se puede ver, en la query generada se agrega un carácter de tipo "?" para indicar que se debe incluir un valor asociado, este carácter permite evitar un SqlInjection, recibiendo como parámetro anexo el valor del dicho "?".

MAPEANDO LOS RESULTADOS DE UNA CONSULTA



- El patrón DAO indica como buena práctica el volcar la data desde el origen a DTO's de Lógica de negocio, los DTO's permiten abstraer la lógica de negocio de la información contenida en bd, en un DTO se pueden contener varias entidades de Base de Datos o fuentes de Información, o lo que se necesite desde las capas mas arriba.
- En este caso utilizaremos las entidades Customer y Books y las uniremos en un DTO con el listado de ambas.

INVOCANDO UN DAO DESDE UN SERVICE

```java
@RestController
public class BookController {

    @Autowired // Test NamedParameterJdbcTemplate
    private BookService bookService;

    // Find
    @GetMapping("/books")
    List<Book> findAll() {
        return bookService.findAll();
    }

  // Find
    @GetMapping("/all")
    DTOCustormersBooks findAllBookCustomer() {
        return bookService.findCustomerBooks();
    }
```

```java
@Service
public class BookService {

    @Autowired
    private DAOBookRepository bookRepository;


    public DTOCustormersBooks findCustomerBooks() {
        return bookRepository.getCustBooks();
    }
```

- Como se explica anteriormente, un @Service permite generar una capa de abstracción entre las capas de controlador y backend, para este caso utilizaremos la siguiente estructura Controller(servicioRest) -> Service(CapaAbstracción -> DAO(Acceso a Datos).
- En este caso se genera un servicio tipo GET que obtiene todos los libros y clientes el endpoint .
- Se realiza la Inyección del DAO y se invoca al método que retorna el DTO con el listado de ambas entidades.
- En la clase DAO se genera la conversión

ACCESO A DATOS MEDIANTE JPA

- La persistencia de datos es un medio mediante el cual una aplicación puede recuperar información desde un sistema de almacenamiento no volátil y hacer que esta persista. La persistencia de datos es vital en las aplicaciones empresariales debido al acceso necesario a las bases de datos relacionales. Las aplicaciones desarrolladas para este entorno deben gestionar por su cuenta la persistencia o utilizar soluciones de terceros para manejar las actualizaciones y recuperaciones de las bases de datos con persistencia. JPA (Java™ Persistence API) proporciona un mecanismo para gestionar la persistencia y la correlación relacional de objetos y funciona desde las especificaciones EJB 3.0.

LA API DE PERSISTENCIA JPA

- Una de las principales características de JPA es que nos entrega APIS que nos evitan el escribir código SQL para acceder a la Base de Datos
- La API EntityManager puede actualizar, recuperar, eliminar o aplicar la persistencia de objetos de una base de datos.
- La API EntityManager y los metadatos de correlación relacional de objetos manejan la mayor parte de las operaciones de base de datos sin que sea necesario escribir código JDBC o SQL para mantener la persistencia.
- JPA proporciona un lenguaje de consulta, que amplía el lenguaje de consulta EJB independiente, conocido también como JPQL, el cual puede utilizar para recuperar objetos sin grabar consultas SQL específicas en la base de datos con la que está trabajando.

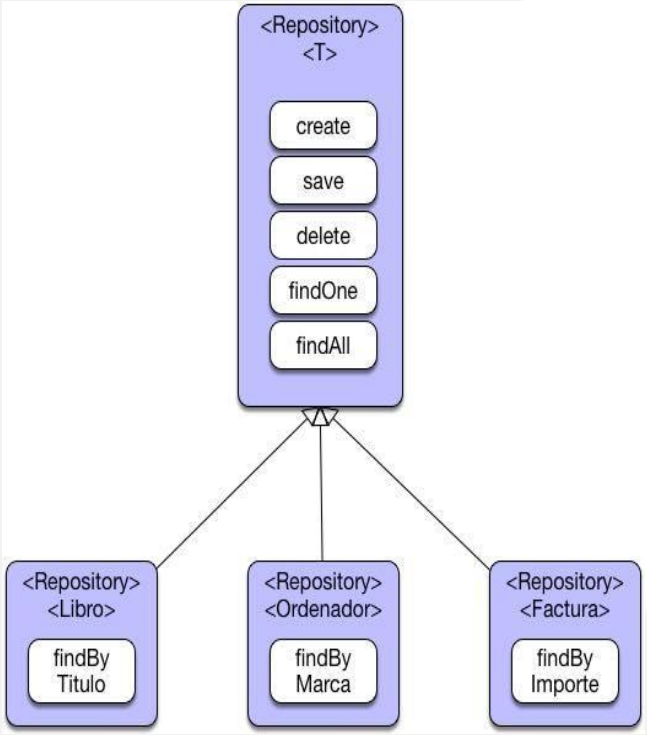
```java
@Entity
public class Book {

    @Id
    @GeneratedValue
    private Long id;
    private String name;
    private String author;
    private BigDecimal price;

    // avoid this "No default constructor for entity"
    public Book() {
    }

    public Book(Long id, String name, String author, BigDecimal price) {
        this.id = id;
        this.name = name;
        this.author = author;
        this.price = price;
    }
```

LA CLASE ENTIDAD JPA

- La clase Entidad o @Entity es una clase Java simple que representa una fila en una tabla de base de datos con su formato más sencillo. Los objetos de entidades pueden ser clases concretas o clases abstractas. Mantienen estados mediante la utilización de propiedades o campos, este tipo de clases ya la hemos visto como ejemplos en el documento.

ENTITY MANAGER JPA

- El entity manager en JPA es un Gestor de recursos que mantiene la colección activa de objetos de entidad que está utilizando la aplicación. EntityManager maneja la interacción y metadatos de bases de datos para las correlaciones relacionales de objetos. Una instancia de EntityManager representa un contexto de persistencia. Una aplicación en un contenedor puede obtener el EntityManager mediante inyección en la aplicación o buscándolo en el espacio de nombres del componente Java.



CLASE REPOSITORIO

- El concepto de Java Generic Repository es muy habitual cuanto trabajamos con tecnologías de persistencia. El concepto de Repository como clase que se encarga de gestionar todas las operaciones de persistencia contra una tabla de la base de datos .
- La mayor parte de los lenguajes de programación soportan el uso de clases genéricas . Estas permiten avanzar en el uso del patrón Repository y generar repositorios genéricos que reducen de forma significativa el código que tenemos que construir. Todos nuestros repositorios extenderán del repositorio genérico y añadirán más operaciones.

RECUPERAR, ACTUALIZAR, ELIMINAR UN OBJETO JPA

```java
// Find
@GetMapping("/books")
List<Book> findAll() {
    return repository.findAll();
}
```

```java
@PutMapping("/books/{id}")
Book saveOrUpdate(@RequestBody Book newBook, @PathVariable Long id) {

    return repository.findById(id)
            .map(x -> {
                x.setName(newBook.getName());
                x.setAuthor(newBook.getAuthor());
                x.setPrice(newBook.getPrice());
                return repository.save(x);
            })
            .orElseGet(() -> {
                newBook.setId(id);
                return repository.save(newBook);
            });
}
```

```java
@DeleteMapping("/books/{id}")
void deleteBook(@PathVariable Long id) {
    repository.deleteById(id);
}
```

- Como se explica anteriormente, un Repository nos permite generar todas las acciones CRUD asociadas a una entidad.

ASOCIACIONES UNO A UNO, UNO A MUCHOS

```java
@Entity
@Table(name = "book_category")
public class BookCategory {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

    private String name;

    @OneToMany(mappedBy = "bookCategory", cascade = CascadeType.ALL)
    private Set<Book> books;

    public BookCategory(){

    }
```

```java
@Entity
public class Book{
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

    private String name;

    @ManyToOne
    @JoinColumn(name = "book_category_id")
    private BookCategory bookCategory;

    public Book() {

    }
```

- @OneToOne Relation
- En una relación uno-a-uno, un elemento puede vincularse al único otro elemento. Significa que cada fila de una entidad se refiere a una y sólo una fila de otra entidad.
- @ManyToOne
- En una relación uno-a-muchos, un elemento puede vincularse a muchos otros elementos. Significa que cada fila de una entidad se refiere a una y sólo una fila de otra entidad.
- Nos permite considerar el ejemplo anterior. Cliente y Libro de manera unidireccional, la relación es relación uno-a-uno. Este tipo de relación fue explicado en el Modulo de Base de datos del curso.
- En el caso de JPA se representa de la siguiente forma.
- @OneToMany

INVOCAR UN REPOSITORIO DESDE UN SERVICIO

```java
@Service
public class BookService {
    @Autowired
    private BookRepository repository;

    public List<Book> findAll() {
        return repository.findAll();
    }
}
```

- Como se explica anteriormente, un @Service permite generar una capa de abstracción entre las capas de controlador y backend, para este caso utilizaremos la siguiente estructura Controller(servicioRest) --> Service(CapaAbstracción --> Repository

¿QUE ES LA TRANSACCIONALIDAD Y PORQUE ES IMPORTANTE?

- Una Transacción es un conjunto de operaciones ejecutadas como una unidad cuyo resultado final es una acción que manipula la base de datos. Un fallo en la ejecución de una de las operaciones provoca una reversión de la transacción, recuperando el estado inicial del a base de datos.
- Spring Transaction Management es una de las características más utilizadas e importantes del marco Spring. La gestión de transacciones es una tarea trivial en cualquier aplicación empresarial. Ya hemos aprendido a utilizar la API de JDBC para la gestión de transacciones. Spring proporciona un amplio soporte para la gestión de transacciones y ayuda a los desarrolladores a centrarse más en la lógica empresarial en lugar de preocuparse por la integridad de los datos en caso de fallas en el sistema.

CONFIGURANDO LA TRANSACCIONALIDAD

- Un aspecto muy interesante, es la propagación de la transaccionalidad a través de nuestra lógica de negocio. Por ejemplo, necesitamos un método transaccional, pero contemplamos la posibilidad de que exista una transacción superior, aun así queremos que esta transacción sea atómica y se ejecute de manera anidada. Para manejar este tipo de lógicas tan concretas tenemos la configuración de la propagación.

CREANDO UN SERVICIO TRANSACCIONAL

```
import java.util.List;

@Service
@Transactional
public class BookService {

      public DTOCustormersBooks findCustomerBooks() {
            return bookRepository.getCustBooks();
        }


    @Autowired
    private DAOBookRepository bookRepository;
```

- Como lo hemos visto en ejemplos anteriores para configurar el proyecto con transaccionalidad, es necesario modificar las clases @Service agregando la anotación @Transaction y seleccionando la forma de propagar si es necesario.

PRUEBA PRÁCTICA NRO3

- El alumno deberá crear un proyecto starter spring boot
- Configurar el proyecto para thymeleaf
- 2 entidades Libro, categoría con la siguiente estructura Libro es a una Categoría, mas una Categoría es a muchos libros
- Crear 3 DTO's DTOLibro, DTOCategoria, DTOListLibroCat
- Crear una clase de tipo DAO con patrón Facade que realice las conversiones de entidad a DTO
- Crear una clase java de tipo service que permita obtener por separado los dtos antes mencionados, además una búsqueda por id para cada entidad.
- Inyectar el Dao en la clase service.
- Inyectar el service en la clase rest controller con 5 métodos
  - Find libro
  - Find Catalogo
  - Save Libro
  - Save Catalogo
  - Obtener ambos
- Crear una clase TEST que valide el cumplimiento de las clase previamente realizadas.
- Incluir las librerías necesarias en el POM.xml para ejecutar las pruebas
- En el método obtener ambos via Thymeleaf template mostrar el listado de libros y categorías por separado.

AIEP
de la Universidad
Andrés Bello

Certified
B
Corporation

MUCHAS GRACIAS