Spring Framework

Spring MVC. Los repositorios

EE. SS. Mª AUXILIADORA

2º DAM
Autor: Manuel Torres Molina
ACCESO A DATOS

Spring Framework

Spring MVC. Los repositorios

Creación de las tablas en la base de datos

Creamos una tabla **course** con varios campos: **idCourse, name, description, price** y **hours**, con cualquiera de los clientes MySQL.

Configurar la persistencia con String Boot

Lo primero es añadir las siguientes dependencias en nuestro fichero **pom.xml**.

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

A continuación, desde un cmd y dentro de nuestra carpeta del proyecto ejecutamos **mvn clean install**, esto nos añadirá estas dependencias en nuestro proyecto.

Añadimos la configuración necesaria del starter en el archivo **application.yml**, que vienen en el Apendix A de Spring Boot.

Serían las siguientes:

```
spring:
  datasource:
    url: jdbc:mysql://localhost:3306/mydb
    username: root
    password:

  jpa:
    show-sql: true
    hibernate:
      ddl-auto: update
      naming:
        strategy: org.hibernate.cfg.ImprovedNamingStrategy
    properties:
      hibernate:
        dialect: org.hibernate.dialect.MySQL5Dialect
```

Creando entidades Hibernate

Creamos un nuevo paquete llamado **entity**, dentro de **src/main/java** y en el paquete de nuestro proyecto; hay vamos a crear las distintas entidades. En este caso crearíamos la clase **Course.java** con el siguiente código y anotaciones.

```
@Entity
@Table(name = "course") // Si la tabla se escribiera igual no haría falta
                                        // indicarlo en las anotaciones
public class Course {

	@Id
	@GeneratedValue
	@Column(name = "id")
	private int id;

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

	@Column(name = "description")
	private String description;

	@Column(name = "price")
	private int price;

	@Column(name = "hours")
	private int hours;

	public Course() {
	};

	public Course(int id, String name, String description, int price, int
			hours) {
		super();
		this.id = id;
		this.name = name;
		this.description = description;
		this.price = price;
		this.hours = hours;
	}

	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;
	}
```

```
	public String getDescription() {
		return description;
	}

	public void setDescription(String description) {
		this.description = description;
	}

	public int getPrice() {
		return price;
	}

	public void setPrice(int price) {
		this.price = price;
	}

	public int getHours() {
		return hours;
	}

	public void setHours(int hours) {
		this.hours = hours;
	}

}
```

Si eliminámos la tabla **course** de nuestra base de datos, al volver a lanzar el servidor Spring Boot y debido a la persistencia que hemos configurado en el fichero **application.yml**, volvería a incluirla en la base de datos dicha tabla,

```
hibernate:
	ddl-auto: update
```

ya que escanea todas las entitys que hay en la aplicación y hace una comprobación con la base de datos y la actualiza.

Por lo tanto, al consultar la base de datos volvería a existir esa tabla **course** que eliminamos antes.

Nuestro primer repositorio JPA

Dentro del paquete **repository**, creamos una interfaz llamada **CourseJpaRepository.java** donde vendrán los métodos de las distintas consultas a realizar y que utilizan a la entidad creada anteriormente. Se le especifica la anotación **@Repository**.

Esa interfaz hereda de la clase **JpaRepository**.

```
@Repository("courseJpaRepository")
public interface CourseJpaRepository extends JpaRepository<Course,
                                                    Serializable>{

	public abstract Course findByPrice(int price);
```

```
	public abstract Course findByPriceAndName(int price, String name);

	public abstract List<Course> findByNameOrderByHours (String name);

	public abstract Course findByNameOrderPrice (String name, int price);
}
```

Creando e integrando todas las capas. El Servicio.

Creamos en la carpeta **service** la interfaz **CourseService.java**:

```
public interface CourseService {

	public abstract List<Course> listAllCourses();
	public abstract Course addCourse(Course course);
	public abstract int removeCourse(int id);
	public abstract Course updateCourse(Course course);

}
```

A continuación, en el paquete **impl** que está dentro de **service**, creamos la clase **CourseServiceImpl.java** que implementa la anterior interfaz. Su código sería:

```
public class CourseServiceImpl implements CourseService {

	@Autowired
	@Qualifier("courseJpaRepository") //inyectamos en repositorio creado
	private CourseJpaRepository courseJpaRepository;

	@Override
	public List<Course> listAllCourses() {
		return courseJpaRepository.findAll();
	}

	@Override
	public Course addCourse(Course course) {

		return courseJpaRepository.save(course);
	}

	@Override
	public int removeCourse(int id) {

		courseJpaRepository.delete(id);
		return 0;
	}

	@Override
	public Course updateCourse(Course course) {
		return courseJpaRepository.save(course);
	}

}
```

Creando e integrando todas las capas. El controller

Crearemos un controlador llamado **CourseController.java** que integraremos con nuestro servicio. Se creará dentro del paquete **controller**. Contendrá dos métodos para listar cursos y añadir curso, se le inyectará el servicio creado en la anterior clase.

Controlador **CourseController.java**

```
@Controller
@RequestMapping("/courses")
public class CourseController {

	private static final String COURSE_VIEW="courses";

	@Autowired
	@Qualifier("courseServiceImpl")
	private CourseService courseService;

	//Métodos para listar
	@GetMapping("/listcourses")
	public ModelAndView listAllCourses() {
		ModelAndView mav=new ModelAndView(COURSE_VIEW);
		mav.addObject("courses", courseService.listAllCourses());
		return mav;

	}

	//Método para añadir curso
	@PostMapping("/addcourse")
	public String addCourse(@ModelAttribute("course") Course course) {
		courseService.addCourse(course);
		return "redirect:/courses/listcourses";
	}

}
```

Creando e integrando todas las capas. Las Vistas (parte 1)

Añadimos en el controlador anterior un LOG para poder comprobar en los métodos que se está realizando correctamente las acciones.

También añadimos un LOG para el servicio **CourseServiceImpl.java**.

También sobreescrimos el **toString** en nuestro Bean **Course**.

```
@Override
	public String toString() {
		return "Course [id=" + id + ", name=" + name + ", description=" +
		description + ", price=" + price + ", hours="+ hours + "]";
	}
```

Clase CourseController.java

```
@Controller
@RequestMapping("/courses")
public class CourseController {

	private static final String COURSES_VIEW="courses";

	private static final Log LOG = LogFactory.getLog(CourseController.class);


	@Autowired
	//@Qualifier("courseServiceImpl")
	private CourseService courseService;


	//Métodos para listar
	@GetMapping("/listcourses")
	public ModelAndView listAllCourses() {
		LOG.info("Call: "+ "ListAllCourses()");
		ModelAndView mav=new ModelAndView(COURSES_VIEW);
		mav.addObject("course", new Course());
		mav.addObject("courses", courseService.listAllCourses());
		return mav;

	}

	//Método para añadir curso
	@PostMapping("/addcourse")
	public String addCourse(@ModelAttribute("course") Course course) {
		LOG.info("Call: "+ "addCourse()"+" -- PARAM: " + course.toString());
		courseService.addCourse(course);
		return "redirect:/courses/listcourses";
	}

}
```

Clase CourseServiceImpl.java.

```
@Service("/courseServiceImpl")
public class CourseServiceImpl implements CourseService {

	private static final Log LOG = LogFactory.getLog(CourseServiceImpl.class);
	@Autowired
	@Qualifier("courseJpaRepository")
	private CourseJpaRepository courseJpaRepository;

	@Override
	public List<Course> listAllCourses() {
		LOG.info("Call: "+ "ListAllCourses()");
		return courseJpaRepository.findAll();
	}

	@Override
	public Course addCourse(Course course) {
		LOG.info("Call: "+ "addCourse()");
```

```
		return courseJpaRepository.save(course);
	}

	@Override
	public int removeCourse(int id) {
		courseJpaRepository.delete(id);
		return 0;
	}

	@Override
	public Course updateCourse(Course course) {
		return courseJpaRepository.save(course);
	}

}
```

Creamos una vista para listar los cursos llamada **courses.html**

```
<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1"/>
<title>Insert title here</title>
</head>
<body>
	<table>
		<thead>
			<tr>
				<th>Name</th>
				<th>Description</th>
				<th>Price</th>
				<th>Hours</th>
			</tr>
		</thead>
		<tbody>
			<tr th:each="course : ${courses}">
				<th><span th:text="${course.name}"></span></th>
			<th><span th:text="${course.description}">
				</span></th>
				<th><span th:text="${course.price}"></span></th>
				<th><span th:text="${course.hours}"></span></th>
			</tr>
		</tbody>
	</table>

</body>
</html>
```

Creando e integrando todas las capas. Las vistas (parte 2)

Aquí añadiremos en la misma vista anterior **courses.html** un formulario Post para la introducción de datos.

Probaremos el segundo método de **addcourse** de la clase **CourseController.java**.

```
<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1"/>
<title>Insert title here</title>
</head>
<body>
	<table>
		<thead>
			<tr>
				<th>Name</th>
				<th>Description</th>
				<th>Price</th>
				<th>Hours</th>
			</tr>
		</thead>
		<tbody>
			<tr th:each="course : ${courses}">
				<th><span th:text="${course.name}"></span></th>
				<th><span th:text="${course.description}"> </span></th>
				<th><span th:text="${course.price}"></span></th>
				<th><span th:text="${course.hours}"></span></th>
			</tr>
		</tbody>
	</table>

	<form action="#" th:action="@{/courses/addcourse}" th:object="${course}"
	method="post">
		<p>Name: <input type="text" th:field="*{name}"/></p>
		<p>Description: <input type="text" th:field="*{description}"/></p>
		<p>Price: <input type="text" th:field="*{price}"/></p>
		<p>Hours: <input type="text" th:field="*{hours}"/></p>
		<p><input type="submit" value="Submit"/></p>
	</form>

</body>
</html>
```

Tenemos que añadir en la clase CourseController.java un objeto para que thymeleaf pueda trabajar en la vista.

```
public ModelAndView listAllCourses() {
		LOG.info("Call: "+ "ListAllCourses()");
		ModelAndView mav=new ModelAndView(COURSES_VIEW);
		mav.addObject("course", new Course());
		mav.addObject("courses", courseService.listAllCourses());
		return mav;

	}
```

Transformar las entidades en modelos y viceversa. Converter

Vamos a ver cómo crear un Converter y utilizarlo en nuestra aplicación.

Hay datos en las tablas que no nos interesan enviar en las vistas, por eso existen los modelos para trabajar con ellos en las vistas. Hasta ahora en los controllers hemos trabajado con entitys pero eso no sería lo correcto, lo suyo es trabajar con modelos.

Para ello vamos a transformar los entitys en modelos y viceversa.

Creamos la clase **CourseConverter,java** con dos métodos, uno para transformar entidades en modelos y otro para transformar modelos en entidades.

Creamos una clase en el paquete **model** llamada **Course.java** que va a tener los mismos campos que la entity a excepción de la id, ya que ese dato no queremos llevarlo a la vista.

CourseModel.java (Modelo)

```
public class CourseModel {

	private String name;
	private String description;
	private int price;
	private int hours;

	public CourseModel() {

	}

	public CourseModel(String name, String description, int price, int hours) {
		super();
		this.name = name;
		this.description = description;
		this.price = price;
		this.hours = hours;
	}

	public String getName() {
		return name;
	}

	public void setName(String name) {
		this.name = name;
	}

	public String getDescription() {
		return description;
	}

	public void setDescription(String description) {
		this.description = description;
	}

	public int getPrice() {
```

```
		return price;
	}

	public void setPrice(int price) {
		this.price = price;
	}

	public int getHours() {
		return hours;
	}

	public void setHours(int hours) {
		this.hours = hours;
	}

}
```

La clase **CourseConverter.java** tendrá el siguiente código:

```
@Component("courseConverter")
public class CourseConverter {

	// Entity-->Model
	public CourseModel entity2model(Course course) {
		CourseModel courseModel = new CourseModel();
		courseModel.setName(course.getName());
		courseModel.setDescription(course.getDescription());
		courseModel.setPrice(course.getPrice());
		courseModel.setHours(course.getHours());
		return courseModel;

	}

	// Model-->Entity
	public Course model2entity(CourseModel courseModel) {
		Course course = new Course();
		course.setName(courseModel.getName());
		course.setDescription(courseModel.getDescription());
		course.setPrice(courseModel.getPrice());
		course.setHours(courseModel.getHours());
		return course;

	}
}
```

En el **CourseController.java** en vez de recibir una entidad, tenemos que recibir un modelo.

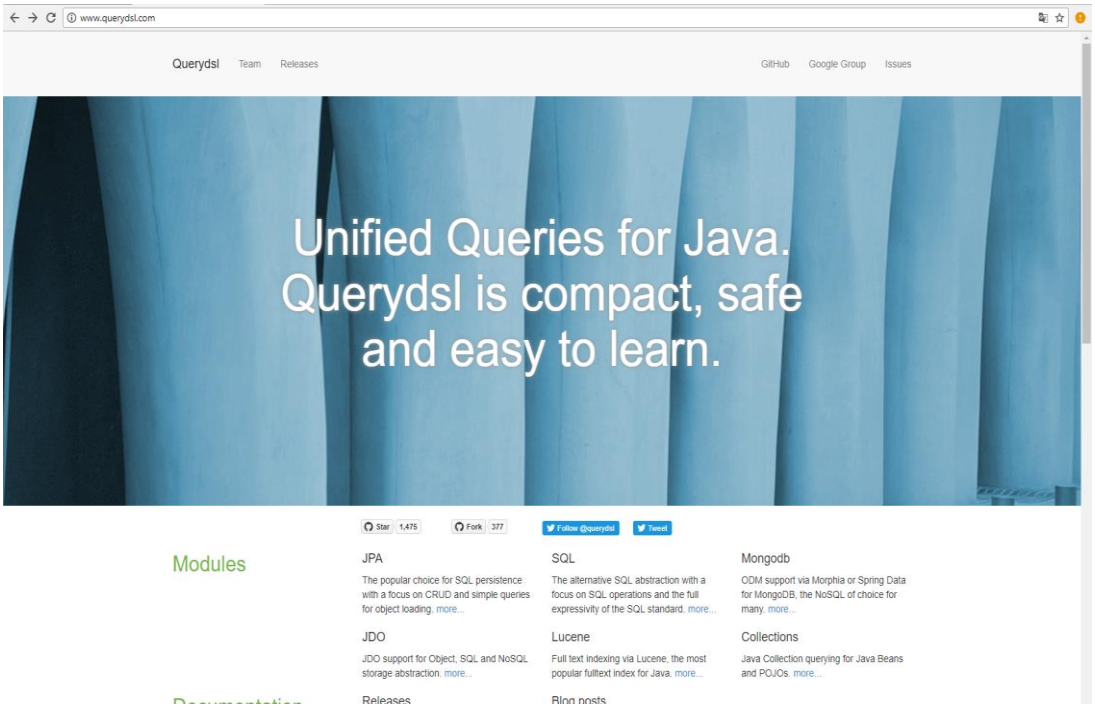
En el servicio es donde se ha de utilizar la clase **CourseConverter.java** que hay que inyectar.

Al devolver el listado no va a devolver un listado de cursos entidades, sino un listado de cursos de objetos de la clase **CourseModel.java**.

Introducción a QueryDSL

QueryDSL es una librería para realizar consultas sql mediante java puro.

Hay que visitar la página web www.querydsl.com, allí el módulo que vamos a utilizar es el de JPA. Nos llevará a la página GitHub donde vienen las dependencias y los pluggins que hay que añadir al fichero pom.xml.



Dependencias:

```
<dependency>
	<groupId>com.querydsl</groupId>
	<artifactId>querydsl-jpa</artifactId>
</dependency>
<dependency>
	<groupId>org.slf4j</groupId>
	<artifactId>slf4j-log4j12</artifactId>
</dependency>
```

Pluggins:

```
<plugin>
		<groupId>com.mysema.maven</groupId>
		<artifactId>apt-maven-plugin</artifactId>
		<version>1.1.3</version>
			<executions>
				<execution>
					<goals>
```

```
                                    <goal>process</goal>
                                </goals>
                                <configuration>
                                    <outputDirectory>target/generated-
                                    sources/java</outputDirectory>
                                    <processor>com.querydsl.apt.jpa.JPAAnnotati
                                    onProcessor</processor>
                                </configuration>
                            </execution>
                        </executions>
                    <dependencies>
                        <dependency>
                            <groupId>com.querydsl</groupId>
                            <artifactId>querydsl-apt</artifactId>
                            <version>${querydsl.version}</version>
                        </dependency>
                    </dependencies>
</plugin>
```

Ejemplo de consulta QueryDSL

```
@Repository("queryDSLExampleRepo")
public class QueryDSLExampleRepo {

        private QCourse qCourse=QCourse.course;

        //Obtenemos el entitymanager

        @PersistenceContext
        private EntityManager em;

        public void find(boolean exist) {
                JPAQuery<Course> query=new JPAQuery<Course>(em);

                BooleanBuilder predicateBuilder=new
                BooleanBuilder(qCourse.description.endsWith("0P"));


                if(exist) {
                        Predicate predicate2=qCourse.id.eq(23);
                        predicateBuilder.and(predicate2);
                }else {
                        Predicate predicate3=qCourse.name.endsWith("OP");
                        predicateBuilder.or(predicate3);
                        }

        query.select(qCourse).from(qCourse).where(predicateBuilder).fetchOne();

        //List <Course> courses =
        query.select(qCourse).from(qCourse).where(qCourse.hours.between(10,
        30)).fetch();

        }

}
```