Spring

Comienzo del ejercicio práctico	31-10-2023

1 Spring y desarrollo de aplicaciones Web

El marco de trabajo Spring es un *framework* maduro, potente y muy flexible para la construcción de aplicaciones Web en Java. Uno de los beneficios principales que nos aporta Spring es que se ocupa de los aspectos de bajo nivel en la construcción de estas aplicaciones y nos permite concentranos en la lógica de negocio de las aplicaciones.

Otro punto fuerte de Spring es que, si bien se le puede considerar ya bastante maduro y está bien definido, está siendo mantenido activamente por una comunidad de desarrollo de aplicaciones Web con mucho éxito. Esto mantiene al *framework* Spring bastante actualizado y alineado con el ecosistema de Java en este momento.

2 Anotaciones usadas en los *beans* de Spring

Hay varias formas de configurar *beans* en un contenedor de Spring:

- Usando la configuración XML
- Usando la anotación `@Bean` en una clase de configuración
- Podemos tambiérn marcar la clase con una de las anotaciones del paquete `org.springframework.stereotype` y dejar el resto para el escaneo del framework

2.1 Escaneo de componentes

`@ComponentScan` configura qué paquetes escanear en busca de clases con anotaciones de configuración. Podemos especificar los nombres de los paquetes base directamente con uno de los `basePackages`:

```java
package com.dss.spring.first.di.config;
import java.util.Date;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.beans.factory.annotation.Qualifier;

@Configuration
@ComponentScan(basePackages = { "com.dss.spring.first.di.model" })
public class Config {
    @Bean
    public Long getId() {
        return Long.valueOf(0);
    }
    @Bean
    @Qualifier("summary")
    public String getSummary() {
        return "Spring: prueba de Inyeccion de Dependencias";
    }
    @Bean
    @Qualifier("description")
    public String getDescription() {
        return "Spring: prueba de Inyeccion de Dependencias y todo lo demas";
    }
    @Bean
    public Boolean isDone() {
        return Boolean.FALSE;
    }
    @Bean
    public Date getDueDate() {
        return new Date();
    }
}
```

Si no se especifica ningún argumento, el escaneo ocurre desde el mismo paquete donde está presente la clase anotada `@ComponentScan`.

De forma predeterminada, los *beans*-instancia de una clase anotada como un componente tienen el mismo nombre que el nombre de la clase con una inicial en minúscula, o bien podemos especificar un nombre diferente usando el argumento de valor opcional.

2.2 `@Component`

Se trata de una anotación a nivel de Clase; durante el escaneo de componentes, Spring detecta automáticamente las clases que hayan sido anotadas con esta palabra:

```java
package com.dss.spring.first.di.model;
import java.util.Date;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;
import org.springframework.beans.factory.annotation.Qualifier;
@Component
public class Todo implements ITodo {
    public final long id;
    private String summary;
    private String description;
    private Boolean done;
    private Date dueDate;
```

```
13      public Todo() {
14          this(-1);
15      }
16      public Todo(long i) {
17         this(i, "");
18      }
19      @Autowired
20      public Todo(long i,  @Qualifier("summary")String summary) {
21        this.id = i;
22        this.summary = summary;
23      }
24      @Override
25      public long getId() {
26         return id;
27      }
28          @Override
29      public String getSummary() {
30         return summary;
31      }
32      @Override
33      public void setSummary(String summary) {
34         this.summary= summary;
35      }
36      @Override
37      public String getDescription() {
38          return description;
39      }
40      @Autowired
41      @Qualifier("description")
42      @Override
43      public void setDescription(String description) {
44          this.description = description;
45      }
46      @Override
47      public boolean isDone() {
48         return done;
49      }
50      @Autowired
51      @Override
52      public void setDone(boolean isDone) {
53         this.done= isDone;
54      }
55      @Override
56      public Date getDueDate() {
57         return dueDate;
58      }
59      @Autowired
60      @Override
61      public void setDueDate(Date dueDate) {
62         this.dueDate= dueDate;
63      }
64      @Override
65      public int hashCode() {
66         final int prime = 31;
67         int result = 1;
68         result = prime * result + (int) (id ^ (id >>> 32));
69         return result;
70      }
71          @Override
72      public boolean equals(Object obj) {
73         if (this == obj)
74            return true;
75         if (obj == null)
76            return false;
77         if (getClass() != obj.getClass())
78            return false;
79         Todo other = (Todo) obj;
80         if (id != other.id)
81            return false;
82         return true;
83      }
```

```
84      @Override
85      public String toString() {
86          return "Todo [id=" + id + ", summary=" + summary + ", description="+description+"]";
87      }
88      @Override
89      public Todo copy() {
90          Todo todo = new Todo(id, summary);
91          todo.setDone(this.isDone());
92          todo.setDueDate(this.getDueDate());
93          todo.setDescription(getDescription());
94          return todo;
95      }
```

2.3 `@Repository`

Las clases `DAO` o `Repository` representan a la capa de acceso a una base de datos en las aplicaciones:

```
1   package com.dss.spring.data.rest2;
2   import org.springframework.data.jpa.repository.JpaRepository;
3   import org.springframework.data.rest.core.annotation.RepositoryRestResource;
4   @RepositoryRestResource(collectionResourceRel="tasks",path="tasks")
5   public interface TodoRepository extends JpaRepository<Todo, Long>{
6   }
```

Una ventaja de usar esta anotación es que tiene habilitada la traducción automática de excepciones relacionadas con la persistencia de entidades. Lo cual es muy útil cuando se usa un marco de persistencia como *Hibernate*.

Se pueden utilizar repositorios especilizados, como en el ejemplo anterior, que se usa un repositorio CRUD para almacenar los datos de servicios REST.

2.4 `@Controller`

Se trata de una anotación a nivel de Clase que le indica al marco de trabajo Spring que esta clase sirve como un controlador del "patrón" MVC:

```
1   package com.dss.spring.data.rest2;
2   import org.springframework.stereotype.Controller;
3   import org.springframework.web.bind.annotation.RequestMapping;
4   import org.springframework.web.bind.annotation.GetMapping;
5   @Controller
6   public class RootUriController {
7     @RequestMapping(value = "/index")
8     public String index() {
9       return "index";
10    }
11  }
```

Lo anterior es el controlador para que una UI de cliente (por ejemplo una aplicación Java o una página JSP, etc.) pueda hacer que se le envíen páginas Web a su navegador. En este caso la página que se solicita es : `index.html`, que ha de estar ubicada en la carpeta *templates* del proyecto.

2.5 `@Configuration`

Las clases con esta anotación pueden contener métodos definidos como *beans*:

```
package com.dss.spring.first.di.config;
import java.util.Date;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.beans.factory.annotation.Qualifier;
@Configuration
@ComponentScan(basePackages = { "com.dss.spring.first.di.model" })
public class Config {
  @Bean
  @Qualifier("summary")
  public String getSummary() {
    return "Spring: prueba de Inyeccion de Dependencias";
  }
  @Bean
  @Qualifier("description")
  public String getDescription() {
    return "Spring: prueba de Inyeccion de Dependencias y todo lo demas";
  }
  @Bean
  public Boolean isDone() {
    return Boolean.FALSE;
  }
  @Bean
  public Date getDueDate() {
    return new Date();
  }
}
```

Interface `ITodo`:

```
import java.util.Date;
public interface ITodo {
  long getId();
  String getSummary();
  void setSummary(String summary);
  String getDescription();
  void setDescription(String description);
  boolean isDone();
  void setDone(boolean isDone);
  Date getDueDate();
  void setDueDate(Date dueDate);
  ITodo copy();
}
```

Ejercicio 1:Programar con Spring una aplicación Java que *"instancie"* varios *beans* conformes con la interfaz `ITodo` y que muestre en la pantalla del cliente su contenido:

```
package com.example.demo;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import com.dss.spring.first.di.config.Config;
import com.dss.spring.first.di.model.ITodo;
public class Application {
  public static void main(String[] args) {
  //Hay que crearse un contexto
AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(Config.class);
  //FALTA CODIGO
  //los beans se consiguen con getBean(interfaz.class), que es un metodo de la entidad contexto
  context.close();
  }
}
```

2.6 `@Qualifier` , `@Autowired` , `@Primary`

2.6.1 `@Autowired`

La anotación @Autowired es una excelente manera de hacer explícito a Spring que tenemos que *inyectar* una nueva dependencia. Aunque esta anotación es muy útil, hay casos de uso en los que esta anotación por sí sola no es suficiente para que Spring comprenda qué *bean* inyectar, ya que Spring resuelve las entradas *autowired* ("conectadas automáticamente") de un contenedor localizándolas por el tipo. Así en el caso del contenedor `Config` las entradas correspondientes a los beans `public String getSummary()` y a `public String getDescription()` devuelven ambos un tipo String y son, por tanto, ambiguos para Spring. Si hay más de un *bean* del mismo tipo disponible en el contenedor, Spring levantará la excepción `NoUniqueBeanDefinitionException`, lo que indica que hay más de un bean disponible para *conectarlo automáticamente* (autowired). Para evitar este problema, existen varias soluciones; la anotación `@Qualifier` es una de ellas.

2.6.2 `@Qualifier`

Podemos eliminar el problema de qué *bean* debe inyectarse incluyendo la anotación `@Qualifier` para indicar qué *bean* queremos usar:

```
1  public Todo(long i, @Qualifier("summary") String summary) {
2    this.id = i;
3    this.summary = summary;
4  }
```

En el caso anterior, indicamos `@Qualifier("summary")` para indicar que al llamar al constructor `Todo` hay que inyectar un *bean* del tipo *summary*, no un *bean* del tipo *description*.

2.6.3 `@Primary`

Existe otra anotación llamada `@Primary`, que podemos usar para decidir qué *bean* inyectar cuando existe ambigüedad con respecto a la inyección de una dependencia en nuestras aplicaciones. La anotación `@Primary` define una preferencia cuando están presentes varios *beans* del mismo tipo.

El *bean* asociado con la anotación `@Primary` se utilizará siempre que no se indique otra cosa, por ejemplo, utilizando la anotación `@Qualifier`.

3 Proyecto Spring Data JPA

Hay que crearse ahora un nuevo proyecto, después de descargarnos e instalar *Spring Tool Suite 4* (`https://spring.io/tools`).

En la entrada del menú *File* de *Spring Tool Suite* (STS): seleccionar *New* ⇒ *Spring Starter Project*. Aparecerá un asistente y durante sus ejecución debemos seleccionar la siguientes librerías: *Lombok, JPA, H2*, para continuar.

3.1 Entidad JPA

Para conseguir que las *"instancias"* de Todo se almacenen en la base de datos H2, tendremos que utilizar las anotaciones ya estudiadas de JPA: `@Entity`, `@Id` y `@GeneratedValue`. Para evitar tener que escribir código repetitivo, podemos utilizar la anotación `@Data`, que generará automáticamente los métodos *getters* y *setters*:

```
1   import javax.persistence.Entity;
2   import javax.persistence.GeneratedValue;
3   import javax.persistence.GenerationType;
4   import javax.persistence.Id;
5   import lombok.Data;
6   @Entity
7   @Data
8   public class Todo {
9    @Id
10   @GeneratedValue(strategy = GenerationType.AUTO)
11   private long id;
12   private String summary;
13   private String description;
14   private Boolean done;
15   private Date dueDate;
16   public Todo() {
17   }
18   public Todo(String summary) {
19     this.summary = summary;
20   }
21   public void setId(long id) {
22   }
23   public Todo copy() {
24     Todo todo = new Todo(summary);
25     todo.setDone(getDone());
26     todo.setDueDate(getDueDate());
27     todo.setDescription(getDescription());
28     return todo;
29     }
30  }
```

3.2 Repositorio para DAO

Cuando se usa *Spring Data JPA*, no es necesario escribir todo el código repetitivo que, generalmente, hemos de programar en *Jersey + JPA* para proporcionar una funcionalidad CRUD común a un DAO. Ahora sólo tenemos que usar una interfaz compatible con `CrudRepository`, que heredaremos (`extends`), y todo lo demás se nos proporcionará automáticamente:

```
1  import org.springframework.data.jpa.repository.JpaRepository;
2  import org.springframework.data.rest.core.annotation.RepositoryRestResource;
3  @RepositoryRestResource(collectionResourceRel="tasks",path="tasks")
4  public interface TodoRepository extends JpaRepository <Todo, Long>{
5  }
```

Un repositorio CRUD como el anteriormente programado contiene los métodos usuales que utilizamos con los DAO y que podemos utilizar automáticamente para conseguir la persistencia de las instancias de `Todo`: `Interface CrudRepository<T, ID extends Serializable>`:

- save(S)<S extends T>: S
- save(Iterable<S>)<S extends T>: Iterable <S>
- findOne(ID): T
- exists(ID): boolean
- findAll(): Iterable<T>
- findAll(Iterable<ID>): Iterable<T>
- count():long
- delete(ID): void
- delete(T): void
- delete(Iterable<? extends T>): void
- deleteAll(): void

Se programará como una *interfaz* dentro del proyecto Java y en el ejemplo aparece con el nombre: `TodoRepository.java`

3.3 Inicializador del Servlet

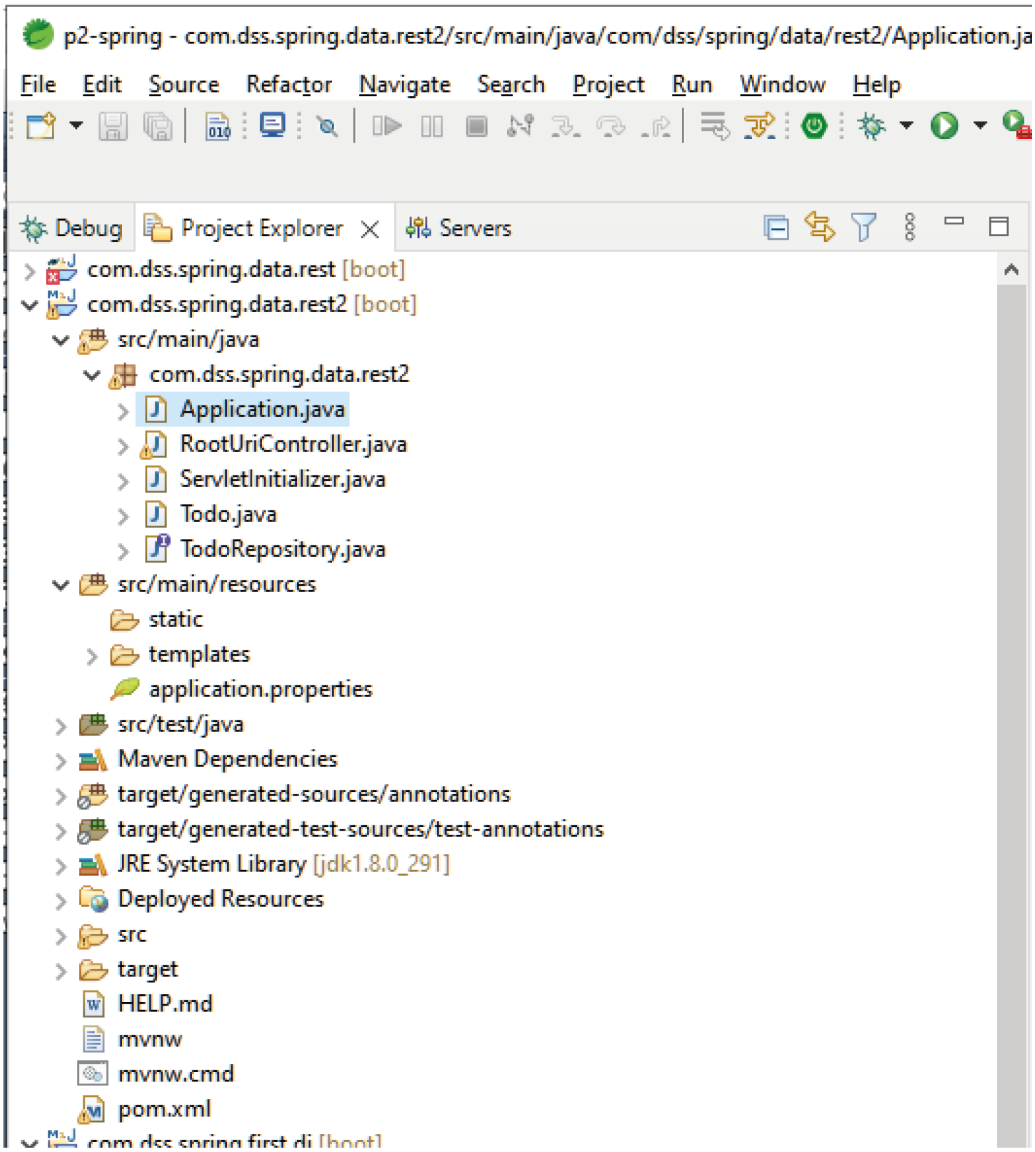
Como en este ejercicio se va a desplegar un servlet, que se encargará de implementar la funcionalidad CRUD en el servidor, entonces hay que inicializarlo en una clase independiente: `ServletInitializer` que extiende a `SpringBootServletInitializer`:

```
1  import org.springframework.boot.builder.SpringApplicationBuilder;
2  import org.springframework.boot.web.servlet.support.SpringBootServletInitializer;
3  public class ServletInitializer extends SpringBootServletInitializer{
4    @Override
5    protected SpringApplicationBuilder configure(SpringApplicationBuilder application) {
6      return application.sources(Application.class);
7    }
8  }
```

3.4 *Command line runner*

Pr último, para poder trabajar con el repositorio que nos hemos creado anteriormente necesitamos crear un objeto `CommandLineRunner` dentro de la clase principal (`Application.java` de nuestro proyecto.

```
1   import org.springframework.boot.CommandLineRunner;
2   import org.springframework.boot.SpringApplication;
3   import org.springframework.boot.autoconfigure.SpringBootApplication;
4   import org.springframework.context.annotation.Bean;
5   import org.springframework.http.ResponseEntity;
6   import org.springframework.web.client.RestTemplate;
7   @SpringBootApplication
8   public class Application {
9     public static void main(String[] args) {
10      SpringApplication.run(Application.class, args);
11        }
12    @Bean
13    public CommandLineRunner jpaSample(TodoRepository todoRepo) {
14      return (args) -> {
15        //Almacenar los 2 "instancias" de Todo en la base de datos H2
16    todoRepo.save(new Todo("Test"));
17    Todo todo = new Todo("Test detallado");
18    Date date = new Date();
19    todo.setDueDate(date);
20    todo.setDescription("Descripcion detallada");
21    todoRepo.save(todo);
22    RestTemplate restTemplate = new RestTemplate();
23    //Ahora los vamos a obtener del servidor REST
24    Todo firstTodo = restTemplate.getForObject("http://localhost:8080/rest/tasks/1", Todo.class);
25    Todo secondTodo = restTemplate.getForObject("http://localhost:8080/rest/tasks/2", Todo.class);
26    System.out.println(firstTodo);
27    System.out.println(secondTodo);
28    //Ejemplo de POST al servidor REST
29    Todo newTodo = new Todo("Nuevo Todo");
30    newTodo.setDescription("Todo anadido por la API rest");
31    newTodo.setDone(true);
32    //Envio y validacion
33    ResponseEntity<Todo> postForEntity =
34    restTemplate.postForEntity("http://localhost:8080/rest/tasks/", newTodo, Todo.class);
35    System.out.println(postForEntity);
36    };
37  }
38 }
```



Ejercicio 2:Programar con Spring una aplicación Web REST Java que *"instancie"* a la entidad `Todo` varias veces, los almacene en el repositorio `TodoRepository`, envíe un POST al servidor con alguno de estos "objetos" `Todo` y que muestre en la pantalla del cliente los envíos realizados de una manera similar a:

```
Todo [id=0, summary=Test, description=null]

Todo [id=0, summary=Test detallado, description=Descripcion detallada]

<201,Todo [id=0, summary=Nuevo Todo, description=Todo anadido por la API rest],[Vary:"Origin",
"Access-Control-Request-Method", "Access-Control-Request-Headers",
Location:"http://localhost:8080/rest/tasks/3", Content-Type:"application/json",
Transfer-Encoding:"chunked", Date:"Wed, 10 Nov 2021 17:35:45 GMT",
Keep-Alive:"timeout=60", Connection:"keep-alive"]>
```

Para que funcione el selector '*rest*' en la indirección (`http://localhost:8080/rest/tasks/`) que se envía a la API REST, hay que escribir la propiedad: `spring.data.rest.base-path=/rest` dentro de *application.properties*, que se encuentra dentro de la carpeta: *src/main/resources/templates/* del proyecto.

Créditos

- `https://www.vogella.com/tutorials/Spring/article.html`
- `https://www.baeldung.com/spring-tutorial`
- `https://spring.io/guides`
- `https://docs.spring.io/spring-framework/docs/current/reference/html/overview.html#overview-modules`
- `https://spring.io/projects/spring-framework`
- Descargas de STS en : `https://spring.io/tools`