Aplicaciones AJAX y REST con Spring MVC

Índice

En la sesión anterior vimos cómo implementar aplicaciones web "clásicas", es decir, aquellas en las que cada "pantalla" de nuestra aplicación se implementa en un HTML distinto y cada comunicación con el servidor implica un cambio de página (y de "pantalla"). No es necesario decir que la gran mayoría de aplicaciones web actuales no son así. Casi todas usan AJAX y Javascript en el cliente de manera intensiva, lo que nos permite comunicarnos con el servidor sin cambiar de página y también cambiar dinámicamente la interfaz sin movernos a otro HTML.

Un paso más allá son las aplicaciones REST, en las que (entre otras cosas) el cliente no recibe el HTML sino directamente los datos en un formato estándar (JSON/XML) y los "pinta" él mismo (o los formatea en HTML para que los "pinte" el navegador) . Esta filosofía permite diseñar capas web multi-dispositivo (escritorio/web/móviles).

Veremos en esta sesión una introducción a las facilidades que nos da Spring para implementar estos dos tipos de aplicaciones.

1. AJAX con Spring

En una aplicación AJAX lo que necesitamos por parte del servidor es que nos permita intercambiar información hacia/desde el cliente fácilmente. En su variante más sencilla esa información sería simplemente texto o pequeños fragmentos de HTML. En casos más complejos serían objetos Java serializados a través de HTTP en formato JSON o XML. Vamos a ver qué funcionalidades nos ofrece Spring para implementar esto, planteando varios casos de uso típicos en AJAX.

1.1. Caso de uso 1: respuesta del servidor como texto/fragmento de HTML

En este caso, el cliente hace una petición AJAX y el servidor responde con un fragmento de texto plano o de HTML que el cliente mostrará en la posición adecuada de la página actual. Por ejemplo, el típico caso del formulario de registro en que cuando llenamos el campo de login queremos ver si está disponible, antes de rellenar los siguientes campos. Cuando el foco de teclado sale del campo de login, se hace una petición AJAX al servidor, que nos devolverá simplemente un mensaje indicando si está disponible o no.

La página HTML del cliente con el javascript que hace la petición AJAX y recibe la respuesta podría ser algo como:

AJAX y Javascript

Mostraremos aquí el código javascript del cliente para tener el ejemplo completo, aunque no podemos ver con detalle cómo funciona al no ser materia directa del curso. Usamos la librería jQuery en los ejemplos para simplificar al máximo el código.

```
<form id="registro" action="#">
        Login: <input type="text" name="login" id="campo_login">
```

```
                <span id="mensaje"></span><br>
        Password: <input type="password" name="password"> <br>
        Nombre y apellidos: <input type="text" name="nombre"> <br>
        <input type="submit" value="registrar">
</form>
<script type="text/javascript">
        $('#campo_login').blur(
                function() {
                        $('#mensaje').load('loginDisponible.do',
"login="+$('#campo_login').val())
                }
        )
</script>
```

Del código jQuery anterior baste decir que cuando el foco de teclado se va (evento 'blur') del campo con id "campo_login" es cuando queremos disparar la petición AJAX. El método load() de jQuery lanza una petición AJAX a una determinada url con determinados parámetros y coloca la respuesta en la etiqueta HTML especificada (en este caso la de id "mensaje", un span que tenemos vacío y preparado para mostrar el mensaje).

El código Spring en el servidor, que respondería a la petición AJAX, sería el siguiente:

```
@Controller
public class UsuarioController {
        @Autowired
        private IUsuarioBO ubo;

        @RequestMapping("/loginDisponible.do")
        public @ResponseBody String loginDisponible(@RequestParam("login")
String login) {
                if (ubo.getUsuario(login)==null)
                        return "login disponible";
                else
                        return "login <strong>no</strong> disponible";
        }

}
```

La única diferencia con lo visto en la sesión anterior es que el valor de retorno del método no debe ser interpretado por Spring como el nombre lógico de una vista. Debe ser el contenido de la respuesta que se envía al cliente. Esto lo conseguimos anotando el valor de retorno del método con `@ResponseBody`. Cuando el valor de retorno es un String, como en este caso, simplemente se envía el texto correspondiente en la respuesta HTTP. Como veremos, si es un objeto Java cualquiera se serializará automáticamente.

Evidentemente no es una muy buena práctica tener "empotrados" en el código Java directamente los mensajes que queremos mostrar al usuario, pero este trata de ser un ejemplo sencillo. En un caso más realista usaríamos el soporte de internacionalización de Spring para externalizar e internacionalizar los mensajes. O quizá sería el propio javascript el que mostraría el mensaje adecuado.

1.2. Caso de uso 2: respuesta del servidor como objeto serializado

Continuando con el ejemplo anterior, supongamos que si el login no está disponible,

queremos, además de saberlo, obtener como sugerencia algunos logins parecidos que sí estén disponibles, como se hace en muchos sitios web.

En lugar de enviarle al cliente simplemente un mensaje, le enviaremos un objeto con un campo booleano que indique si el login está disponible o no, y una lista de Strings con las sugerencias. En caso de que esté disponible, no habría sugerencias. La clase Java que encapsularía esta información desde el lado del servidor sería algo como lo siguiente (no se muestran constructores, getters o setters, solo las propiedades)

```
public class InfoLogin {
        private boolean disponible;
        private List<String> sugerencias;

        ...
}
```

Y el método de Spring que respondería a la petición HTTP ahora devolvería un objeto `InfoLogin`.

```
@Controller
public class UsuarioController {
        @Autowired
        private IUsuarioBO ubo;

        @RequestMapping("/loginDisponible.do")
        public @ResponseBody InfoLogin loginDisponible(
                                                @RequestParam("login") String
login) {
                if (ubo.getUsuario(login)==null)
                        //Si está disponible, no hacen falta sugerencias
                        return new InfoLogin(true, null);
                else
                        //si no lo está, generamos las sugerencias con la
ayuda del IUsuarioBO
                        return new InfoLogin(false,
ubo.generarSugerencias(login));
        }

}
```

Por lo demás, como se ve, a nivel de API de Spring no habría cambios. Automáticamente se serializará el objeto al formato adecuado. Por defecto, lo más sencillo en Spring es generar JSON. Si usamos Maven, basta con incluir en el proyecto la dependencia de la librería Jackson, una librería Java para convertir a/desde JSON que no es propia de Spring, pero con la que el framework está preparado para integrarse:

```
<dependency>
    <groupId>org.codehaus.jackson</groupId>
    <artifactId>jackson-mapper-asl</artifactId>
    <version>1.9.11</version>
</dependency>
```

En el lado del cliente, el Javascript debería obtener el objeto enviado desde Spring (sencillo si lo que se envía es JSON) y mostrar las sugerencias en el HTML.

Simplificando, podría ser algo como lo siguiente:

```
<form id="registro" action="#">
    Login: <input type="text" name="login" id="campo_login">
           <span id="mensaje"></span><br>
    Password: <input type="password" name="password"> <br>
    Nombre y apellidos: <input type="text" name="nombre"> <br>
    <input type="submit" value="registrar">
</form>
<script type="text/javascript">
$('#campo_login').blur(
        function() {
                $.getJSON('loginDisponible.do',
                                "login="+$('#campo_login').val(),
                                function(obj) {
                                        var mens;
                                        if (obj.disponible)
                                                mens = "El login está
disponible";
                                        else {
                                                mens = "El login no está
disponible. Sugerencias: ";
                                                for (var i=0;
i<obj.sugerencias.length; i++) {
                                                        mens +=
obj.sugerencias[i] + " ";
                                                }
                                        }
                                        $('#mensaje').html(mens);
                                }
                )
        }
)
</script>
```

Si quisiéramos serializar el objeto en formato XML, bastaría con anotarlo con anotaciones JAXB, como se vio en las sesiones de servicios REST del módulo de Componentes Web (solo se muestran los getters relevantes por estar anotados)

```
@XmlRootElement
public class InfoLogin {
        private boolean disponible;
        private List<String> sugerencias;

        @XmlElement
        public boolean isDisponible() {
                return disponible;
    }

        @XmlElementWrapper(name="sugerencias")
        @XmlElement(name="sugerencia")
        public List<String> getSugerencias() {
                return sugerencias;
        }

        ...
}
```

1.3. Caso de uso 3: enviar objetos desde el cliente

En AJAX lo más habitual es que el cliente envíe los datos a través de un formulario HTML. Ya vimos en la primera sesión de Spring MVC cómo tratar ese caso de uso, recordemos que los datos se podían "empaquetar" automáticamente en un objeto Java y, como ya veremos, validar declarativamente con JSR303. Pero también podríamos hacer que el cliente envíe al servidor un objeto serializado en JSON o XML. Este objeto se envía entonces en el cuerpo de la petición HTTP del cliente y el trabajo de Spring es deserializarlo y "transformarlo" a objeto Java. En el método del controller que responda a la petición simplemente anotamos el parámetro que queremos "vincular" al objeto con `@RequestBody`.

Continuando con el ejemplo del registro de usuarios, supongamos que queremos enviar desde el cliente el nuevo usuario en formato JSON (por ejemplo, porque usamos un cliente de escritorio). Desde el lado del servidor bastaría con usar `@RequestBody`:

```
@RequestMapping("/altaUsuario.do")
public void altaUsuario(@RequestBody Usuario usuario) {
        ...
}
```

Para completar el ejemplo, mostraremos el código correspondiente del lado del cliente. Este código envía los datos del formulario pero en formato JSON. Como en los ejemplos anteriores, usamos el API de jQuery para la implementación, consultar su documentación para más información de cómo funciona el código.

```
        <form id="registro" action="#">
                Login: <input type="text" name="login" id="login"> <span
id="mensaje"></span><br>
                Password: <input type="password" name="password"
id="password"> <br>
                Nombre y apellidos: <input type="text" name="nombre"
id="nombre"> <br>
                <input type="submit" value="registrar">
        </form>
        <script type="text/javascript">
                $('#registro').submit(function(evento) {
                        $.ajax({
                                url: 'altaUsuario.do',
                                type: 'POST',
                                data: JSON.stringify({login:
$('#login').val(),
                                        password: $('#password').val(),
nombre: $('#nombre').val()}),
                                processData: false,
                                contentType: "application/json"
                        })
                        evento.preventDefault();
                });
        </script>
```

2. Servicios web REST

Desde Spring 3.0, el módulo MVC ofrece soporte para aplicaciones web RESTful, siendo precisamente esta una de las principales novedades de esta versión. Actualmente Spring ofrece funcionalidades muy similares a las del estándar JAX-RS, pero perfectamente integradas con el resto del *framework*. Nos limitaremos a explicar aquí el API de Spring para REST obviando los conceptos básicos de esta filosofía, que ya se vieron en el módulo de componentes web.

2.1. URIs

Como ya se vio en el módulo de servicios web, en aplicaciones REST cada recurso tiene una URI que lo identifica, organizada normalmente de modo jerárquico. En un sistema en el que tenemos ofertas de alojamiento de distintos hoteles, distintas ofertas podrían venir identificadas por URLs como:

```
/hoteles/excelsiorMad/ofertas/15
/hoteles/ambassador03/ofertas/15 (el id de la oferta es único solo dentro
del hotel)
/hoteles/ambassador03/ofertas/   (todas las del hotel)
```

Las ofertas aparecen determinadas primero por el nombre del hotel y luego por su identificador. Como puede verse, parte de la URL es la misma para cualquier oferta, mientras que la parte que identifica al hotel y a la propia oferta es variable. En Spring podemos expresar una URL de este tipo poniendo las partes variables entre llaves: `/hoteles/{idHotel}/ofertas/{idOferta}`. Podemos asociar automáticamente estas partes variables a parámetros del método java que procesará la petición HTTP. Por ejemplo:

```
@Controller
public class OfertaRESTController {
    @Autowired
    IOfertasBO obo;

    @RequestMapping(value="hoteles/{idHotel}/ofertas/{idOferta}",
                                method=RequestMethod.GET)
    @ResponseBody
    public Oferta mostrar(@PathVariable String idHotel, @PathVariable int
idOferta) {
        Oferta oferta =  obo.getOferta(idHotel, idOferta);
        return oferta;
    }
}
```

Las partes variables de la URL se asocian con los parámetros Java del mismo nombre. Para que esta asociación funcione automáticamente, hay que compilar el código con la información de debug habilitada. En caso contrario podemos asociarlo explícitamente: `@PathVariable("idHotel") String idHotel`. Spring puede convertir las `@PathVariable` a los tipos más típicos: String, numéricos o Date.

Nótese que Spring no asociará este *controller* a una URI como `hoteles/cr124/ofertas/`, ya que se espera una `PathVariable` para la oferta que aquí

no existe. Lo que tiene su lógica, ya que esta URL significa que queremos hacer alguna operación con *todas* las ofertas del hotel, y esto es mejor que sea tarea de otro método Java distinto a `mostrar`.

Nótese además que el método java `mostrar` viene asociado solo a las peticiones de tipo GET. En una aplicación REST, típicamente el método Java encargado de editar una oferta se asociaría a la misma URL pero con PUT, y lo mismo pasaría con insertar/POST, y borrar/DELETE

Por otro lado, como ya hemos visto en la sección de AJAX, la anotación `@ResponseBody` hace que lo que devuelve el método Java se serialice en el cuerpo de la respuesta HTTP que se envía al cliente. La serialización en JSON o XML se hace exactamente de la misma forma que vimos en AJAX.

Nótese que el ejemplo anterior se podría modificar de modo sencillo para una aplicación web convencional con JSP, como veníamos haciendo en las sesiones de MVC. Únicamente habría que poner un parámetro de tipo `Model`, añadirle el objeto `oferta` encontrado y devolver un `String` con el nombre lógico de la vista, donde se mostrarían los datos.

2.2. Obtener recursos (GET)

La implementación del apartado anterior era demasiado básica, ya que no estamos controlando explícitamente elementos importantes de la respuesta HTTP como el código de estado o las cabeceras. Para controlarlos, en Spring podemos hacer uso de la clase `ResponseEntity`, que modela la respuesta HTTP y con la que podemos devolver los objetos serializados, fijar el código de estado y añadir cabeceras. Vamos a ver cómo se implementaría una versión algo más sofisticada y con una sintaxis más "propia de REST"

```
@Controller
@RequestMapping("/hoteles/{idHotel}/ofertas")
public class OfertasController {
        @Autowired
        IOfertasBO obo;


        @RequestMapping(method=RequestMethod.GET,
                        value="{idOferta}",
                        produces=MediaType.APPLICATION_JSON_VALUE)
        public ResponseEntity<Oferta> obtenerOferta(@PathVariable String
idHotel,
                                                    @PathVariable int
idOferta)
                                                      throws
OfertaInexistenteException {
                Oferta oferta = obo.getOferta(idHotel, idOferta);
                return new ResponseEntity<Oferta>(oferta, HttpStatus.OK);
        }
}
```

Del código anterior destacar que:

- Para simplificar las URIs de cada método se hace que el controller en general responda a la parte "fija" y común a todas las URIs de las que se encargará, y en cada método se pone solo la parte que va a continuación. Nótese que los parámetros del método pueden referenciar cualquier `@PathVariable` de la URI, aunque aparezca en la anotación de la clase (como el id. del hotel).
- Usamos el atributo `produces` de `@RequestMapping` para fijar el valor de la cabecera HTTP "content-type", al igual que hacíamos en AJAX.
- Como ya hemos dicho, la clase `ResponseEntity` representa la respuesta HTTP. Cuando en el cuerpo de la respuesta queremos serializar un objeto usamos su clase para parametrizar `ResponseEntity`. Hay varios constructores de esta clase. El más simple admite únicamente un código de estado HTTP (aquí 200 OK). El que usamos en el ejemplo tiene además otro parámetro en el que pasamos el objeto a serializar.
- Nótese que en caso de solicitar una oferta inexistente se generaría una excepción, lo que en la mayoría de contenedores web nos lleva a una página HTML de error con la excepción, algo no muy apropiado para un cliente REST, normalmente no preparado para recibir HTML. Veremos luego cómo arreglar esto, convirtiendo automáticamente las excepciones en códigos de estado HTTP.
- Ya no hace falta la anotación `@ResponseBody` ya que al devolver un `ResponseEntity<Oferta>`, ya estamos indicando que queremos serializar un objeto en la respuesta HTTP.

2.3. Crear o modificar recursos (POST/PUT)

En este caso, el cliente envía los datos necesarios para crear el recurso y el servidor le debería responder, según la ortodoxia REST, con un código de estado 201 (Created) y en la cabecera "Location" la URI del recurso creado. Veamos cómo se implementaría esto en Spring:

```
@RequestMapping("/hoteles/{idHotel}/ofertas")
public class OfertasController {
        @Autowired
        IOfertasBO obo;

        //Ya no se muestra el código de obtenerOferta
        //...

        @RequestMapping(method=RequestMethod.POST, value="")
        public ResponseEntity<Void> insertarOferta(@PathVariable idHotel,
                                                   @RequestBody Oferta
oferta,
                                                   HttpServletRequest
peticion) {
                int idOFerta = obo.crearOferta(idHotel, oferta);
                HttpHeaders cabeceras = new HttpHeaders();
                try {
                        cabeceras.setLocation(new
URI(peticion.getRequestURL()+
Integer.toString(idOFerta)));
                } catch (URISyntaxException e) {
                        e.printStackTrace();
                }
```

```
                return new ResponseEntity<Void>(cabeceras,
HttpStatus.CREATED);
        }
}
```

A destacar del código anterior:

- Como no tenemos que serializar ningún objeto en el cuerpo de la respuesta, usamos un `ResponseEntity<Void>`. Le pasamos la cabecera "Location" generada y el código de estado 201, indicando que el recurso ha sido creado.
- Para generar la cabecera "Location" usamos el API de Spring (la clase `HttpHeaders`). La URI del nuevo recurso será la actual seguida del id de la nueva oferta.
- La anotación `@RequestBody` indica que en el cuerpo de la petición debemos enviar un objeto Oferta serializado. Esto guarda un paralelismo con el caso de uso de AJAX en que el cliente envía un objeto JSON al servidor. Nótese que el API es exactamente el mismo.

El caso de modificar recursos con PUT es similar en el sentido de que se envían datos serializados en la petición, pero es más sencillo en cuanto al valor de retorno, ya que nos podemos limitar a devolver un código de estado 200 OK si todo ha ido bien, sin necesidad de más cabeceras HTTP:

```
@RequestMapping(method=RequestMethod.PUT, value="{idOferta}")
public ResponseEntity<Void> insertarOferta(@PathVariable idHotel,
                                           @RequestBody Oferta oferta) {
        int idOFerta = obo.modificarOferta(idHotel, oferta);
        return new ResponseEntity<Void>(HttpStatus.CREATED);
}
```

2.4. Eliminar recursos (DELETE)

Este caso de uso suele ser sencillo, al menos en lo que respecta a la interfaz. Simplemente debemos llamar a la URI del recurso a eliminar, y devolver 200 OK si todo ha ido bien:

```
@RequestMapping(method=RequestMethod.DELETE, value="{idOferta}")
public ResponseEntity<Void> borrarOferta(@PathVariable
idHotel,@PathVariable idOferta) {
        obo.eliminarOferta(idHotel, idOferta);
        return new ResponseEntity<Void>(HttpStatus.OK);
}
```

2.5. Parte del cliente

El código del cliente REST podría escribirse usando directamente el API de JavaSE, o librerías auxiliares como Jakarta Commons HttpClient, que permite abrir conexiones HTTP de manera sencilla. No obstante, que sea sencillo no implica que no sea tedioso y necesite de bastantes líneas de código. Para facilitarnos la tarea, Spring 3 ofrece la clase `RestTemplate`, que permite realizar las peticiones REST en una sola línea de código Java.

Método HTTP	Método de RestTemplate
DELETE	delete(String url, String? urlVariables)
GET	getForObject(String url, Class<T> responseType, String? urlVariables)
HEAD	headForHeaders(String url, String? urlVariables)
OPTIONS	optionsForAllow(String url, String? urlVariables)
POST	postForLocation(String url, Object request, String? urlVariables)
	postForObject(String url, Object request, Class<T> responseType, String? uriVariables)
PUT	put(String url, Object request, String?urlVariables)

Table 1: Métodos de RestTemplate y su correspondencia con los de HTTP

Por ejemplo, una petición GET a la URL `hoteles/ambassador/ofertas/1` se haría:

```
RestTemplate template = new RestTemplate();
String uri =
"http://localhost:8080/ServidorREST/hoteles/{idHotel}/ofertas/{idOferta}";
Oferta oferta = template.getForObject(uri, Oferta.class, "ambassador", 1);
System.out.println(oferta.getPrecio() + "," + oferta.getFin());
```

Como se ve, en el cliente se usa la misma notación que en el servidor para las URLs con partes variables. En los parámetros del método Java `getForObject` se coloca el tipo esperado (la clase `Oferta`) y, por orden, las partes variables de la URL. Igual que en el servidor, si tenemos las librerías de Jackson en el *classpath* se procesará automáticamente el JSON. Así, Jackson se encarga de transformar el JSON de nuevo en un objeto Java que podemos manipular de modo convencional.

Si hubiéramos implementado en el servidor un método en el controller para dar de alta ofertas, podríamos llamarlo desde el cliente así:

```
RestTemplate template = new RestTemplate();
String uri =
"http://localhost:8080/TestRESTSpring3/hoteles/{idHotel}/ofertas";
//aquí le daríamos el valor deseado a los campos del objeto
Oferta oferta = new Oferta( ..., ..., ...);
URI result = template.postForLocation(uri, oferta, "ambassador");
```

El método de `RestTemplate` llamado `postForLocation` crea un nuevo recurso, y obtiene su URI, que necesitaremos para seguir interactuando con el objeto. La clase URI es del API estándar de JavaSE. En una aplicación REST el servidor debería devolver dicha URI como valor de la cabecera HTTP `Location`.

3. Tratamiento de errores en aplicaciones AJAX y REST

Como ya se ha visto en el módulo de componentes web, podemos especificar qué página HTML deseamos que se muestre cuando se genera una excepción en la capa web que acaba capturando el contenedor. Tanto la página de error que muestra el contenedor como la información que aparece en ella son totalmente configurables, dando una solución aceptable para aplicaciones web "clásicas". No obstante este mecanismo de gestión de errores no es apropiado para aplicaciones AJAX o REST, ya que el HTML con la información de error no es un formato apropiado ni para el javascript en el primer caso ni para el cliente REST en el segundo. Lo más adecuado en estos casos es devolver un código de estado HTTP que indique de la manera más precisa posible qué ha pasado y en el cuerpo de la respuesta un mensaje en texto plano con más datos sobre el error. La forma más sencilla de implementar esto en Spring es mediante el uso de *exception handlers*.

Un *exception handler* no es más que un método cualquiera del *controller* anotado con `@ExceptionHandler`. Este método capturará el o los tipos de excepciones que deseemos, y lo que haremos en el método será devolver la excepción en una forma amigable a los clientes AJAX/REST. Por ejemplo, para gestionar el caso de las ofertas inexistentes podríamos hacer algo como:

```
@ExceptionHandler(OfertaInexistenteException.class)
public ResponseEntity<String>
gestionarNoExistentes(OfertaInexistenteException oie) {
        return new ResponseEntity<String>(oie.getMessage(),
HttpStatus.NOT_FOUND);
}
```

En el ejemplo simplemente enviamos un código 404 y en el cuerpo de la respuesta colocamos el mensaje de la excepción, aunque podríamos hacer cualquier otra cosa que deseáramos (colocar cabeceras, serializar un objeto en la respuesta, o incluso devolver un String que se interpretaría, como es habitual, como el nombre de una vista que mostrar).

La anotación `@ExceptionHandler` admite varias excepciones, de modo que podemos usar un único método gestor para varias distintas, por ejemplo:

```
@ExceptionHandler({OfertaInexistenteException.class,
HotelInexistenteException.class})
public ResponseEntity<String> gestionarNoExistentes(Exception e) {
        return new ResponseEntity<String>(e.getMessage(),
HttpStatus.NOT_FOUND);
}
```

Spring MVC genera unas cuantas excepciones propias, caso por ejemplo de que se produzca un error de validación en un objeto que hemos querido chequear con JSR303, ya veremos cómo (`BindException`),o que el cliente no acepte ninguno de los formatos que podemos enviarle (`HttpMediaTypeNotAcceptableException`), o que intentemos llamar con POST a un método que solo acepta GET (`HttpRequestMethodNotSupportedException`), entre otros. En esos casos actúa un *exception handler* definido por defecto que lo único que hace es capturar las excepciones

y generar códigos de estado HTTP (400 en el primer caso, 406 en el segundo y 405 en el tercero). Si queremos que se haga algo más, como enviar un mensaje con más información en el cuerpo de la respuesta, tendremos que definir nuestros propios *handlers* para esas excepciones.