Spring Framework

Spring MVC. Los servicios y componentes

EE. SS. Mª AUXILIADORA

2º DAM
Autor: Manuel Torres Molina
ACCESO A DATOS

Spring Framework

Spring MVC. Los servicios y componentes

Creación de un componente

Dentro de nuestro proyecto vamos a crear un nuevo paquete llamado **component** dentro del paquete de nuestra aplicación que hay en la carpeta **src/main/java**. Ahí crearemos una clase llamada **ExampleComponent.java** que será nuestro componente y le añadiremos la anotación **@Component** para identificarlo como tal.

```
@Component("exampleComponent")
public class ExampleComponent {

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

	public void sayHello() {

		LOG.info("HELLO FROM EXAMPLECOMPONENT");

	}
}
```

A continuación, nos vamos a nuestra clase ExampleController.java para inyectar este componente mediante la anotación **@Autowired** y **@Qualifier** para poder utilizar su método **sayHello()**.

Comprobaremos en el Log de consola como no hemos tenido que instanciar la clase **ExampleComponent.java** para poder utilizar su método.

```
@Autowired  //Indica a Spring que vamos a inyectar un componente que está en memoria
@Qualifier("exampleComponent") //Indica a Spring el nombre del Bean que está en
memoria
private ExampleComponent exampleComponent; //Declaramos el componente

	@GetMapping ("/exampleString")

	public String exampleString(Model model) {
		exampleComponent.sayHello(); //Llamamos al método del componente sin
		instanciar el componente
		model.addAttribute("people", getPeople());
		return EXAMPLE_VIEW;
	}
```

Si ponemos en nuestro navegador http://localhost:8080/example/exampleString nos saldrá por consola:

```
2017-08-02 09:09:04.130  INFO 12456 --- [nio-8080-exec-1]
com.udemy.component.ExampleComponent     : HELLO FROM EXAMPLECOMPONENT
```

Controlando el tiempo de las peticiones

Para calcular el tiempo de las peticiones hay que crear un componente llamado **RequestTimeIntercept.java** dentro de la carpeta **component** de nuestro proyecto, con el siguiente código:

```
@Component("requestTimeInterceptor")
public class RequestTimeInterceptor extends HandlerInterceptorAdapter{

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

	@Override
	public boolean preHandle(HttpServletRequest request, HttpServletResponse
	response, Object handler) //Método que se ejecuta antes de entrar en el
	método del controlador
			throws Exception {
			request.setAttribute("startTime", System.currentTimeMillis());
				return true;
	}

	@Override
	public void afterCompletion(HttpServletRequest request, HttpServletResponse
	response, Object handler, Exception ex) //Método que se ejecuta antes de
	lanzar la vista en el navegador
			throws Exception {
		long startTime=(long) request.getAttribute("startTime");
		LOG.info("--REQUEST URL: '"+request.getRequestURI().toString()+ "' --
		-TOTAL TIME: '" + (System.currentTimeMillis()-startTime));

	}

}
```

Sobreescribimos los métodos **preHandle** y **afterCompletion**.

Para que funcione este componente tendremos que darlo de alta en una clase **WebMvcConfiguration.java** que creamos en un paquete nuevo llamado **configuration** dentro de la ruta **src/main/java** y del paquete de nuestro proyecto. Su código es el siguiente:

```
@Configuration
public class WebMvcConfiguration extends WebMvcConfigurerAdapter{

	@Autowired
	@Qualifier("requestTimeInterceptor")
	private RequestTimeInterceptor requestTimeInterceptor;
	@Override
	public void addInterceptors(InterceptorRegistry registry) {

		registry.addInterceptor(requestTimeInterceptor);


		 }
}
```

Hemos identificado el componente, lo hemos llamado y hemos añadido el método sobreescrito **addInterceptors** para poder dar de alta este componente.

Ahora si realizamos peticiones en nuestro navegador, funcionará los métodos del componente creado y podremos comprobarlo por consola con las siguientes salidas:

```
2017-08-02 10:12:00.662  INFO 252 --- [nio-8080-exec-2]
c.u.component.RequestTimeInterceptor     : --REQUEST URL: '/example3/showForm' ---
TOTAL TIME: '13

2017-08-02 10:12:18.352  INFO 252 --- [nio-8080-exec-3]
c.u.component.RequestTimeInterceptor     : --REQUEST URL: '/example3/addperson' --
-TOTAL TIME: '9
```

Creación de un servicio

Para crear un servicio con Spring lo primero que tenemos que hacer es dentro de la carpeta **service** de **src/main/java** y el paquete de nuestro proyecto, crear una interfaz **ExampleService.java**.

```
public interface ExampleService {

	public abstract List<Person> getListPeople();
}
```

A continuación, crear una clase llamada **ExampleServiceImpl.java** que implemente esa interfaz dentro de un paquete **impl**, que está situado dentro del paquete **service**, a la que le anotamos con el nombre del **Bean** que le queramos dar mediante **@Service**.

```
@Service("exampleService")
public class ExampleServiceImpl implements ExampleService{

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

	public List<Person> getListPeople() {
		List<Person> people=new ArrayList<Person>();
		people.add(new Person("Sarabel", 18));
		people.add(new Person("David", 3));
		people.add(new Person("Rubén", 1));
		LOG.info("HELLO FROM SERVICE");
		return people;
	}
}
```

Y por último en la clase donde queramos utilizar ese servicio mediante **@Autowired** y **@Qualifier("Nombre asignado al servicio")** y la declaración de esa clase, ya podemos usarla sin problemas dentro de los métodos de esa clase.

```
@Controller
@RequestMapping("/example")
public class ExampleController {
```

```
	public static final String EXAMPLE_VIEW="example"; //Vista a retornar
	example.html

	@Autowired
	@Qualifier("exampleService")
	private ExampleService exampleService;

	@Autowired  //Indica a Spring que vamos a inyectar un componente que está en
	memoria
	@Qualifier("exampleComponent")  //Indica a Spring el nombre del Bean que
	está en memoria
	private ExampleComponent exampleComponent;

	//Primera forma
	@GetMapping ("/exampleString")
	public String exampleString(Model model) {
		exampleComponent.sayHello(); //Llamamos al método del componente sin
		instanciar el componente
		model.addAttribute("people",exampleService.getListPeople());
		return EXAMPLE_VIEW;
	}

	//Segunda forma
	@GetMapping ("/exampleMAV")
	public ModelAndView exampleMAV() {
		ModelAndView mav=new ModelAndView(EXAMPLE_VIEW);
		mav.addObject("people", exampleService.getListPeople());
		return mav;
	}
}
```

Al ejecutar por ejemplo en el navegador la ruta http://localhost:8080/example/exampleMAV podremos comprobar cómo se llama al servicio en el log de la consola:

```
2017-08-02 10:36:36.541  INFO 7544 --- [nio-8080-exec-1]
c.udemy.service.impl.ExampleServiceImpl  : HELLO FROM SERVICE
```

Validando datos de formulario

Lo primero que necesita Spring para poder validar datos en formularios es que el modelo que le vayamos a pasar tenga ciertas anotaciones en sus atributos.

```
public class Person {

	@NotNull
	@Size(min=2, max=6)
	private String name;

	@NotNull
	@Size(min=18)
	private int age;

	...
}
```

Se añade en la clase **Example3Controller.java** lo siguiente en el método que controla si se van a añadir o no los datos enviados por el formulario y que utiliza el modelo **Person**. Lo que se añade es la anotación **@Valid** y la clase que controla esa validación que es **BindingResult**.

Si tuviera algún error de validación devolvería el propio formulario con unas anotaciones que indicaremos ahora en la plantilla del formulario, si no tuviera errores de validación devolvería la vista con el resultado de la inserción.

Método modificado de la clase **Example3Controller.java**:

```
public ModelAndView addPerson(@Valid @ModelAttribute("person") Person person,
BindingResult bindingResult) {
		ModelAndView mav=new ModelAndView();
		if(bindingResult.hasErrors()) {
			mav.setViewName(FORM_VIEW);
		}else {
			mav.setViewName(RESULT_VIEW);
			mav.addObject("person", person);
		}

		return mav;
	}
```

Plantilla **form.html**

```
<!DOCTYPE html>
<html>
<head>
	<meta charset="ISO-8859-1"/>
	<title>FORM</title>
</head>
<body>
	<form action="#" th:action="@{/example3/addperson}" th:object="${person}"
	method="post">
	<p> NAME: <input type="text" th:field="*{name}"/> </p>
	<p th:if="${#fields.hasErrors('name')}" th:errors="*{name}">Name has
	errors</p>
	<p> AGE: <input type="number" th:field="*{age}"/> </p>
	<p th:if="${#fields.hasErrors('age')}" th:errors="*{age}">Age has
	errors</p>
	<p><input type="submit" value="Submit"/></p>

	</form>
</body>
</html>
```