PlugIn Tapestry

Desarrollo de aplicaciones y páginas web
con Apache Tapestry

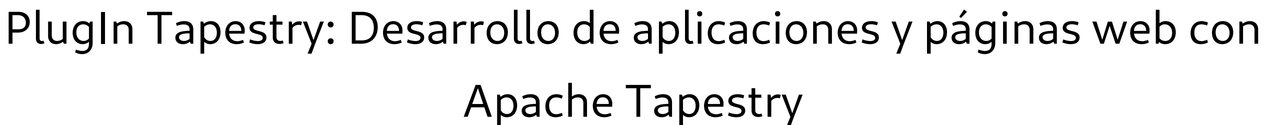
Autor
@picodotdev
https://picodotdev.github.io/blog-bitix/

2019 1.4.2 5.4

A tod@s l@s programador@s que en su trabajo
no pueden usar el framework, librería o lenguaje que quisieran.
Y a las que se divierten programando y aprendiendo
hasta altas horas de la madrugada.

Non gogoa, han zangoa

Hecho con un esfuerzo en tiempo considerable
con una buena cantidad de software libre
y más ilusión en una región llamada Euskadi.



PlugIn Tapestry: Desarrollo de aplicaciones y páginas web con Apache Tapestry

@picodotdev

2014 - 2019

Prefacio

Empecé El blog de pico.dev y unos años más tarde Blog Bitix con el objetivo de poder aprender y compartir el conocimiento de muchas cosas que me interesaban desde la programación y el software libre hasta análisis de los productos tecnológicos que caen en mis manos. Las del ámbito de la programación creo que usándolas pueden resolver en muchos casos los problemas típicos de las aplicaciones web y que encuentro en el día a día en mi trabajo como desarrollador. Sin embargo, por distintas circunstancias ya sean propias del cliente, la empresa o las personas es habitual que solo me sirvan meramente como satisfacción de adquirir conocimientos. Hasta el día de hoy una de ellas es el tema del que trata este libro, Apache Tapestry.

Para escribir en el blog solo dependo de mí y de ninguna otra circunstancia salvo mi tiempo personal, es completamente mío con lo que puedo hacer lo que quiera con él y no tengo ninguna limitación para escribir y usar cualquier herramienta, aunque en un principio solo sea para hacer un ejemplo muy sencillo, en el momento que llegue la oportunidad quizá me sirva para aplicarlo a un proyecto real.

Pasados ya unos pocos años desde que empecé el blog allá por el 2010 he escrito varios artículos tratando en cada una de ellos diferentes temas relacionados con Apache Tapestry y que toda aplicación web debe realizar independientemente del lenguaje o framework que se use. Con el blog me divierto mucho pero no se si es la forma más efectiva para difundir todas las bondades que ya conozco de este framework y que a medida voy conociéndolo más sigo descubriendo. Ya llevaba pensándolo bastante tiempo y ha llegado un punto en que juntando todos los artículos que he escrito en el blog completándolas con alguna cosa más podría formar un libro y el resultado es lo que tienes en la pantalla del dispositivo que uses para leerlo.

¿Es realmente necesario que escribiese este libro? Pues sí y no. No, porque ya hay otros muy buenos libros sobre Tapestry algunos escritos por los commiters del framework, como Tapestry 5 - Rapid web application development in Java, quizá mejor y de forma más completa que lo explicado en este y que alguien con interés podría adquirir sin ningún problema. Y sí, porque escribiendo uno en español hay más posibilidades de hacérselo llegar a mi entorno más o menos cercano.

Mi objetivo con este libro es difundir la palabra para que otra gente disfrute con este framework tanto como lo hago yo cuando programo con él y finalmente aumentar aunque sea un poco las posibilidades de que pueda dedicar mi jornada laboral completa usándolo (guiño, guiño). Tapestry no tiene el «hype» de otros frameworks, ni lleva la etiqueta ágil (aunque podría) que parece que ahora si no se le pone a algo no «mola» y no merece consideración pero tiene muchas características desde casi sus inicios en que fue publicado en el 2002 con la versión 2 que ya desearían para sí muchos otros aún en la actualidad.

Como habrás notado este libro no te ha costado ni un céntimo, ¿por qué lo distribuyo al precio de 0,00€ impuestos incluidos? La razón es simple, porque quiero. Si cobrase algo por él probablemente la audiencia que

tuviese no sería muy amplia y de todos modos no saldría de pobre, siendo gratis espero que unos cuantos desarrolladores al menos lo vean por encima simplemente por cultura general y lo comparen con lo que usen para programar ahora, ya sea Struts, Grails, Play!, Django, Symfony, Silex, Ruby on Rails, .NET MVC u otros similares. Si de entre esos que lo leen hay unos cuantos que se animan a probarlo ya me sentiría satisfecho, si además alguno lo usase para un proyecto real con éxito me haría muy feliz.

Gran parte de este libro está basado en lo que he aprendido desde el 2004 mediante su documentación oficial y usándolo principalmente de forma autodidacta. No constituye una guía completa y exhaustiva, ni lo pretende, simplemente es un manual suficientemente amplio para transmitir al lector los conceptos más importantes y que una vez aprendidos sea capaz de aprender el resto profundizando por sí mismo si consigo despertar su curiosidad. La documentación oficial del proyecto es amplia, buena, completa y suficiente para aprender desde cero (algunas buenas partes de este libro son poco más que una traducción) pero además de la documentación puramente técnica quiero aportar la experiencia y algunas buenas prácticas que he obtenido como usuario durante estos años y desde mi comienzo como programador no solo de este framework.

Lo que viene a continuación

En los siguientes capítulos encontrarás una explicación detallada de las características del framework y la forma de resolver una gran parte de los aspectos con los que tienen que tratar las aplicaciones o páginas web: el entorno de desarrollo, generar el html con plantillas, la lógica de presentación, la internacionalización y localización, la persistencia de la capa de presentación y persistencia en la base de datos, el contenedor de inversión de control, la seguridad, peticiones ajax y datos en json, enviar formularios, recibir archivos y devolverlos, como crear layouts para dar un aspecto común a las páginas sin duplicar código, reutilización de código con componentes y con librerías de componentes, pruebas unitarias, de integración y funcionales, assets (estilos, imágenes, javascript) y algunas cosas más adicionales en las que no entraré en muchos detalles pero que daré las indicaciones de como realizarlas como el envió de correos, generación de informes, gráficas, una API REST y analizadores estáticos de código que pueden ser necesarios en algunos casos.

Teniendo experiencia y habiendo trabajado en proyectos reales con JSP/Servlets, Struts, JSF, Grails y Apache Tapestry me quedo con una diferencia significativa con la última opción como puedes suponer si he dedicado una gran cantidad de tiempo personal a escribir este libro y el que dedico en mi blog. Trataré de exponer en las siguientes páginas muchos de los motivos que Tapestry me da para ello y que quizá tú también consideres.

¡Empieza la diversión! ¿estás preparad@?

Indice

1 Introducción **13**

1.1 Principios 13

1.2 Características 18

1.3 Un poco de historia 23

1.4 Opciones alternativas 24

1.5 Arquitectura de aplicaciones web 26

1.6 Casos de éxito y de referencia 32

2 Inicio rápido **33**

2.1 Instalación JDK 33

2.2 Inicio rápido 33

2.3 Entorno de desarrollo 37

2.4 Integración con el servidor de aplicaciones 38

2.4.1 Spring Boot 38

2.4.2 Spring Boot generando un jar 47

2.4.3 Spring Boot generando un war 47

2.4.4 Servidor de aplicaciones externo 49

2.5 Debugging 50

2.6 Código fuente de los ejemplos 52

3 Páginas y componentes **53**

3.1 Clase del componente 54

3.2 Plantillas 60

3.2.1 Content Type y markup 70

3.3 Parámetros del los componentes 70

3.3.1 Bindings de parámetros 72

3.4 La anotación @Parameter 75

3.4.1 Parámetros requeridos 75

3.4.2 Parámetros opcionales 76

3.4.3 Parámetros informales 77

3.4.4 Conversiones de tipo en parámetros 79

3.5 La anotación @Cached 80

3.6 Conversiones de tipos 82

3.7 Renderizado de los componentes 83

3.7.1 Fases de renderizado 83

3.7.2 Conflictos y ordenes de métodos 89

3.8 Navegación entre páginas 90

3.9 Peticiones de eventos de componente y respuestas 91

3.10 Peticiones de renderizado de página 94

3.11 Patrones de navegación de páginas 95

3.12 Eventos de componente 99

3.12.1 Métodos manejadores de evento 99

3.13 Componentes disponibles 104

3.14 Página Dashboard 106

3.15 Productividad y errores de compilación 109

4 Contenedor de dependencias (IoC) 115

4.1 Objetivos . . . 116

4.2 Terminología . . . 117

4.3 Inversión de control (IoC) . . . 118

4.4 Clase contra servicio . . . 122

4.5 Inyección . . . 122

4.5.1 Configuración en Tapestry IoC . . . 127

4.6 Tutores de servicios . . . 132

4.7 Conversiones de tipos . . . 135

4.8 Símbolos de configuración . . . 139

5 Assets y módulos RequireJS 141

5.1 Assets en las plantillas . . . 141

5.2 Assets en las clases de componente . . . 141

5.3 Minimizando assets . . . 144

5.4 Hojas de estilo . . . 144

5.5 JavaScript . . . 146

5.5.1 Añadiendo JavaScript personalizado . . . 146

5.5.2 Combinando librerías de JavaScript . . . 148

5.5.3 Minificando librerías de JavaScript . . . 148

5.5.4 Pilas de recursos . . . 149

5.5.5 RequireJS y módulos de Javascript . . . 151

5.6 Assets con Webjars . . . 154

5.7 Actualizar versiones assets incorporados por defecto . . . 156

6 **Formularios** 161

6.1 Eventos del componente Form . . . 161

6.2 Seguimiento de errores de validación . . . 162

6.3 Almacenando datos entre peticiones . . . 162

6.4 Configurando campos y etiquetas . . . 164

6.5 Errores y decoraciones . . . 165

6.6 Validación de formularios . . . 168

6.6.1 Validadores disponibles . . . 168

6.6.2 Centralizando la validación . . . 169

6.6.3 Personalizando los errores de validación . . . 170

6.6.4 Configurar las restricciones de validación en el catálogo de mensajes . . . 171

6.6.5 Macros de validación . . . 171

6.7 Subiendo archivos . . . 172

6.8 Conversiones . . . 174

7 **Internacionalización (i18n) y localización (l10n)** 177

7.1 Catálogos de mensajes . . . 177

7.1.1 Catálogo global de la aplicación . . . 178

7.1.2 Accediendo a los mensajes localizados . . . 178

7.2 Imágenes . . . 180

7.3 Selección del locale . . . 181

7.4 JavaScript . . . 182

7.5 Convenciones para archivos properties l10n . . . 183

8 **Persistencia en la capa de presentación** 185

8.1 Persistencia de página . . . 185

8.1.1 Estrategias de persistencia . . . 185

8.2 Valores por defecto . . . 187

8.3 Persistencia de sesión . . . 188

8.3.1 Datos de sesión externalizados con Spring Session . . . 190

9 Persistencia en base de datos **195**

9.1 Bases de datos relacionales 195

9.1.1 Propiedades ACID 196

9.1.2 Lenguaje SQL 197

9.2 Bases de datos NoSQL 197

9.3 Persistencia en base de datos relacional 198

9.4 Transacciones 210

9.4.1 Anotación CommitAfter 210

9.4.2 Transacciones con Spring 211

10 AJAX **213**

10.1 Zonas 213

10.1.1 Retorno de los manejadores de evento 213

10.1.2 Actualización del múltiples zonas 214

10.2 Peticiones Ajax que devuelven JSON 216

10.3 Lanzar eventos desde JavaScript 218

11 Seguridad **223**

11.1 Autenticación y autorización 223

11.2 XSS e inyección de SQL 228

11.3 Cross-site request forgery (CSRF) 229

11.4 ¿Que hay que hacer para evitar estos problemas? 229

11.5 Usar el protocolo seguro HTTPS 237

11.6 Salted Password Hashing 243

12 Librerías de componentes **249**

12.1 Crear una librería de componentes 249

12.2 Informe de componentes 253

13 Pruebas unitarias y de integración 255

13.1 Pruebas unitarias 256

13.1.1 Pruebas unitarias incluyendo código HTML 258

13.1.2 Pruebas unitarias incluyendo código HTML con XPath 262

13.2 Pruebas de integración y funcionales 263

13.3 Ejecución con Gradle y Spring Boot 266

14 Otras funcionalidades habituales 269

14.1 Funcionalidades habituales 269

14.1.1 Integración con Spring 269

14.1.2 Plantillas 275

14.1.3 Documentación Javadoc 281

14.1.4 Páginas de códigos de error y configuración del servidor con Spring Boot 283

14.1.5 Página de informe de error 288

14.1.6 Logging 293

14.1.7 Internacionalización (i18n) en entidades de dominio 293

14.1.8 Relaciones jerárquicas en bases de datos relacionales 294

14.1.9 Multiproyecto con Gradle 297

14.1.10 Máquina de estados finita (FSM) 297

14.1.11 Configuración de una aplicación en diferentes entornos con Spring Cloud Config 297

14.1.12 Información y métricas con Spring Boot Actuator 298

14.1.13 Aplicaciones que tratan con importes 299

14.1.14 Cómo trabajar con importes, ratios y divisas en Java 299

14.1.15 Validar objetos con Spring Validation 304

14.1.16 Java para tareas de «scripting» 305

14.1.17 DAO genérico 305

14.1.18 Mantenimiento de tablas con un CRUD 306
14.1.19 Doble envío (o N-envío) de formularios 306
14.1.20 Patrón múltiples vistas de un mismo dato 309
14.1.21 Servir recursos estáticos desde un CDN propio u otro como CloudFront 314
14.1.22 Ejecución en el servidor de aplicaciones JBoss o WildFly 317
14.1.23 Aplicación «standalone» 321
14.2 Funcionalidades de otras librerías 323
14.2.1 Ansible y Docker 323
14.2.2 Librerías JavaScript 323
14.2.3 Interfaz REST 324
14.2.4 Interfaz RPC 324
14.2.5 Portlets 325
14.2.6 Cache 325
14.2.7 Plantillas 325
14.2.8 Informes 326
14.2.9 Gráficas 326
14.2.10 Análisis estático de código 326
14.2.11 Facebook y Twitter 326
14.2.12 Fechas 326

15 Notas finales **327**

15.1 Comentarios y feedback 327
15.2 Más documentación 327
15.3 Ayuda 329
15.4 Sobre el autor 329
15.5 Lista de cambios 330
15.6 Sobre el libro 333
15.7 Licencia 334

Capítulo 1

Introducción

En este capítulo te describiré Tapestry de forma teórica, sus principios y características que en parte te permitirán conocer por que puedes considerar útil la inversión de tiempo que vas a dedicar a este libro y a este framework. Espero que despierten tu curiosidad y continúes leyendo el resto del libro ya viendo como como se hacen de forma más práctica muchas de las cosas que una aplicación o página web tiene que abordar y que un framework debe facilitar.

Podemos empezar.

1.1 Principios

Tapestry es un framework orientado a componentes para el desarrollo de aplicaciones web programadas en el lenguaje Java dinámicas, robustas y altamente escalables. Se sitúa en el mismo campo que Wicket y JSF en vez de junto con Spring MVC, Grails y Play!, estos últimos orientados a acciones como Struts.

Posee muchas características, algunas muy buenas desde sus inicios como el ser lo máximo informativo que le es posible cuando se producen excepciones y que aún frameworks más recientes y nuevos aún no poseen. Muchas de estas características han cambiado y sido añadidas con cada nueva versión mayor y otras siguen siendo muy similares a como eran anteriormente. Aparte de las características principales hay muchos otros detalles más pequeños que hacen agradable y simple usar este framework.

Los principios que guían el desarrollo de Tapestry son los siguientes.

Simplicidad

Esto implica que las plantillas y código sea legible y conciso. También implica que no se extiende de las clases de Tapestry. Al ser las clases POJO (Plain Old Java Objects) se facilitan las pruebas unitarias y se evitan problemas al realizar actualizaciones a versiones superiores.

Consistencia

Significa que lo que funciona a un nivel, como en una página completa, funciona también para los componentes dentro de una página. De esta manera los componentes anidados tienen la misma libertad de persistir datos, anidar otros componentes y responder a eventos como en la página de nivel superior. De hecho los componentes y páginas se distinguen más bien en poco salvo por que los componentes están contenidos en otras páginas o componentes y las páginas están en el nivel superior de la jerarquía.

Feedback

Una de la más importante. En los primeros días de los servlets y JSP la realidad era que en el mejor de los casos obteníamos una excepción en el navegador o el log del servidor en el momento de producirse una. Únicamente a partir de esa traza de la excepción uno tenía que averiguar que había sucedido. Tapestry proporciona más información que una excepción, genera un informe en el que se incluye toda la información necesaria para determinar la causa real. Disponer de únicamente la traza de un NullPointerException (por citar alguna) se considera un fallo del framework y simplemente eso no ocurre en Tapestry (al menos en el momento de desarrollo). El informe incluye líneas precisas de código que pueden decir que tienes un error en la línea 50 de una plantilla y mostrar un extracto del código directamente en el informe de error además de los parámetros, algunos atributos de la request, las cabeceras y cookies que envío del navegador además de las variables de entorno, el classpath y atributos y valores de la sesión, por supuesto también incluye la traza de la excepción.

Eficiencia

A medida que ha evolucionado ha ido mejorando tanto en velocidad operando de forma más concurrente y en consumo de memoria. Se ha priorizado primero en escalar verticalmente que horizontalmente mediante clusters, que aún con afinidad de sesiones complican la infraestructura considerablemente y supone retos a solucionar. Tapestry usa la sesión de forma diferente a otros frameworks tendiendo a usar valores inmutables y simples como números y cadenas que están más acorde con el diseño de la especificación de la API de los servlets.

Estructura estática, comportamiento dinámico

El concepto de comportamiento dinámico debería ser obvio al construir una aplicación web, las cosas deben ser diferentes ante diferentes usuarios y situaciones. Pero, ¿qué significa que Tapestry tiene estructura estática?

La estructura estática implica que cuando se construye una página en Tapestry se define todos los tipos de los componentes que son usando en la página. En ninguna circunstancia durante la fase de generación o el procesado de un evento la página podrá crear dinámicamente un nuevo tipo de componente y colocarlo en el árbol de componentes de la página.

En un primer momento, esto parece bastante limitante... otros frameworks permiten crear nuevos elementos dinámicamente, es también una característica común de las interfaces de usuario como Swing. Pero una estructura estática resulta no tan limitante después de todo. Puedes crear nuevos elementos «re-rendering» componentes existentes usando diferentes valores para las propiedades y tienes multitud de opciones para obtener comportamiento dinámico de la estructura estática, desde un simple condicional y componentes de bucle a implementaciones más avanzadas como los componentes BeanEdit o Grid. Tapestry proporciona el control sobre que se genera y cuando, e incluso cuando aparece en la página. Y desde Tapestry 5.3 se puede usar incluso el componente Dynamic que genera lo que esté en un archivo de plantilla externo.

¿Por qué Tapestry eligió una estructura estática como un principio básico? En realidad es debido a los requerimientos de agilidad y escalabilidad.

Agilidad

Tapestry está diseñado para ser un entorno de desarrollo ágil , «code less, deliver more». Para permitir escribir menos código Tapestry realiza mucho trabajo en las clases POJO de páginas y componentes cuando se cargan por primera vez. También usa instancias de clases compartidas de páginas y componentes (compartidas entre múltiples hilos y peticiones). Tener una estructura modificable implica que cada petición tiene su instancia, y es más, que la estructura entera necesitaría ser serializada entre peticiones para que puedan ser restablecidas para procesar peticiones posteriores.

Con Tapestry se es más ágil al acelerar el ciclo de desarrollo con la recarga de clases dinámica. Tapestry monitoriza el sistema de archivos en búsqueda de cambios a las clases de las páginas, clases de componentes, implementaciones de servicios, plantillas HTML y archivos de propiedades y aplica los cambios con la aplicación en ejecución sin requerir un reinicio o perder los datos de sesión. Esto proporciona un ciclo muy corto de codificación, guardado y visualización que no todos los frameworks pueden igualar.

Escalabilidad

Al construir sistemas grandes que escalen es importante considerar como se usan los recursos en cada servidor y como esa información va a ser compartida entre los servidores. Una estructura estática significa que las instancias de las páginas no necesitan ser almacenadas dentro de HttpSession y no se requieren recursos extras en navegaciones simples. Este uso ligero de HttpSession es clave para que Tapestry sea altamente escalable, especialmente en configuraciones cluster. De nuevo, unir una instancia de una página a un cliente particular requiere significativamente más recursos de servidor que tener solo una instancia de página compartida.

Adaptabilidad

En los frameworks Java tradicionales (incluyendo Struts, JSF e incluso el antiguo Tapestry 4) el código del usuario se esperaba que se ajustase al framework. Se creaban clases que extendiesen de las clases base o implementaba interfaces proporcionadas por el framework. Esto funciona bien hasta que actualizas a la siguiente versión del framework, con nuevas características, y más a menudo que no se rompe la compatibilidad hacia atrás. Las interfaces o clases base habrán cambiado y el código existente necesitará ser cambiado para ajustarse.

En Tapestry 5, el framework se adapta al código. Se tiene el control sobre los nombres de los métodos, los parámetros y el valor que es retornado. Esto es posible mediante anotaciones que informan a Tapestry que métodos invocar. Por ejemplo, podemos tener una página de inicio de sesión y un método que se invoca cuando el formulario es enviado:

Listado 1.1: Login.java

```
public class Login {

  @Persist
  @Property
  private String userId;

  @Property
  private String password;

  @Component
  private Form form;

  @Inject
  private LoginAuthenticator authenticator;

  void onValidateFromForm() {
    if (!authenticator.isValidLogin(userId, password)) {
      form.recordError("Invalid user name or password.");
    }
  }

  Object onSuccessFromForm() {
    return PostLogin.class;
  }
}
```

Este pequeño extracto muestra un poco acerca de como funciona Tapestry. Las páginas y servicios en la aplicación son inyectados con la anotación @Inject. Las convenciones de los nombres de métodos, onValidateFromForm() y onSuccessFromForm(), informan a Tapestry de que métodos invocar. Los eventos, «validate» y «success», y el id del componente determinan el nombre del método según la convención.

El método «validate» es lanzado para realizar validaciones sobre los datos enviados y el evento «success» solo es lanzado cuando no hay errores de validación. El método onSuccessFromForm() retorna valores que indican

a Tapestry que hacer después: saltar a otra página de la aplicación (en el código identificado por la clase de la página pero existen otras opciones). Cuando hay excepciones la página es mostrada de nuevo al usuario.

Tapestry te ahorra esfuerzo, la anotación @Property marca un campo como leible y escribible, Tapestry proporcionará métodos de acceso (get y set) automáticamente.

Finalmente, Tapestry separa explícitamente acciones (peticiones que cambian cosas) y visualizaciones (peticiones que generan páginas) en dos tipos de peticiones separadas. Realizar una acción como hacer clic en un enlace o enviar un formulario después de su procesado resulta en una redirección a una nueva página. Este es el patrón Enviar/Redirección/Obtener (Post/Redirect/Get, Post-then-Redirect o Redirect AfterPost). Este hace que todas las URLs son añadibles a los marcadores... pero también requiere que un poco de información sea almacenada en la sesión entre peticiones (usando la anotación @Persist).

Diferenciar público de API interna

Un problema de versiones anteriores de Tapestry (4 y anteriores) fue la ausencia clara de una distinción entre APIs internas privadas y APIs externas públicas. Diseñado de una nueva base, Tapestry es mucho más inflexible acerca de que es interno y externo. Primero de todo, cualquier cosa dentro del paquete org.apache.tapestry5.internal es interno. Es parte de la implementación de Tapestry. No se debería usar directamente este código. Es una idea mala hacerlo porque el código interno puede cambiar de una versión a la siguiente sin importar la compatibilidad hacia atrás.

Asegurar la compatibilidad hacia atrás

Las versiones antiguas de Tapestry estaban plagadas de problemas de incompatibilidades con cada nueva versión mayor. Tapestry 5 ni siquiera intentó ser compatible con Tapestry 4. En vez de eso, puso las bases para una verdadera compatibilidad hacia atrás en un futuro. Las APIs de Tapestry 5 se basan en convenciones y anotaciones. Los componentes son clases Javas ordinarias, se anotan propiedades para permitir a Tapestry mantener su estado o permitiendo a Tapestry inyectar recursos y se dan nombre a los métodos (o anotan) para informar a Tapestry bajo que circunstancias un método debe ser invocado.

Tapestry se adaptará a las clases, llamará a los métodos pasando valores mediante los parámetros. En vez de la rigidez de una interfaz fija a implementar, Tapestry simplemente se adaptará a las clases usando las indicaciones proporcionadas por anotaciones y simples convenciones de nombres.

Por esta razón, Tapestry 5 puede cambiar internamente en una grado alto sin afectar a ninguna parte del código de la aplicación facilitando actualizar a futuras versiones sin romper las aplicaciones. Esto ya ha sido evidente en Tapestry 5.1, 5.2, 5.3 y 5.4 donde se han añadido características importantes y mejoras manteniendo una compatibilidad hacia atrás en un muy alto grado, siempre que que se haya evitado la tentación de usar APIs internas. Lógicamente las aplicaciones siguen necesitando cambios para aprovechar las nuevas funcionalidades que se añaden en cada nueva versión.

1.2 Características

A pesar de todo estos solo son principios y están en constante evolución, lo principal al añadir nuevas funcionalidades es cuán útiles son. En ocasiones añadir una funcionalidad implica mejorar un principio y bajar en otros. A continuación veamos unas cuantas de sus características más importantes.

Java

El lenguaje de programación empleado habitualmente para el código asociado a las páginas y componentes de programación es Java. Es un lenguaje orientado a objetos compilado a bytecode que a su vez es interpretado y ejecutado por la máquina virtual de Java (JVM, Java Virtual Machine). El bytecode es el mismo e independiente de la arquitectura de la máquina donde realmente se ejecuta por la JVM. Esto permite que una vez escrito el código y compilado pueda ser ejecutado en cualquier máquina que tenga una JVM instalada. Esto nos permite desarrollar y producir el archivo war de una aplicación web en Windows o Mac para finalmente ser ejecutado en un servidor con GNU/Linux.

A pesar de que hay mucha competencia en el ámbito de los lenguajes de programación, Java está disponible desde 1995, C# desde el 2000 (similar en conceptos) y de otros tantos como Python (1991) y Ruby (1995) y otros de la misma plataforma como Groovy (2003) y Scala (2003), a día de hoy sigue siendo uno de los lenguajes más usados y demandados en puestos de trabajo. A diferencia de muchos de los anteriores Java es un lenguaje compilado (a bytecode) y fuertemente tipado. No es interpretado (salvo el bytecode de la JVM) con lo que obtendremos mucha ayuda del compilador que nos informará de forma muy precisa cuando algo en el código no pueda ser entendido de forma inmediata antes de ejecutar siquiera el código. En los lenguajes interpretados es habitual que se produzcan errores en tiempo de ejecución que un compilador hubiese informado, estos pueden ser introducidos por un error de escritura del programador o por una mala fusión de un conflicto en la herramienta de control de versiones con lo que hay que tener especial cuidado al usar lenguajes interpretados. El compilador es una gran ayuda y una de sus razones de existencia además de producir bytecode es evitar que lleguen estos errores al momento de ejecución, para nada hay que menospreciar al compilador y sobrevalorar la interpretación, desde luego tampoco hay que confundir dinámico con ágil.

Personalmente habiendo trabajado con Groovy lo único que echo de menos de él es el concepto de Closure y los DSL en menor medida pero por muy útil que me parezcan ambas cosas si es a costa de no tener un compilador que avise de los errores y la ayuda de los asistentes de los entornos de desarrollo integrados (IDE, Integrated Development Environment) como en los refactors no lo cambio a día de hoy, en un futuro es muy posible que mejoren las herramientas y estos problemas se solucionen en parte porque completamente será difícil por la naturaleza no completamente fuertemente tipada de Groovy. Las lambdas han sido añadidas en la versión 8 de Java.

Políglota

Dicho lo dicho en el apartado Java si prefieres programar los componentes y páginas en cualquier otro lenguaje soportado por la JVM es perfectamente posible. Tapestry acepta cualquiera de ellos (Groovy, Scala, ...) en teoría.

No fullstack

El no ser un framework fullstack tiene ciertas ventajas y desventajas. Entre las desventajas es que al no darte un paquete tan completo y preparado deberás pasar un poco más de tiempo en seleccionar las herramientas que necesites para desarrollar la aplicación, como podría ser la herramienta de construcción del proyecto, la librería para hacer pruebas unitarias, de integración o para persistir los datos en la base de datos. Las ventajas son que tú eres el que elige las herramientas con las que trabajar y la forma de hacerlo es como tú decidas. Tapestry proporciona lo necesario para la capa de presentación (con el añadido del contenedor de dependencias) y tiene algunas librerías de integración con otras herramientas para persistencia con Hibernate, seguridad con Shiro, etc... Tú eres el que decide, no debes aceptar lo que alguien creyó más conveniente para todas las aplicaciones que tal vez en tu caso ni siquiera necesites ni uses. Como tú decides puedes usar las piezas que más convenientes creas, si dentro de un tiempo sale la «megaherramienta» y quieres usarla no estarás limitado por el framework e incluso si quieres reemplazar Tapestry y has diseñado la aplicación por capas, salvo el código de presentación que quieres sustituir por algo equivalente gran parte del código te seguirá siendo perfectamente válido como sería toda la lógica de negocio.

Live class reloading

Hace no tanto tiempo había que estar reiniciando el servidor constantemente para ver los cambios que se iban haciendo al código. Mediante la recarga en caliente para muchos casos ya no será necesario reiniciar, Tapestry aplicará inmediatamente cualquier cambio que se produzca en el código de las páginas y componentes, de los servicios que gestiona su contenedor de dependencias y de los recursos de imágenes, css, javascript y catálogos de mensajes con lo que simplemente con hacer el cambio y actualizar el navegador los veremos aplicados. En algunos casos sigue siendo necesario reiniciar pero ahora no es necesario tan a menudo y cuando sea necesario Tapestry arranca muy rápidamente si no se hace nada extraño al inicio de la aplicación, en unos 10 segundos la aplicación puede estar cargada y en 15 sirviendo páginas.

Basado en componentes

Esta es la esencia de las aplicaciones de este framework, todo son componentes incluso las páginas lo son. Esto implica que aprendiendo como funciona este único concepto ya tendremos mucho aprendido.

Un componente es completamente autónomo, esto es, incluye en la página todo lo necesario para funcionar, como usuarios de uno no necesitaremos conocer más de él que los parámetros que necesita para usarlo. Es una caja negra que no deberemos abrir ni necesitaremos saber que hace por dentro para usarlo. Las imágenes que vayan a mostrar, los textos localizados, las hojas de estilo y los archivos javascripts serán incluidos en la página únicamente en el caso de que se use, de manera que no deberemos incluir previamente, ni globalmente y en todas las páginas todos los posibles recursos que se necesiten aunque en determinadas sepamos que algunos no son realmente necesarios. Esto hace que las páginas sean más eficientes y carguen más rápido.

Los componentes son la forma de reutilizar código. ¿Tienes una funcionalidad común a muchas páginas o en una misma varias veces? Con crear un componente podrás reutilizar ese código. También a través de ellos se consigue un alta productividad y un completo DRY (Don’t Repeat Yourself). Por si fuera poco los componentes

pueden almacenarse en librerías y para tenerlos disponibles en una aplicación sólo será necesario incluir una dependencia en el proyecto o un archivo jar. Como son autónomos en el jar están todos los recursos necesarios, solo deberemos preocuparnos por sus parámetros y como usarlos. Tapestry se encargará de extraer del jar los recursos y servirlos. Si los componentes permiten reutilizar código en una misma aplicación, las librerías de componentes permiten reutilizar código en distintas aplicaciones o por parte de terceros.

Algunas diferencias con Servlets/JSP y Grails

La tecnología de presentación de páginas web Java con Java Server Pages o JSP permiten encapsular con un tag la generación de un trozo de HTML no en el propio JSP sino en ese tag que en código Java pudiendo incluir la llamada a un JSP. Los tags y librerías de tags son una forma de reutilizar esas partes de generación de código en el mismo proyecto y entre proyectos. Los tags además son una forma de abstraernos del funcionamiento interno del tag haciendo que solo necesitemos conocer sus parámetros.

Si usamos JSP usar librerías de tags es una buena idea, sin embargo, tiene algunas limitaciones como que requieren un archivo descriptor en formato XML que las defina y aunque pudiendo saber que parámetros definen y cuáles son requeridos no define el tipo de los parámetros que requiere. Los archivos XML en la época actual han caído en desuso porque son propensos a errores, errores que no son detectados hasta tiempo de ejecución, de los peores tipos de errores. Por otro lado, que los tags no especifiquen el tipo de parámetro que requiere cada uno hace que debamos inspeccionar el código fuente del tag con lo que la ventaja de abstraerse del funcionamiento no es del todo completa. Si por algún cambio el tipo de parámetro cambia hay que adaptar todos los usos del tag, si alguno no se hace nuevamente se producirán errores en tiempo de ejecución.

Grails usa GSP, una tecnología de presentación similar a los JSP. También dispone de tags que no requieren definir los tags en un archivo XML simplificando su uso pero que igualmente adolecen de algunos problemas como los JSP. Por un lado, los tags de Grails no disponen un mecanismo para hacer requerido un determinado parámetro con lo que deberemos incluir la comprobación con código nosotros, tampoco define el tipo de parámetros que requiere. También aunque hacer más simple su desarrollo al no tener un descriptor XML como en los tag JSP hace que haya que inspeccionar el código fuente para saber qué parámetros tiene, si son requeridos y cuál es el tipo del parámetro. Todo esto hace que puedan producirse errores en tiempo de ejecución y errores que no son producidos hasta que se ejercita el tag con un mal uso o un uso desactualizado al igual que usando los tag JSP.

En Apache Tapestry todo son componentes, las páginas también son componentes con la característica de que no están embebidos en otro componente. Un componente de Apache Tapestry sería similar a un tag de JSP o un tag de Grails, con ciertas similitudes pero no iguales en aspectos importantes. De pronto, un componente de Tapestry define los parámetros que necesita y si son requeridos pero también define el tipo del parámetro. Como se aprecia en las páginas de documentación de los componentes integrados de serie en Apache Tapestry se puede conocer esta información sin necesidad de conocer el código fuente del componente, documentación que podemos generar para los componentes que nosotros desarrollemos. Los parámetros, si son requeridos y sus tipos forman el contrato del componente y es lo único que deberemos conocer para usarlos, su funcionamiento interno nos es irrelevante que incluye el código JavaScript que necesite, podría que CSS y literales internacionalizados.

Pero esas no son las únicas diferencias con los tags de JSP o de Grails y es que estas son solo tecnologías de presentación, la V del patrón MVC. Los componentes de Tapestry aparte de encapsular la lógica de presentación también pueden encapsular lógica de controlador, en el conocido patrón MVC además de V pueden ser C con lo que encapsulan aún más funcionalidad. La lógica de presentación y controlador en los JSP y Grails está separada pero ambas lógicas no son independientes, están relacionadas, en Tapestry está encapsulada en el mismo componente.

Los componentes de Tapestry usan el modelo pull en vez del modelo push haciendo innecesario construir un objeto Map que pasar a la vista, haciendo que sea la plantilla la que solicite al controlador los datos que necesita y haciendo que el controlador no sepa que datos necesita la vista. El controlador solo deberá tener las propiedades y métodos que necesite la vista. Dado que en las plantillas tml de la vista no se pueden incluir expresiones complejas hace que no contengan lógica que estará en el controlador asociado que es código Java donde tendremos la ayuda del compilador para detectar errores.

Quizá la diferencia más importante es que los componentes de Tapestry soportan eventos, los tags simplemente son una tecnología de visualización por el contrario un componente tiene lógica, puede lanzar un evento y la reacción ser diferente según el comportamiento que desee el contenedor. La sección 3.12 está dedicada a los eventos, son usados abundantemente en el tratamiento de formuarios, peticiones AJAX y como resultado de acciones de enlaces y botones.

Modular, adaptable y extensible

Este es otro punto fuerte del framework. Usa su propio contenedor de dependencias (IoC, Inversion of Control) que viene incluido de serie en el framework encargándose de administrar los servicios, controlar su ciclo de vida, construirlos únicamente en el momento en que se necesitan y proporcionales las dependencias sobre otros servicios de los que hagan uso.

Usa su propio contenedor de dependencias porque no había ningún otro que permitiese una configuración distribuida. La configuración distribuida significa que para incluir nuevos servicios en el contenedor basta con dejar caer en la aplicación un jar y automáticamente los servicios y configuraciones que tenga ese jar serán administrados por el contenedor. Los servicios se definen en módulos, los módulos no son mas que clases Java especiales usadas para conocer los servicios y contribuciones del módulo. También posee un potente sistema de configuración, los servicios se configuran mediante contribuciones que cualquier módulo puede hacer, un módulo puede hacer contribuciones a servicios de otros módulos. El contenedor en el momento de construir el servicio le pasa el objeto con las contribuciones realizadas por cualquier módulo a ese servicio además de las dependencias que tenga sobre otros servicios.

Dado que mucha de la funcionalidad propia de Tapestry está proporcionada mediante servicios y que la implementación de un servicio puede reemplazarse por otra en el contenedor hace de él altamente adaptable y extensible tanto si necesitamos añadir nuevos servicios como si necesitamos que los existentes se comporten de otra forma, solo deberemos proporcionarle al contenedor la interfaz del servicio (sólo si es nuevo) y la implementación que deseamos. Con unas pocas líneas se puede personalizar casi todo aunque a veces puede llevar un tiempo saber cuales son esas líneas.

Toda esta definición de servicios y configuraciones se hace a través de código Java con lo que tendremos la ayuda del compilador y cualquier error de escritura, al contrario de lo que ocurre en archivos XML, lo detectaremos rápidamente.

Convención sobre configuración

Las convenciones permiten evitar la configuración y los posibles errores que podemos cometer al realizarla. Pero más importante, hace que cualquier programador que conozca las convenciones sepa inmediatamente como están organizadas todas las cosas con lo que el tiempo de aprendizaje se reduce considerablemente. Por estos motivos Tapestry es un framework en el que se usan varias convenciones.

De esta manera los XML interminables propensos a errores pueden ser erradicados de la aplicación, esto se consigue con una mezcla de inyección de dependencias proporcionada por el contenedor IoC y metaprogramación proporcionada por anotaciones y convenciones de nomenclatura.

Documentado

Ya tiene una década y todo este tiempo ha servido para que tenga una documentación bastante extensa y de calidad que por si sola sirve para aprender cada concepto de este framework de forma autodidacta. Además de la documentación de los conceptos está disponible el correspondiente Javadoc y la documentación de los componentes como consulta para el desarrollo.

Existen otros varios libros escritos como Tapestry 5 - Rapid web application development in Java de 482 páginas y tiene listas de distribución tanto para usuarios como para desarrolladores bastante activas con gente dispuesta a ayudar.

Informativo

A pesar de que pueda pasar desapercibida es una de las mejores cosas que tiene Tapestry cuando las cosas van mal y se producen excepciones en el servidor. Cuando ello ocurre el framework recopila toda la información de la que dispone y genera un informe de error muy completo que incluye desde la traza de la excepción, la línea exacta del archivo tml mostrando un extracto del código fuente del mismo así como otra información de la petición tales como los parámetros, diversa información que envió el navegador en las cabeceras, nombre y valores de la sesión y las propiedades de entorno del sistema. Por si fuera poco el informe de error también es generado para las peticiones Ajax.

Toda esa información precisa nos sirve de gran ayuda para descubrir más rápidamente y fácilmente cual fue la causa del error haciendo que tardemos menos en corregir los errores.

Productivo

Tapestry es un framework con el que se es bastante productivo. Por la facilidad para encapsular funcionalidad en componentes que son fáciles de crear y reutilizar. Por la recarga en caliente de los cambios que permiten ver los cambios inmediatamente evitando reinicios del servidor y esperas. Y por ser un framework que proporciona mucha información cuando se produce un error en forma de excepción que ayuda a resolver los problemas rápidamente.

Por estas tres cosas entre otros detalles se consigue una alta productividad.

1.3 Un poco de historia

El framework fue ideado por Howard Lewis Ship (HLS) en el año 2000 cogiendo ideas similares al WebObjects de esa época. En el 2006 se gradúa como un proyecto de alto nivel de la fundación Apache. HLS en el año 2010 recibe el premio Java Champion.

En cada nueva versión la forma de hacer las cosas cambian aplicando nuevas ideas que facilitan el desarrollo, estos cambios hacían que el paso de una versión mayor a otra no fuese simple. Ya en la versión 5 este problema se soluciona en gran medida y los cambios se limitan a no usar las cosas marcadas como obsoletas o que fueron quitadas.

Tapestry 3

En esta versión los componentes poseen 2 archivos como mínimo, uno para el código Java y otro jwc para la especificación del componente o page para la especificación de la página aunque normalmente suele incluir otro más para la plantilla que genera el html. Además, pueden necesitar otros recursos de estilos, imágenes o archivos de internacionalización. Para hacer un nuevo componente se ha de utilizar herencia extendiendo de las clases de Tapestry.

Tapestry 4

Esta versión hace varias aportaciones entre las principales incluir su propio contenedor de dependencias, Hivemind, también desarrollado por HLS, que hace uso de XML para su configuración. Se sigue teniendo que extender de clases del framework y los archivos jwc y page siguen existiendo.

Tapestry 5

Esta versión sigue suponiendo una nueva ruptura con la versión anterior. Las mejoras que incluye son numerosas, se hace un uso extensivo de las nuevas características de Java como las anotaciones y generics y se desarrolla un módulo de contenedor de dependencias más integrado con el framework. Ahora en vez de usar un XML

para la definición del contenedor IoC se usan clases Java. Ya no es necesario extender de clases de Tapestry, esto hace que las clases de componentes y páginas sean POJO que simplifica las pruebas unitarias. Se proporcionan numerosas anotaciones que describen las clases y que en tiempo de ejecución añaden la funcionalidad. Las anotaciones hacen innecesario el archivo jwc de manera que no tenemos que mantenerlo sincronizado con el código Java y tml. Se añade la carga en caliente de los cambios que permiten aumentar la productividad. Se hace políglota soportando cualquier lenguaje ejecutable en la JVM. Deja de usar las expresiones ognl en las plantillas y desarrolla un lenguaje de expresiones similar. Se liberan varias versiones menores 5.1, 5.2, 5.3 en los que cambiar a la nueva versión es poco más que actualizar las dependencias, las actualizaciones son mucho más pacíficas gracias a las anotaciones y la no herencia.

Ha sido un líder desde una perspectiva puramente tecnológica. Estas son algunas cosas que hizo primero y todavía su autor, Howard Lewis Ship, piensa que lo hace mejor que nadie:

- Componentes reusables (2001)
- Detallado y útil informe de excepciones (2001)
- Instrumentación invisible en plantillas (2002)
- Informe de excepción con extractos de código fuente (2004)
- Metaprogramación de bytecode integrada (2005)
- Recarga en caliente de cambios (2006)
- Informe completo para errores en peticiones Ajax (2012)

1.4 Opciones alternativas

Si aún así lo que te cuento en este libro no te convence (espero que lo haga al menos un poco) dispones de varias alternativas en otros lenguajes y dentro de la misma plataforma Java. Aunque las que pondré a continuación son basadas en acciones y no en componentes. Muchos de los siguientes se diferencian en poco del modelo básico de arquitectura modelo-vista-controlador que en la plataforma Java Struts fue uno de sus máximos precursores. Aunque nuevos frameworks publicados posteriormente simplifican la forma de codificar muchas de las tareas siguiendo convenciones, DSL y requiriendo menos archivos cuyo contenido hay que mantener sincronizado pero en esencia no siguen siendo muy distintos de Struts con sus aciertos y defectos. La mayor diferencia que se puede encontrar entre ellos es el lenguaje de programación empleado.

Java

La cantidad de frameworks y librerías en Java es muy numerosa, según sean los requisitos de la aplicación y después de nuestras preferencias podemos seleccionar opciones diferentes o alternativas. Dentro del grupo de frameworks basados en acciones tenemos Spring MVC, Struts, Play! o para algo simple Spark Framework. Dentro del grupo de los basados en componentes el estándar de la plataforma Java EE, JSF.

Si la aplicación debe ser altamente eficiente y requerir menos recursos podemos optar por Vert.x basado en eventos y en la programación reactiva aunque significativamente diferente de la imperativa también usada en los lenguajes orientados a objetos, es el equivalente a Node.js de JavaScript en la plataforma Java.

PHP

Es un lenguaje muy popular para el desarrollo de páginas web. Hay varios frameworks basados en este lenguaje con librerías de funcionalidad similar alternativas a las que encontramos en la plataforma Java. Algunas opciones son: Symfony, Silex, CakePHP, CodeIgniter.

Python

Es un lenguaje interpretado que desde hace un tiempo ha ido ganando popularidad. Tiene una sintaxis limpia y genera código legible. Es un lenguaje que soporta varios paradigmas, orientado a objetos, funcional, imperativo y de tipado dinámico. Hay varios frameworks web basados en Python el más popular Django.

Groovy

Es un lenguaje que se ejecuta en la JVM que hace innecesario mucho del código que usaríamos en Java para hacer lo mismo. Soporta closures y DSL. El tipado puede ser dinámico o estático y puede ser interpretado. El framework web más popular es Grails.

C#

En la plataforma .NET Microsoft ha ido evolucionando sus herramientas para el desarrollo de aplicaciones web, de Web Forms se ha pasado a ASP.NET MVC de características más similares a frameworks basados en acciones.

Ruby

Este lenguaje se define así mismo como dinámico de sintaxis elegante natural al leerla y fácil de escribirla enfocado a la simplicidad y productividad. Es el lenguaje empleado en el framework Ruby on Rails.

Figura 1.1: Arquitectura del modelo de tres capas

1.5 Arquitectura de aplicaciones web

Modelo de 3 capas

Las aplicaciones se han de organizar de alguna forma en partes, haciendo que cada una se centre en una responsabilidad permite puedan ser modificada sin afectar de forma considerable al resto. En las aplicaciones web es habitual seguir el modelo de tres capas, dividido en:

- Capa de presentación: se encarga de generar la interfaz de usuario y permitir la interacción. En las aplicaciones web la interfaz de usuario consiste en el html, las imágenes, las hojas de estilo, el javascript, la internacionalización y localización que se mostrarán al usuario a través del navegador con el que acceda el usuario a la aplicación. Se encarga de ser la interfaz entre el usuario y la lógica de negocio. En esta capa la aplicación se ejecuta en el navegador del usuario pero habitualmente se genera en el servidor.
- Lógica de negocio: es la parte que tiene el conocimiento del ámbito que maneja la aplicación. Está compuesto por diferentes entidades denominadas servicios que a su vez se encargan de una parte individual de la lógica, también suele incluir las entidades de dominio persistentes en una base de datos. Es utilizada por la capa de presentación y utiliza la capa de datos. Esta capa se ejecuta en el servidor de aplicaciones o de negocio junto con el framework web que genera el código para la capa de presentación.
- Capa de datos: guarda de forma permanente los datos manejados por la aplicación hasta que se requiera su uso de nuevo, habitualmente en una base de datos relacional aunque hay otras opciones. Es utilizada por la capa de lógica de negocio. Esta capa suele tener un servidor de bases de datos.

Modelo cliente/servidor

Las tres capas anteriores son independientes y se comunican a través de la red donde en cada par una parte actúa de cliente y otra de servidor. En el caso de la capa de lógica de negocio actúa de servidor para la capa de presentación pero como cliente para la capa de datos. Cada capa de las anteriores es lógica no física, es decir, pueden ejecutarse en la misma máquina (como puede ser en caso en el momento de desarrollo) o cada una en una máquina diferente (como será el caso del momento de producción).

Figura 1.2: Modelo cliente servidor

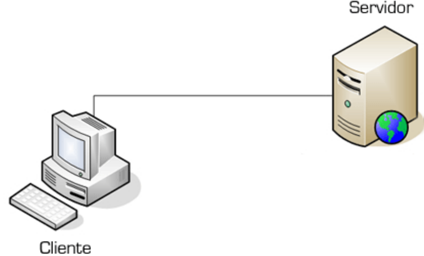
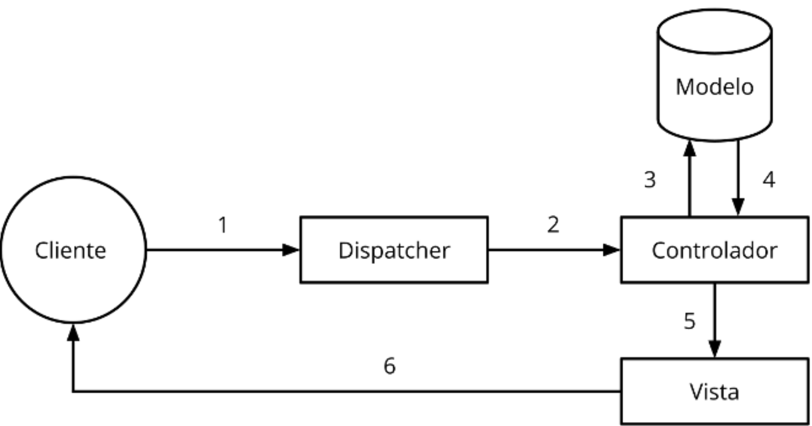


Figura 1.3: Modelo vista controlador (push)



Modelo Vista Controlador

El patrón modelo vista controlador (MVC, Model View Controller) es muy usado en los frameworks web. Separa cada una de las partes de la aplicación tratando de que sean independientes para minimizar el cambio que una parte puede provocar en otra. El modelo contiene los datos y junto con los servicios la lógica de la aplicación que permite manipularlos. La vista proporciona una representación del modelo para el usuario y es la interfaz para producir las acciones, finalmente el controlador en base a las acciones realizadas por el usuario se encarga de usar los servicios, manipular el modelo y proporcionar una nueva vista al usuario. El modelo MVC empleado por Tapestry es un tanto diferente del empleado normalmente en los frameworks basados en acciones, como veremos el controlador y la vista en Tapestry están más íntimamente unidos.

Modelo «push» contra modelo «pull» en frameworks web

En la mayoría de frameworks de desarrollo de aplicaciones o páginas web para producir el contenido HTML que se envía al cliente se emplea un modelo en el que el controlador proporciona los datos que combinados

Figura 1.4: Modelo vista controlador (pull)

con una plantilla producen el HTML. Este modelo también es el empleado habitualmente en muchos motores de plantillas (thymeleaf, mustache, ...). Sin embargo, hay dos modelos que se pueden seguir para producir un texto como resultado dada una plantilla y datos:

- Push: este es el modelo comentado. El controlador recupera de antemano todos los datos que necesita la vista, el controlador también determina la vista o plantilla que se usar. Combinando los datos y la plantilla se produce el resultado.
- Pull: en este modelo el controlador no conoce los datos que usará la vista y es esta la que los solicita según necesita. La vista tira del controlador, el controlador solo debe ofrecer el soporte par que la vista pueda recuperar los datos que necesite.

Los pasos que se siguen en el modelo push son (ver figura Modelo vista controlador (push)):

- La petición llega al servidor
- El dispatcher redirige la petición al controlador
- El controlador solicita los datos a la base de datos
- El controlador obtiene los datos de la base de datos
- El controlador redirige a la vista y le envía los datos que necesita
- La vista genera el contenido y se envía al cliente

Los pasos que se siguen en el modelo pull varían ligeramente del modelo push pero de forma importante, son:

- La petición llega al servidor
- El dispatcher redirige la petición al controlador

- El controlador puede acceder a la base de datos previamente a redirigir a la vista
- La vista pide los datos que necesita al controlador y el controlador devuelve los recuperados previamente o los pide a la base de datos
- La vista obtiene los datos que ha pedido del controlador
- La vista genera el contenido y se envía al cliente

El modelo push es empleado en muchos de los frameworks web más usados, algunos ejemplos son Symfony, Django, Grails o ASP.NET MVC. En la categoría de frameworks que usan un modelo pull está Apache Tapestry.

El modelo push puede presentar algunos problemas. Un de ellos es que el controlador debe conocer que datos necesita la vista y si la vista tiene cierta lógica esta la tendremos duplicada tanto en en controlador como en la vista. Supongamos que en una aplicación tenemos un usuario y dirección con una relación de 1 a 1 entre ambos y que debemos mostrar en una página el usuario y su dirección solo si solo si es un usuario VIP. En el controlador tendremos que recuperar el usuario, comprobar si es VIP y si lo es recuperar su dirección. El problema está que en la vista deberemos hacer también una comprobación si el cliente es VIP o al menos si a la vista se le ha proporcionado una dirección, como resultado la comprobación la tendremos duplicada tanto en el controlador como en la vista, como sabemos la duplicación de código y lógica habitualmente no es buena idea ya que a la larga dificulta el mantenimiento de la aplicación si es que peor aún produce algún error.

En Grails (pero podría ser cualquier otro framework o motor de plantillas push) podríamos visualizar el usuario y su dirección si es VIP de la siguiente forma:

```
// Grails
// Controlador (groovy)
def showUsuario() {
    def usuario = Usuario.get(params.long('id'))
    def direccion = null
    if (usuario.isVIP()) {
        direccion = usuario.direccion
    }
    render(view:'show', model: [usuario:usuario, direccion:direccion])
}

// Vista (gsp)
Nombre: ${usuario.nombre}
<g:if test="${usuario.vip}">
    Dirección: ${direccion.toString()}
</g:if>
```

Si usamos hibernate la recuperación de la dirección podemos hacerla navegando la relación pero he querido recuperarla en el controlador expresamente para el ejemplo, si no usásemos hibernate para recuperar el dato relacionado probablemente lo que haríamos es recuperar el dato en el controlador como en el ejemplo.

Otro problema del modelo push es que si la vista es usada en múltiples controladores, y precisamente la separación entre vistas y controladores uno de sus motivos es para esto, todos estos controladores van a compartir

el código para recuperar los datos que necesite la vista, dependiendo del número de datos y de veces que empleemos una vista en múltiples controladores quizá debamos hacer una clase asociada a la vista que recupere los datos para evitar tener código duplicado (y exactamente esto es lo que se hace en el código Java de los componentes de Tapestry).

En el modelo pull el controlador no debe conocer que datos necesita la vista y si hay lógica para mostrar ciertos datos está lógica solo la tendremos en la vista. Aunque el controlador no deba conocer que datos en concreto necesite la vista si debe ofrecer el soporte para que la vista los recupere cuando necesite. Como se puede ver el código en el siguiente ejemplo la comprobación de si el usuario es VIP solo está en la vista. En Tapestry cada vista tiene asociado una clase Java que es la encargada de ofrecer el soporte para que la vista pueda recuperar los datos, el conjunto de controlador más vista es lo que en Tapestry se conoce como componente, si el componente se usa varias veces en el mismo proyecto no necesitamos duplicar código.

```
// Tapestry
// Controlador (java)
public Usuario getUsuario() {
    return usuarioDAO.get(id);
}

public Direccion getDirecion() {
    return getUsuario().getDireccion();
}

// Vista (tml)
Nombre: ${usuario.nombre}
<t:if test="usuario.vip">
    Direccion: ${direccion.toString()}
<t:if>
```

¿Podemos emplear un modelo pull en un framework que normalmente se suele usar un modelo push? Sí, basta que en el modelo de la vista pasemos un objeto que le permita recuperar los datos que necesite. En Grails empleando un modelo pull el código podría quedarnos de la siguiente forma:

```
// Grails
// Controlador (groovy)
def showUsuario() {
    render(view:'show', model: [view:new View(params)])
}

private class View {

    Map params

    View(Map params) {
        this.params = params
    }

    def getUsuario() {
        return Usuario.get(params.long('id'))
```

```
    }

    def getDireccion() {
20      return usuario.direccion
    }
}

// Vista (gsp)
25 Nombre: ${view.usuario.nombre}
<g:if test="${view.usuario.vip}">
    Dirección: ${view.direccion.toString()}
</g:if>
```

Como se ve el if de comprobación en el controlador desaparece, a pesar de todo si la vista fuese usada por varios controladores deberíamos crear algo para evitar tener duplicado el código que permite recuperar los datos a la vista. Aunque esto es perfectamente posible no es la forma habitual de usar los frameworks que siguen el modelo push.

Este ejemplo es muy sencillo y empleando cualquiera de los dos modelos es viable, pero cuando el número de datos a recuperar en las vistas y el número de veces que se reutiliza una vista aumenta (y en teoría la separación entre controlador y vista uno de sus motivos es posiblemente para reutilizarlas) el modelo push presenta los problemas que he comentado que el modelo pull no tiene.

Modelos de datos anémicos

Un modelo de datos anémico es aquel que apenas tiene lógica de negocio en las propias entidades y prácticamente sirve para actuar como contenedores para transferencia de datos (DTO, Data Transfer Object). Son criticados porque no utilizan correctamente el paradigma de la programación orientada a objetos donde se recomienda que aquella lógica le pertenezca a una entidad sea incluida en la misma, de esta forma los datos están encapsulados y solo pueden manipularse mediante la lógica de la entidad de una forma correcta. Sin embargo, esto tampoco significa incluir cualquier cosa en la entidad, cuando una entidad tiene demasiados imports o a ciertos servicios es indicativo de que tiene responsabilidades que no le corresponden.

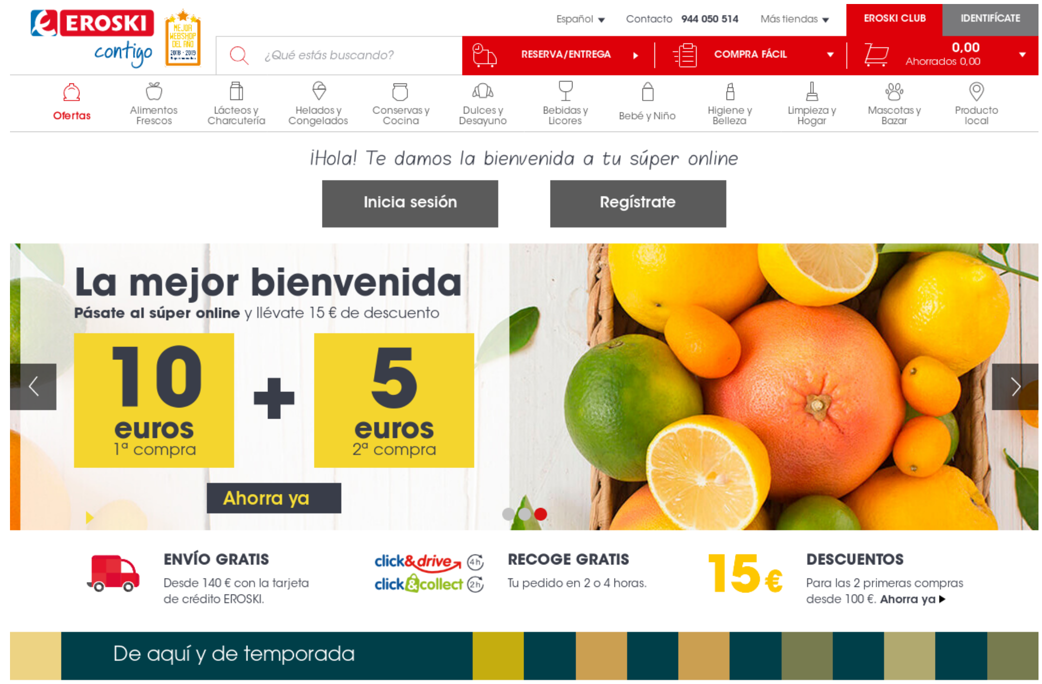
Supongamos que tenemos una entidad que cuando se hace una operación en ella debe hacer un cálculo y mandar un correo electrónico. En el cálculo si no intervienen muchas otras entidades y sólo depende de datos propios de esa entidad es adecuado incluirlo en esa entidad. Pero enviar el correo electrónico probablemente sea proporcionado por un servicio externo ¿debería ser la entidad la que enviase el mensaje haciendo uso de ese servicio? Hacer que la entidad además del cálculo envíe un correo electrónico quizá sea demasiada responsabilidad. Se puede utilizar un servicio que orqueste ambas acciones, el cálculo y el envío del correo electrónico mediante su servicio, también se podría usar un servicio para los casos en que una lógica implique acciones sobre varias entidades.

Evitar un modelo de datos anémico es buena idea pero dar demasiadas responsabilidades a la entidades no significa que sea buena idea, en definitiva conviene evitar los modelos de datos anémicos pero también los modelos supervitaminados.

1.6 Casos de éxito y de referencia

En la página BuiltWith recopilan un montón de sitios web que usan Apache Tapestry y son ejemplos de casos de éxito usándolo. Voy a destacar uno que no aparece en esa página y que además es una web española, el Supermercado online de Eroski. Eroski es una empresa vasca dedicada a la alimentación, distribución y comercialización, tiene su sede en Elorrio, Vizcaya, País Vasco con más de 36.000 trabajadores en plantilla. La web es la parte dedicada a la compra online con un muy buen aspecto, buen diseño y con un resultado final excelente.

Se que utiliza Apache Tapestry porque inspeccinonando su código fuente así lo demuestra, con una etiqueta meta generator con el valor Apache Tapestry Framework y los archivos de recursos característicos del framework. No usa la última versión disponible pero si una reciente, la 5.4.1.



Capítulo 2

Inicio rápido

El capítulo anterior ha sido muy teórico pero había que contarla, aún no hemos visto mucho código de como se hacen las cosas en Tapestry. Este capítulo es una guía de inicio rápida en la que veremos como tener un esqueleto de aplicación de la que podemos partir en unos pocos minutos y con la que podrás empezar a sacar conclusiones por ti mismo no a partir de una descripción de la características principales sino de la experiencia donde se ven mucho mejor los detalles menores que aún no he contado. También veremos como tener un entorno de desarrollo y que herramientas podemos utilizar. Puedes partir de la aplicación que crearemos para ir probando el contenido de los capítulos siguientes en el libro.

2.1 Instalación JDK

Lo único realmente necesario para desarrollar con Tapestry es el JDK (Java Development Kit) en su versión 1.5 o superior. Esta es la versión mínima necesaria ya que Tapestry necesita y aprovecha muchas de las características que se añadieron a partir de esta versión y que también podremos aprovechar para nuestra aplicación como los generics, anotaciones, enums, varargs y algunas otras cosas más. La instalación dependerá del sistema operativo, en windows o mac os x descargaremos la última versión y en linux nos sera mas cómodo utilizar el paquete openjdk de la distribución que usemos.

Para la futura versión de Tapestry 5.5 es muy posible que la versión mínima del JDK y Java sea la 8, siendo compatible con la versión 11 con soporte a largo plazo (LTS).

2.2 Inicio rápido

Un proyecto web en Java requiere de unos cuantos archivos con cierta estructura que nos puede llevar un tiempo en crearlos. Normalmente cuando empezamos un nuevo proyecto solemos basarnos en otro existente copiando y pegando contenido de él. Pero ademas de tiempo podemos cometer errores o no seguir algunas convenciones propias de Java o del framework web que usemos. Para un proyecto grande esa dedicación al inicio del proyecto

no nos importará pero para un proyecto pequeño o para hacer una prueba puede que queramos tener algo más rápido y con menos esfuerzo para estar en disposición de empezar a desarrollar en muy poco tiempo.

Para crear el esqueleto de una aplicación rápidamente en Apache Tapestry hay disponible un arquetipo de Maven que puede generar una aplicación en unos pocos minutos. Para usarlo deberemos instalar Maven previamente. Una vez instalado Maven basta con que usemos el siguiente comando.

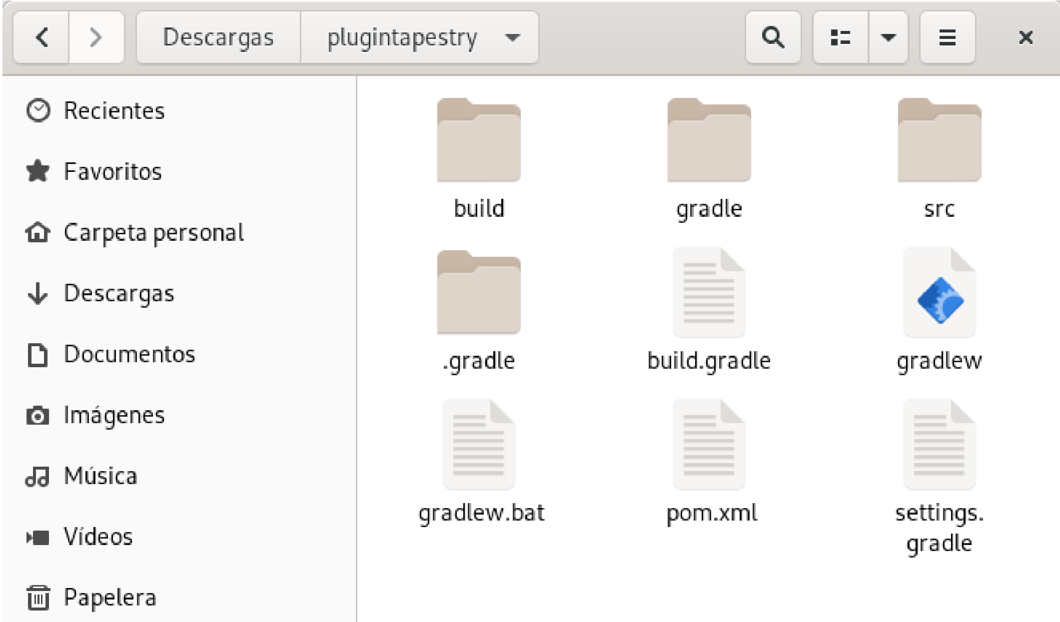
Listado 2.1: mvn.sh

```
1 $ mvn archetype:generate -B \
2        -DarchetypeGroupId=org.apache.tapestry \
         -DarchetypeArtifactId=quickstart \
         -DarchetypeVersion=5.5.0 \
         -DgroupId=io.github.picodotdev \
         -DartifactId=plugintapestry \
7        -Dpackage=io.github.picodotdev.plugintapestry \
         -Dversion=1.0
```

Este comando requiere que los cambios de esta petición en el JIRA de Tapestry sea resuelta.

Explorando los archivos generados

Aunque el arquetipo lo realizamos con Maven los archivos que genera son válidos tanto para trabajar con Maven como con Gradle (opción que recomiendo por sus ventajas), una vez que tenemos la aplicación generada podemos usar el que prefiramos. Los archivos generados son los siguientes:



La estructura de archivos del proyecto generado es la de un proyecto maven. En src/main tendremos tres carpetas:

- java: donde estarán ubicadas las clases Java de nuestro proyecto, tanto de los servicios, del código Java asociado a los componentes y paginas, de utilidad, las entidades que persistiremos en la base de datos, etc...

- resources: en esta carpeta estarán el resto de archivos que no son archivos Java como puede ser la configuración de logging, archivos de localización, plantillas de los componentes o páginas, etc... Una vez construido el war todos estos archivos se colocaran en la carpeta WEB-INF/classes junto con las clases compilada de los archivos java.

- webapp: esta carpeta será el contexto web de la aplicación, podemos colocar las imágenes, css, archivos javascript.

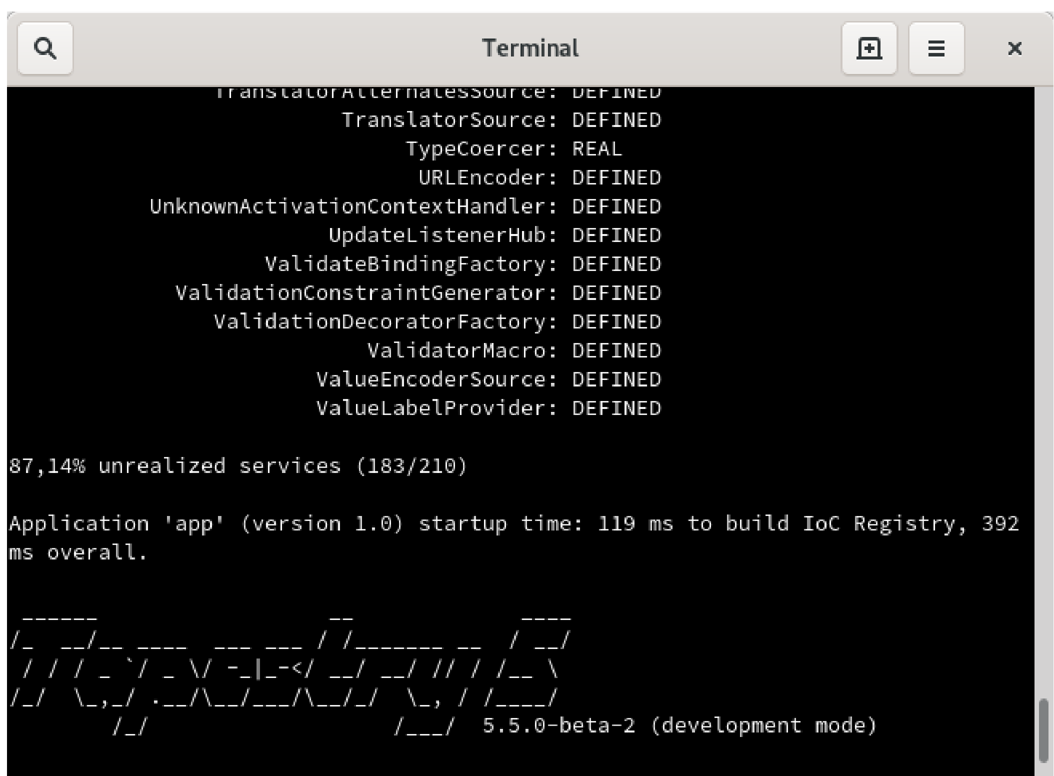
Dentro de la carpeta webapp/WEB-INF tendremos el archivo web.xml que describirá algunas cosas importantes de la aplicación. Si lo abrimos veremos que Tapestry funciona a través de un filtro y no de un servlet, esto es así porque también se encarga de procesar los recursos estáticos lo que proporciona algunas funcionalidades adicionales como compresión, minimización y localización. Otra cosa importante definida en este archivo es el paquete de la aplicación Tapestry con el parámetro de contexto tapestry.app-package en el cual por convención se buscarán los componentes, páginas, servicios y módulos de la aplicación. El paquete de la aplicación que usaré es io.github.picodotdev.plugintapestry. Con el arquetipo se crean varios módulos para el contenedor de dependencias tapestry-ioc, AppModule es el único necesario, en su estado inicial define los locales que soportara la aplicación y número de versión, además existen aunque realmente no son necesarios, DevelopmentModule hace que la aplicación se arranque en modo desarrollo y QaModule para las pruebas automatizadas, no usaremos estos dos últimos.

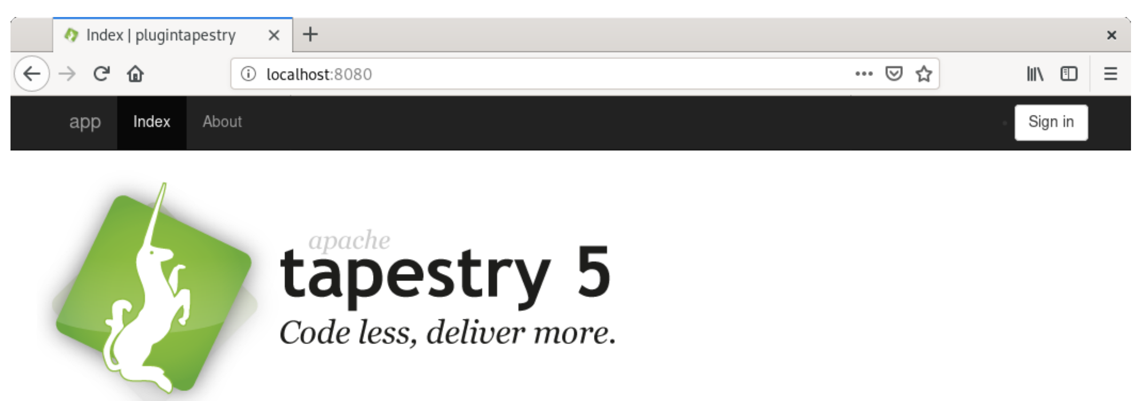
Otro archivo importante es build.gradle y gradlew (o gradle.bat) con el que podremos automatizar diversas tareas del ciclo de vida del proyecto usando la herramienta de construcción Gradle. Gradle tienen varias ventajas sobre Maven entre ellas que no se usa un XML para la descripción del proyecto sino un archivo basado en un DSL del lenguaje groovy, lenguaje mucho más apropiado para programar que el XML de Maven o Ant.

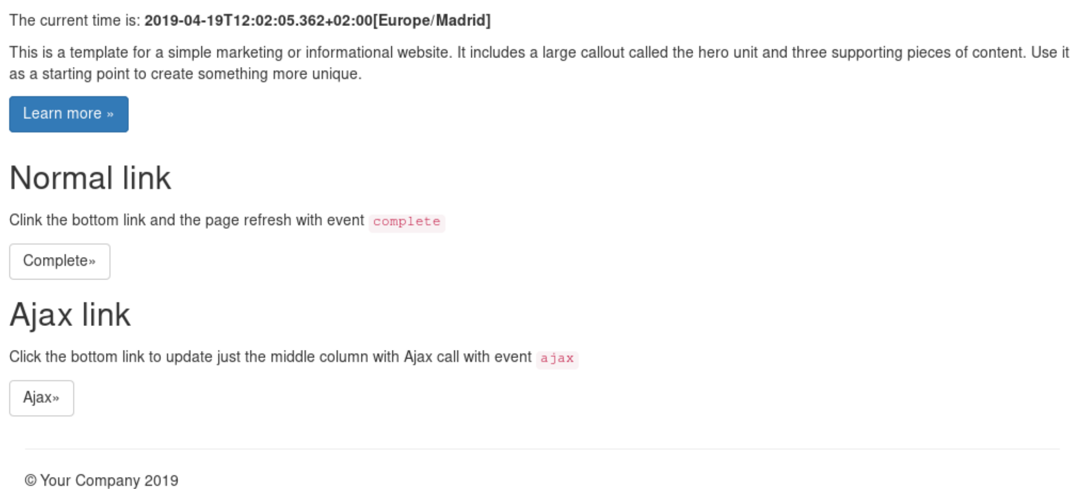
Una vez generada la aplicación podemos iniciarla con un servidor embebido Jetty con la aplicación desplegada en él ya usando Gradle:

```
1 $ ./gradlew run
```

Y accediendo con el navegador a la URL que nos indica Tapestry al final de las trazas veremos la aplicación en funcionamiento.


Terminal
TranslatorSource: DEFINED
TypeCoercer: REAL
URLEncoder: DEFINED
UnknownActivationContextHandler: DEFINED
UpdateListenerHub: DEFINED
ValidateBindingFactory: DEFINED
ValidationConstraintGenerator: DEFINED
ValidationDecoratorFactory: DEFINED
ValidatorMacro: DEFINED
ValueEncoderSource: DEFINED
ValueLabelProvider: DEFINED
87,14% unrealized services (183/210)
Application 'app' (version 1.0) startup time: 119 ms to build IoC Registry, 392 ms overall.
5.5.0-beta-2 (development mode)

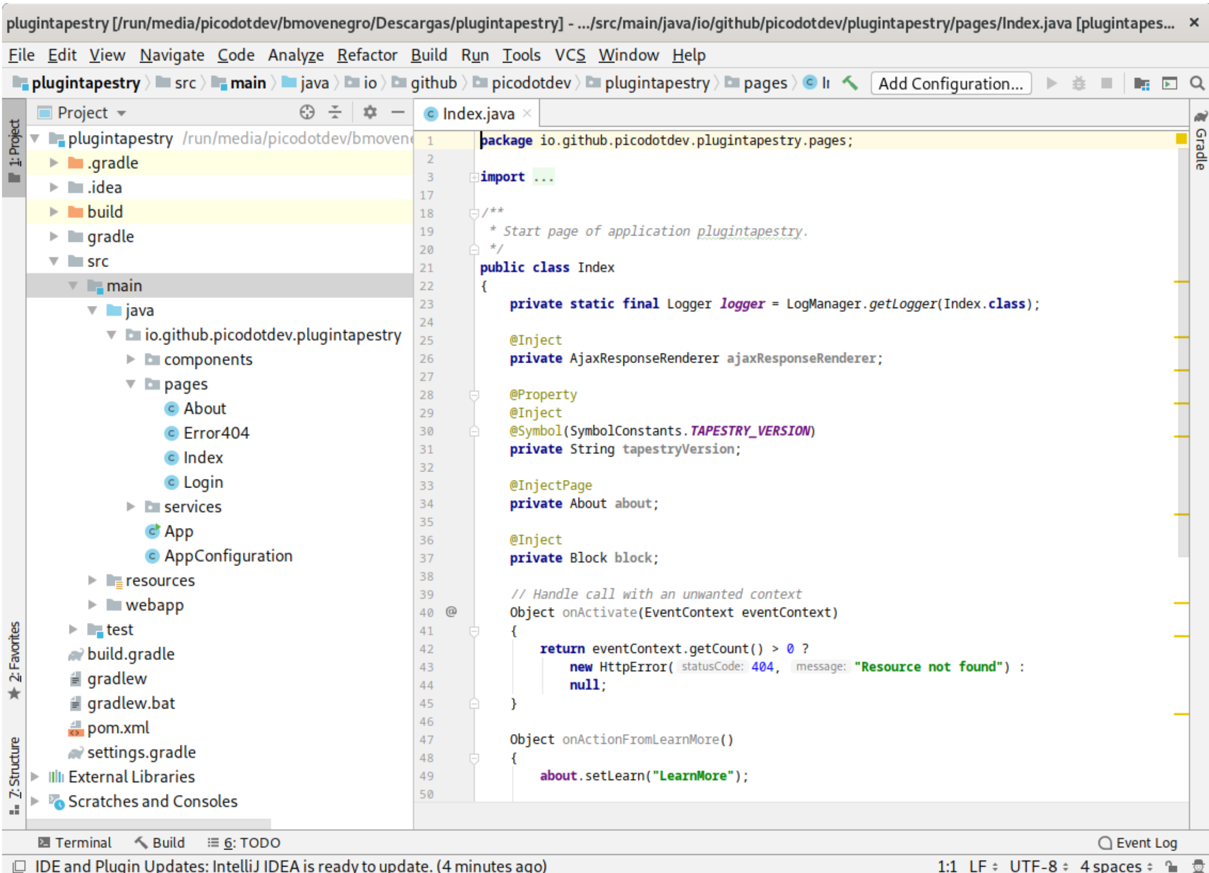




Tapestry no es un framework fullstack y es responsabilidad nuestra disponer de esas características si necesitamos pero con la libertad de elegir las herramientas que nosotros decidamos. En definitiva, con este arquetipo de Maven en unos pocos minutos y con poco esfuerzo podemos disponer de una aplicación Apache Tapestry a partir de la que empezar a desarrollar.

2.3 Entorno de desarrollo

Habiendo arrancado la aplicación y accedido a ella con el navegador ya podemos empezar explorar el código y tal vez hacer alguna modificación. Pero para desarrollar probablemente necesitaremos un IDE (Integrated Development Enviroment, entorno integrado de desarrollo). Aunque probablemente cada desarrollador tendrá sus preferencias de herramientas con las que mas cómodo se siente al programar. Mi preferencia es IntelliJ DEA ya que ofrece asistencia al escribir el código java, posee resaltado de sintaxis tanto para archivos Java como html/tml/css/javascript, los errores de compilación son notificados inmediatamente, es posible hacer refactors y renombrados que sustituyen todas las referencias y se integra con el sistema de control de versiones svn o git, además de poder hacer debug. Algunas otras opciones son: eclipse, Netbeans, Visual Studio Code o vim.



Como herramienta de construcción recomiendo usar Gradle en vez de maven ya que tiene varias ventajas. El arquetipo de maven ya nos crea un archivo básico de construcción gradle.

2.4 Integración con el servidor de aplicaciones

Para desarrollar deberíamos usar el mismo servidor de aplicaciones en el que se vaya a ejecutar la aplicación en el entorno de producción para evitar sorpresas. Ya sea Tomcat, JBoss/Wildfly, Jetty, Weblogic u otro. A continuación explicaré como ejecutar nuestra aplicación de tres formas diferentes:

- Con Spring Boot podremos tanto en desarrollo como en producción ejecutar la aplicación de forma «standalone» con un servidor embebido Tomcat, Jetty o Undertow desde Gradle o generando un archivo jar ejecutable.
- Generando un archivo war para desplegarlo de forma tradicional en estos u otros servidores.
- Para desarrollo también podremos usar cualquier servidor de aplicaciones de forma externa.

2.4.1 Spring Boot

Tradicionalmente las aplicaciones Java web han sido instaladas en un contenedor de servlets como Tomcat o Jetty y WildFly, JBoss o Weblogic si necesita más servicios que son ofrecidos por la plataforma Java EE comple-

ta como JMS, JPA, JTA o EJB. Aunque las aplicaciones se ejecutan independientemente unas de otras comparten el entorno de ejecución del servidor de aplicaciones, algunas aplicaciones no necesitarán todos los servicios que ofrecen los servidores de aplicaciones en su implementación del perfil completo Java EE y algunas nuevas aplicaciones pueden necesitar hacer uso de una nueva versión de un servicio como JMS con funcionalidades mejoradas. En el primer caso algunos servicios son innecesarios y en el segundo la actualización del servidor de aplicaciones se ha de producir para todas las aplicaciones que en él se ejecuten o tener varias versiones del mismo servidor de aplicaciones e ir instalando las aplicaciones en la versión del servidor según las versiones de los servicios para las que se desarrolló la aplicación.

Los microservicios proponen una aproximación diferente al despliegue de las aplicaciones prefiriendo entre otros aspectos que sean autocontenidos de tal forma que puedan evolucionar independientemente unas de otras. Una forma de hacer la aplicación autocontenida es con Spring Boot, internamente usa una versión embebible del servidor de aplicaciones de la misma forma que lo podemos usar directamente, la ventaja al usar Spring Boot es que soporta Tomcat, Jetty o Undertow y pasar de usar uno a otro es muy sencillo y prácticamente transparente para la aplicación, además proporciona algunas características adicionales como inicializar el contenedor IoC de Spring, configuración, perfiles para diferentes entornos (desarrollo, pruebas, producción), información, monitorización y métricas del servidor de aplicaciones y soporte para la herramienta de automatización Gradle entre algunas más. En el ejemplo asociado al libro usaré Spring Boot junto con Gradle.

Deberemos añadir algo de configuración en el archivo build.gradle para añadir las dependencias de Spring Boot y su plugin.

Listado 2.2: build.gradle

```
1  plugins {
       id 'eclipse'
       id 'idea'
4      id 'java'
       id 'groovy'
       id 'war'
       id 'application'
       id 'project-report'
9      id 'pmd'
       id 'org.springframework.boot' version '2.1.4.RELEASE'
       id 'nu.studer.jooq' version '3.0.3'
       id 'org.liquibase.gradle' version '2.0.1'
   }
14
   description = 'PlugInTapestry application'
   group = 'io.github.picodotdev.plugintapestry'
   version = '1.8'
   mainClassName = 'io.github.picodotdev.plugintapestry.Main'
19
   sourceCompatibility = "1.8"
   targetCompatibility = "1.8"

   ext {
24     versions = [
           tapestry:              '5.4.4',
```

```
        tapestrySecurity:     '0.7.1',
        tapestryTestify:      '1.0.4',
        tapestryXpath:        '1.0.1',
29      commons_dbcp:         '1.4',
        hibernate:            '5.1.16.Final',
        hibernateValidator:   '6.0.13.Final',
        geb:                  '2.3.1',
        groovy:               '2.5.4',
34      guava:                '27.0.1-jre',
        h2:                   '1.4.197',
        htmlunitDriver:       '2.33.3',
        htmlunitCsspaser:     '1.2.0',
        jackson:              '2.9.6',
39      javaee:               '8.0',
        json:                 '1.1.2',
        jooq:                 '3.11.10',
        junit:                '4.12',
        liquibase:            '3.6.2',
44      liquibaseGroovyDsl:   '2.0.2',
        log4j2:               '2.11.1',
        mockito:              '2.23.4',
        selenium:             '3.141.59',
        shiro:                '1.3.2',
49      spock:                '1.3-RC1-groovy-2.5',
        spring:               '5.0.11.RELEASE',
        springBoot:           '2.1.4.RELEASE',
        jbossVfs:             '3.2.14.Final',
        jbossLogging:         '3.3.2.Final',
54      webjars: [
            requirejs:        '2.3.5',
            jquery:           '3.3.1-1',
            underscore:       '1.9.1',
            bootstrapSelect:  '1.13.8'
59      ],
        yasson:               '1.0.1'
    ]
}

64 repositories {
    mavenLocal()
    mavenCentral()
    maven {
        url 'https://repository.apache.org/content/repositories/staging/'
69  }
}

// This simulates Maven's 'provided' scope, until it is officially supported by Gradle
// See http://jira.codehaus.org/browse/GRADLE-784
74 configurations {
```

```
    provided
    providedRuntime
    appJavadoc
}

sourceSets {
    main {
        compileClasspath += configurations.provided
    }
    test {
        compileClasspath += configurations.provided
        runtimeClasspath += configurations.provided
    }
}

configurations.all {
    resolutionStrategy {
        force "org.hibernate:hibernate-core:$versions.hibernate"
        force "org.apache.shiro:shiro-all:$versions.shiro"
        force "org.codehaus.groovy:groovy:$versions.groovy"
    }
}

dependencies {
    implementation platform("org.springframework.boot:spring-boot-dependencies:$versions.
    springBoot")

    def excludeSpringBootStarterLogging = { exclude(group: 'org.springframework.boot',
    module: 'spring-boot-starter-logging') }

    // Tapestry
    compile("org.apache.tapestry:tapestry-core:$versions.tapestry")
    compile("org.apache.tapestry:tapestry-hibernate:$versions.tapestry")
    compile("org.apache.tapestry:tapestry-beanvalidator:$versions.tapestry")

    // Compresión automática de javascript y css en el modo producción
    compile("org.apache.tapestry:tapestry-webresources:$versions.tapestry")
    appJavadoc("org.apache.tapestry:tapestry-javadoc:$versions.tapestry")

    // Spring
    compile("org.apache.tapestry:tapestry-spring:$versions.tapestry")
    compile("org.springframework:spring-jdbc:$versions.spring")
    compile("org.springframework:spring-orm:$versions.spring")
    compile("org.springframework:spring-tx:$versions.spring")

  // Spring Boot
    compile("org.springframework.boot:spring-boot-starter",
    excludeSpringBootStarterLogging)
    compile("org.springframework.boot:spring-boot-starter-web",
```

```
     excludeSpringBootStarterLogging)
      compile("org.springframework.boot:spring-boot-starter-actuator",
     excludeSpringBootStarterLogging)
      compile("org.springframework.boot:spring-boot-starter-log4j2")
      compile("org.springframework.boot:spring-boot-autoconfigure")
124   compile("org.springframework.boot:spring-boot-starter-tomcat")

      // Persistencia
      compile("org.jooq:jooq:$versions.jooq")
      compile("commons-dbcp:commons-dbcp:$versions.commons_dbcp")
129   compile("com.h2database:h2:$versions.h2")

      // Seguridad
      compile("org.tynamo:tapestry-security:$versions.tapestrySecurity")

134   // Otras
      compile("javax:javaee-api:$versions.javaee")
      compile("com.google.guava:guava:$versions.guava")

      // Pruebas unitarias
139   testCompile("org.apache.tapestry:tapestry-test:$versions.tapestry") { exclude(group:
     'org.testng'); exclude(group: 'org.seleniumhq.selenium') }
      testCompile("net.sourceforge.tapestrytestify:tapestry-testify:$versions.
     tapestryTestify")
      testCompile("net.sourceforge.tapestryxpath:tapestry-xpath:$versions.tapestryXpath")
      testCompile("junit:junit:$versions.junit")
      testCompile("org.mockito:mockito-core:$versions.mockito")
144   testCompile("org.spockframework:spock-core:$versions.spock")
      testCompile("org.spockframework:spock-spring:$versions.spock")

      // Pruebas de integración
      testCompile("org.gebish:geb-spock:$versions.geb")
149   testCompile("org.gebish:geb-junit4:$versions.geb")
      testCompile("org.seleniumhq.selenium:selenium-support:${versions.selenium}")
      testCompile("org.seleniumhq.selenium:htmlunit-driver:${versions.htmlunitDriver}")
      testCompile("org.springframework.boot:spring-boot-starter-test",
     excludeSpringBootStarterLogging)

154   // Webjars
      runtime("org.webjars:requirejs:$versions.webjars.requirejs")
      runtime("org.webjars:jquery:$versions.webjars.jquery")
      runtime("org.webjars.bower:underscore:$versions.webjars.underscore")
      runtime("org.webjars:bootstrap-select:$versions.webjars.bootstrapSelect")
159
      providedCompile("org.jboss:jboss-vfs:$versions.jbossVfs")

      runtime("org.jboss.logging:jboss-logging:$versions.jbossLogging")
      runtime("org.eclipse:yasson:$versions.yasson")
164   runtime("org.glassfish:javax.json:$versions.json")
```

```
    runtime("com.fasterxml.jackson.core:jackson-databind:$versions.jackson")
    runtime("com.fasterxml.jackson.dataformat:jackson-dataformat-yaml:$versions.jackson")

    jooqRuntime("com.h2database:h2:$versions.h2")
    liquibaseRuntime("org.liquibase:liquibase-core:$versions.liquibase")
    liquibaseRuntime("org.liquibase:liquibase-groovy-dsl:$versions.liquibaseGroovyDsl")
    liquibaseRuntime("com.h2database:h2:$versions.h2")
}

pmd {
  ruleSets = ["java-basic", "java-braces"]
  ruleSetFiles = files("misc/ruleset.xml")
}

test {
    // Excluir de los teses unitarios los teses de integración
    exclude '**/geb/*Spec.*'
}

task integrationTest(type: Test) {
    group = 'Verification'
    description = 'Runs the integration/functional tests.'
    systemProperty 'geb.driver', 'htmlunit'

    // Incluir los teses de integración
    include '**/geb/*Spec.*'
}

task appJavadoc(type: Javadoc) {
    classpath = sourceSets.main.compileClasspath
    source = sourceSets.main.allJava
    destinationDir = reporting.file('appJavadoc')
    options.tagletPath = configurations.appJavadoc.files.asType(List)
    options.taglets = ['org.apache.tapestry5.javadoc.TapestryDocTaglet']

    doLast {
        copy {
            from sourceSets.main.java.srcDirs
            into appJavadoc.destinationDir
            exclude '**/*.java'
            exclude '**/*.xdoc'
            exclude '**/package.html'
        }

        copy {
            from file('src/javadoc/images')
            into appJavadoc.destinationDir
        }
    }
```

```
}

jooq {
    version = '3.11.2'
    edition = 'OSS'
    h2(sourceSets.main) {
        jdbc {
            driver = 'org.h2.Driver'
            url = 'jdbc:h2:./misc/database/app'
            user = 'sa'
            password = 'sa'
        }
        generator {
            name = 'org.jooq.codegen.DefaultGenerator'
            database {
                name = 'org.jooq.meta.h2.H2Database'
                inputSchema = 'PLUGINTAPESTRY'
                includes = '.*'
                excludes = ''
                forcedTypes {
                    forcedType {
                        types = 'TIMESTAMP'
                        userType = 'java.time.LocalDateTime'
                        converter = 'io.github.picodotdev.plugintapestry.misc.
    TimestampConverter'
                    }
                }
            }
            generate {
                interfaces = true
                pojos = true
                relations = true
                validationAnnotations = true
            }
            target {
                packageName = 'io.github.picodotdev.plugintapestry.entities.jooq'
                directory = 'src/main/java'
            }
        }
    }
}

liquibase {
    activities {
        main {
            changeLogFile 'misc/database/changelog.xml'
            url 'jdbc:h2:./misc/database/app'
            username 'sa'
            password 'sa'
```

```
        }
    }
    runList = 'main'
}
```

Con el comando ./gradlew run se iniciará la aplicación siendo accesible en la URL http://localhost:8080/.

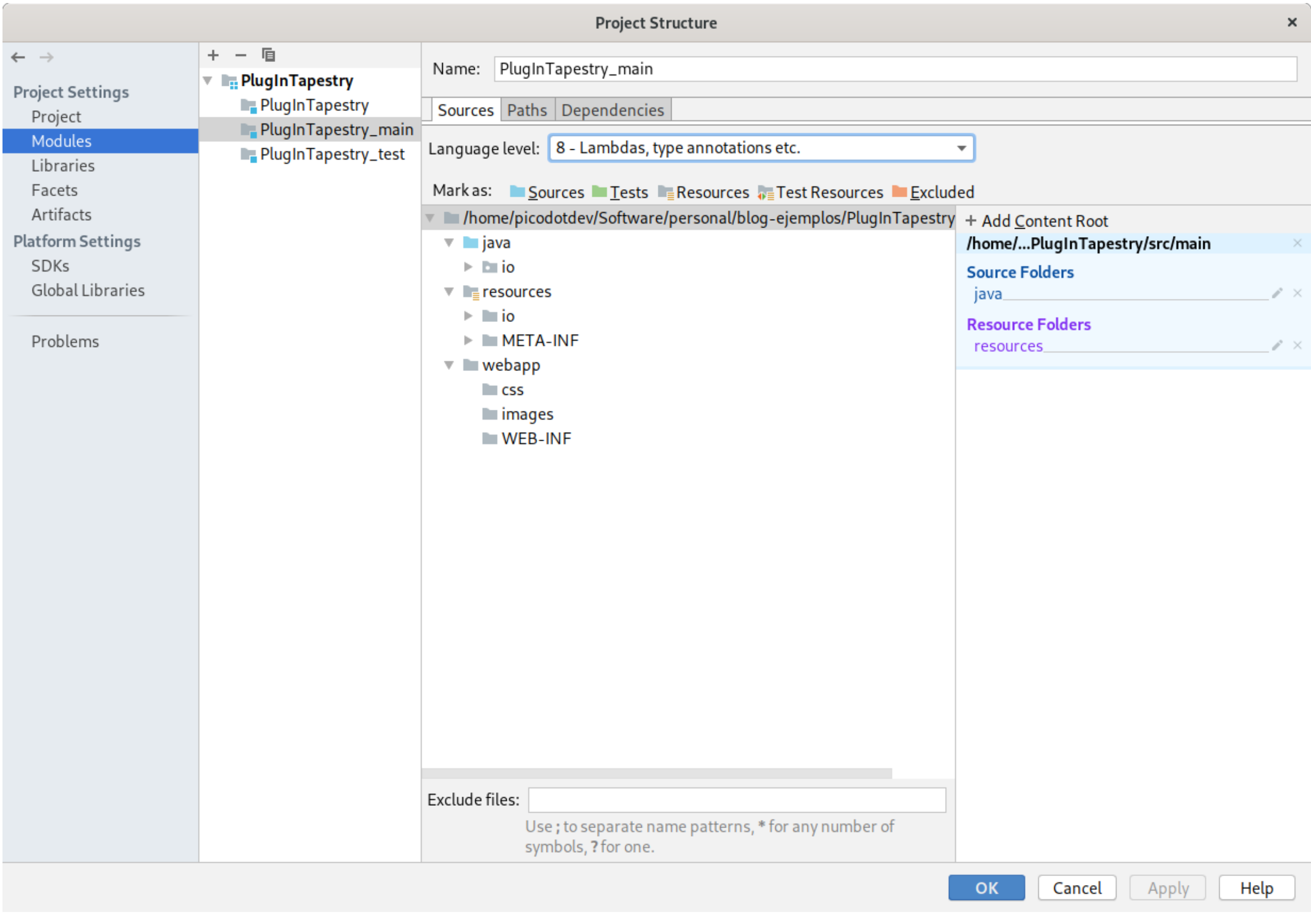
```
$ ./gradlew run
```

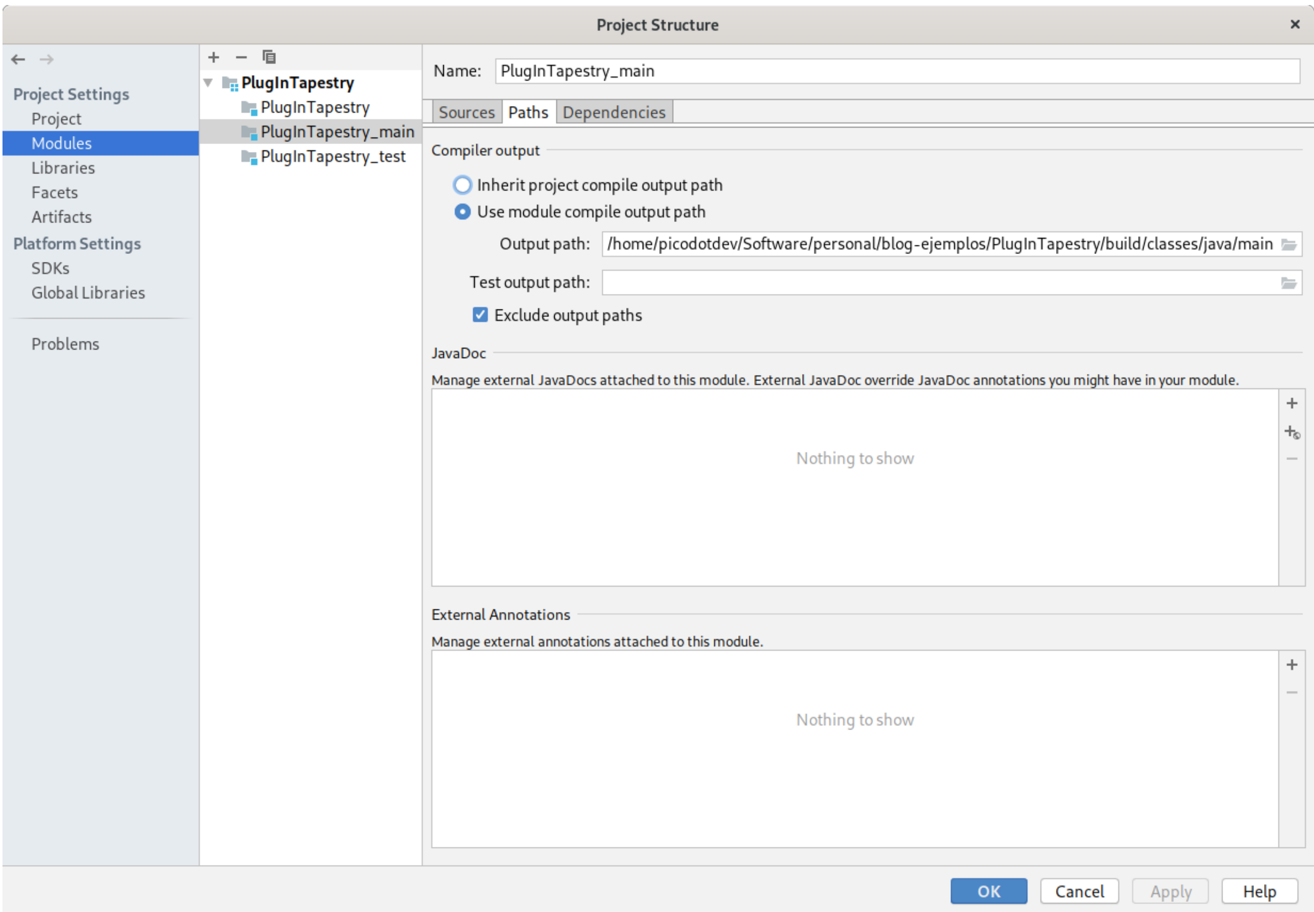
Modificando algunas dependencias podemos cambiar de usar Tomcat a usar Jetty o Undertow.

Listado 2.3: build.gradle

```
// Jetty
configurations {
    compile.exclude module: "spring-boot-starter-tomcat"
}

dependencies {
    ...

    compile("org.springframework.boot:spring-boot-starter-web:$versions.spring_boot")
    compile("org.springframework.boot:spring-boot-starter-jetty:$versions.spring_boot")

    ...
}

// Undertow
configurations {
    compile.exclude module: "spring-boot-starter-tomcat"
}

dependencies {
    ...

    compile("org.springframework.boot:spring-boot-starter-web:$versions.spring_boot")
    compile("org.springframework.boot:spring-boot-starter-undertow:$versions.spring_boot"
    )

    ...
}
```

Si queremos aprovecharnos de la recarga en caliente de clases deberemos configurar eclipse o IntelliJ para que genere las clases en las carpetas build/clases/main y los recursos los situe en build/resources/main. Deberemos tener un directorio de salida diferente por cada capeta de código fuente.


Project Structure
Project Settings
Project
Modules
Libraries
Facets
Artifacts
Platform Settings
SDKs
Global Libraries
Problems
PlugInTapestry
PlugInTapestry
PlugInTapestry_main
PlugInTapestry_test
Name: PlugInTapestry_main
Sources Paths Dependencies
Language level: 8 - Lambdas, type annotations etc.
Mark as: Sources Tests Resources Test Resources Excluded
/home/picodotdev/Software/personal/blog-ejemplos/PlugInTapestry
java
io
resources
io
META-INF
webapp
css
images
WEB-INF
+ Add Content Root
/home/...PlugInTapestry/src/main
Source Folders
java
Resource Folders
resources
Exclude files:
Use ; to separate name patterns, * for any number of symbols, ? for one.
OK Cancel Apply Help



2.4.2 Spring Boot generando un jar

Con el plugin de Spring Boot para Gradle dispondremos algunas tareas adicionales en el proyecto como bootRun para ejecutar la aplicación desde Gradle (similar a la opción run y el parámetro mainClassName que añade el plugin application) y bootJar para poder ejecutar la aplicación con el comando java -jar una vez generado el archivo jar en build/build/libs/PlugInTapestry-1.8.jar.

```
1 $ ./gradlew bootJar
  $ java -jar build/libs/PlugInTapestry-1.8.jar
```

2.4.3 Spring Boot generando un war

Puede que sigamos prefiriendo desplegar la aplicación de forma tradicional en el servidor de aplicaciones usando un archivo war. O puede que queramos usar otro servidor de aplicaciones para los que Spring Boot proporciona un servidor embebido. Hay que indicar como providedRuntime la dependencia del servidor embebido para Spring Boot e incluir el plugin war para poder generar el archivo war con Gradle, adicionalmente la clase Main de la aplicación debe extender de SpringBootServletInitializer.

Listado 2.4: build.gradle

```
...
dependencies {
    ...
    providedRuntime("org.springframework.boot:spring-boot-starter-tomcat:$versions.
    spring_boot") { exclude(group: 'ch.qos.logback') }
    ...
}
...
```

```
$ ./gradlew build
```

Listado 2.5: Main.java

```
package io.github.picodotdev.plugintapestry;

...

@SpringBootApplication
public class Main extends SpringBootServletInitializer implements CommandLineRunner {

    ...

    @Override
    public void run(String... args) {
  StringBuilder banner = new StringBuilder();
  banner.append("   ___  __          ____   ______                 __          \n");
  banner.append("  / _ \\/ /_ _____ _/  _/__/_  __/__ ____  ___ ___ / /_______ __\n");
  banner.append(" / ___/ / // / _ `// // _ \\/ / / _ `/ _ \\/ -_|_-</ __/ __/ // /\n");
  banner.append("/_/  /_/\\_,_/\\_, /___/_//_/_/  \\_,_/ .__/\\__/___/\\__/_/  \\_, / \n"
    );
  banner.append("            /___/                /_/                  /___/");
  logger.info("\n" + banner.toString());
  logger.info("Application running");
    }

    @Override
    protected SpringApplicationBuilder configure(SpringApplicationBuilder application) {
        return application.sources(AppConfiguration.class);
    }

    public static void main(String[] args) throws Exception {
        SpringApplication application = new SpringApplication(Main.class);
        application.setApplicationContextClass(AnnotationConfigWebApplicationContext.
    class);
        SpringApplication.run(Main.class, args);
    }
}
```

Finalmente, la dependencia jboss-logging Spring Boot la coloca en el directorio WEB-INF/lib-provided/ y en mis pruebas la he tenido que mover al directorio WEB-INF/lib/ del archivo war para que Tomcat no produzca la siguiente excepción y el funcionamiento sea el correcto.

```
Caused by: java.lang.NoClassDefFoundError: Could not initialize class org.hibernate.
    validator.internal.engine.ConfigurationImpl
  at org.hibernate.validator.HibernateValidator.createGenericConfiguration(
    HibernateValidator.java:31
)
  at javax.validation.Validation$GenericBootstrapImpl.configure(Validation.java:276)
```

2.4.4 Servidor de aplicaciones externo

Si tenemos un servidor de aplicaciones externo probablemente la mejor opción sea crear varios enlaces simbólicos en el directorio de despliegue del servidor que apunten a las carpetas del código fuente de la aplicación. Por ejemplo si quisiésemos usar un JBoss externo podríamos crear los siguientes enlaces:

- PlugInTapestry -> src/main/webapp
- PlugInTapestry/classes -> build/external/classes
- PlugInTapestry/lib -> build/external/libs

Básicamente lo que hacemos con estos enlaces simbólicos es construir la estructura de un archivo war sin comprimir. En el caso de tomcat deberemos hacer uso del atributo allowLinking del descriptor de contexto.

Listado 2.6: PlugInTapestry.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Context path="/PlugInTapestry" allowLinking="true">
</Context>
```

Para el contenido de la carpeta build/external/libs con las librerías que use la aplicación en tiempo de ejecución podemos definir la siguiente tarea en el archivo build.gradle y acordarnos de ejecutarla después de hacer una limpieza con la tarea clean.

Listado 2.7: build.gradle

```
task copyToLib(type: Copy) {
  into "build/external/libs"
  from configurations.runtime
}
```

Sin embargo, aún tendremos que resolver un problema que es como sincronizar el contenido de la carpeta del enlace simbólico PlugInTapestry/classes que apunta hacia build/external/classes. Necesitamos que su contenido sea el de la carpeta build/classes/main y build/resources/main que es donde hemos configurado el IDE para

que genere los archivos compilados. Además, necesitamos que esta sincronización se haga de forma constante para que los cambios que hagamos sean aplicados inmediatamente gracias a la recarga en caliente de las clases, para ello podemos hacer uso de la herramienta rsync y watch con el siguiente comando:

```
1 $ watch -n 5 rsync -ar --delete build/classes/main/ build/resources/main/ build/external/
    classes/
```

Rsync mantiene sincronizado el contenido de las dos carpetas y watch ejecuta el comando rsync en este caso cada segundo. Si fuésemos a trabajar solo con un servidor de aplicaciones externo podríamos hacer que el IDE generase directamente el contenido en build/external/classes/ tanto para las clases Java como para los recursos y el watch + rsync sería innecesario. Para el caso de querer desarrollar con un servidor externo debemos hacer unas pocas cosas pero para las que disponemos de varias opciones y que nos sirven para cualquier servidor.

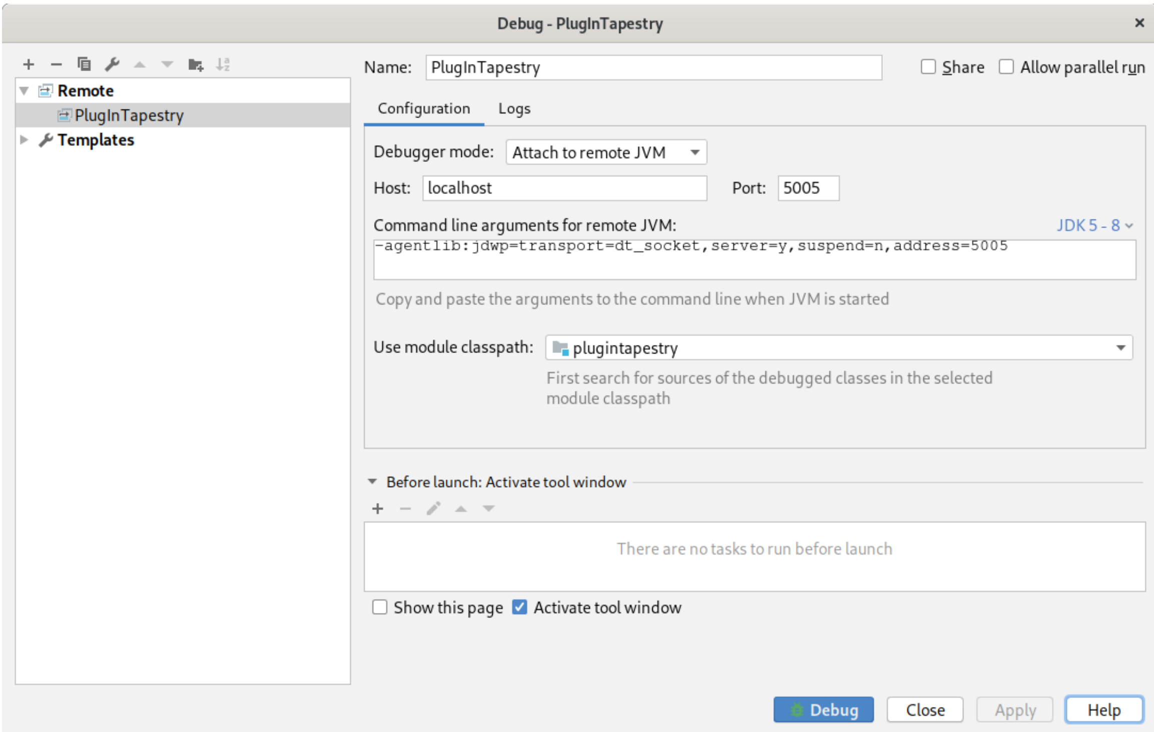
2.5 Debugging

La última pieza que comentaré para tener un entono de desarrollo es como hacer depurado para aquellos casos más complicados. Esto nos puede resultar muy útil para tratar de descubrir la causa del error cuando las trazas, la excepción o el informe de error por si mismo no nos de suficiente información de lo que esta ocurriendo.

Para hacer debug de la aplicación que arrancamos con Gradle mientras estamos desarrollando debemos indicar el parámetro –debug-jvm y Gradle esperará hasta que desde el IDE iniciemos la depuración.

```
1 $ ./gradlew run --debug-jvm
```

Por defecto se utiliza el puerto 5005 pero podemos utilizar cualquiera que esté libre. Para hacer debug desde nuestro IDE también deberemos configurarlo, en el caso de eclipse con la siguiente configuración de aplicación remota.



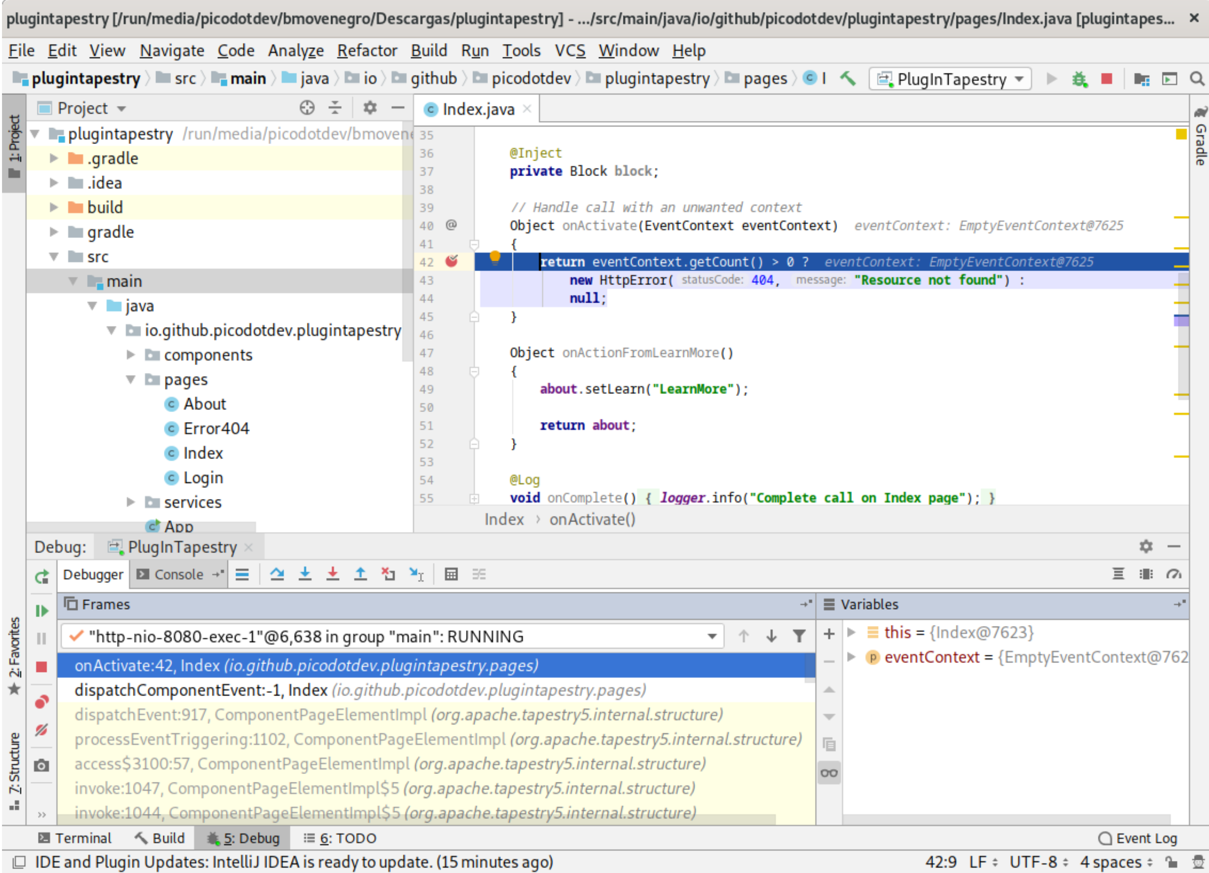
Para un tomcat externo deberemos arrancarlo con la posibilidad de hacer debugging, si normalmente los arrancamos con el comando startup.sh para arrancarlo en modo debug deberemos usar:

```
1 ./catalina.sh jpda start
```

Para jboss deberemos modificar el archivo standalone.conf y descomentar la linea:

```
1 JAVA_OPTS="$JAVA_OPTS -Xrunjdwp:transport=dt_socket,address=8787,server=y,suspend=n"
```

Una vez realizado esto nos podremos conectar a la máquina virtual de tomcat en el puerto 8000 y en el caso de jboss en el 8787 para hacer debug.



2.6 Código fuente de los ejemplos

Este libro viene acompañado de una aplicación en la que podrás ver el código fuente completo de los ejemplos tratados en los diferentes capítulos que por brevedad en el libro solo he incluido los extractos relevantes. Para probarlos solo necesitarás obtener el código fuente y lanzar un comando desde la terminal. En la sección Más documentación tienes más detalles de como obtenerlos, probarlos e incluso si quieres hacer alguna modificación en tu equipo.

Capítulo 3

Páginas y componentes

Tapestry se define como un framework basado en componentes. Esto es así porque las aplicaciones se basan en la utilización de piezas individuales y autónomas que agregadas proporcionan la funcionalidad de la aplicación. Se trabaja en términos de objetos, métodos y propiedades en vez de URL y parámetros de la petición, esto es, en vez de trabajar directamente con la API de los Servlets (o parecida) como requests, responses, sessions, attributes, parameters y urls, se centra en trabajar con objetos, métodos en esos objetos y JavaBeans. Las acciones del usuario como hacer clic en enlaces y formularios provocan cambios en propiedades de objetos combinado con la invocación de métodos definidos por el usuario que contienen la lógica de la aplicación. Tapestry se encarga de la fontanería necesaria para conectar esas acciones del usuario con los objetos.

Los componentes es una aproximación distinta a los frameworks basados en acciones (como Struts, Grails, Symfony, ...). En estos creas acciones que son invocadas cuando el usuario hace clic en un enlace o envía un formulario, eres responsable de seleccionar la URL apropiada y el nombre y tipo de cualquier parámetro que debas pasar. También eres responsable de conectar las páginas de salida (jsp, gsp, php, ...) con esas operaciones.

Esta aproximación orientada a componentes usando un modelo de objetos similar a las interfaces de usuario tradicionales proporciona los siguiente beneficios:

- Alta reutilización de código, dentro y entre proyectos. Cada componente es una pieza reusable y autónoma de código.
- Libera a los desarrolladores de escribir código aburrido y propenso a errores. Codifica en términos de objetos, métodos y propiedades no URL y parámetros de la petición. Se encarga de mucho del trabajo mundano y propenso a errores como los enlaces de las peticiones, construir e interpretar las URL codificadas con información.
- Permite a las aplicaciones escalar en complejidad. El framework realiza la construcción de los enlaces y el envío de eventos transparentemente y se pueden construir componentes más complejos a partir de componentes más simples.
- Fácil internacionalización y localización. El framework selecciona la versión localizada no solo de los textos sino también de las plantillas e imágenes.

- Permite desarrollar aplicaciones robustas y con menos errores. Usando el lenguaje Java se evitan muchos errores de compilación en tiempo de ejecución y mediante el informe de error avanzado con precisión de linea los errores son más fácilmente y rápidamente solucionados.
- Fácil integración entre equipos. Los diseñadores gráficos y desarrolladores puede trabajar juntos minimizando el conocimiento de la otra parte gracias a la instrumentación invisible.

Las aplicaciones se dividen en un conjunto de páginas donde cada página está compuesta de componentes. Los componentes a su vez pueden estar compuestos de otros componentes, no hay límites de profundidad. En realidad las páginas son componentes con algunas responsabilidades adicionales. Todos los componentes pueden ser contenedores de otros componentes. Las páginas y la mayoría de los componentes definidos por el usuario tienen una plantilla, un archivo cuyo contenido es parecido a html que define las partes estáticas y dinámicas, con marcadores para para los componentes embebidos. Muchos componentes proporcionados por el framework no tienen una plantilla y generan su respuesta en código Java.

Los componentes pueden tener parámetros con nombre que puede ser establecidos por el componente o página que los contiene. Al contrario que los parámetros en Java los parámetros en Tapestry pueden ser bidireccionales, un componente puede leer un parámetro para obtener su valor o escribir en el parámetro para establecerlo. La mayoría de los componentes generan código html, un subconjunto se encarga de generar enlaces y otros se encargan de los elementos de formularios.

Por otro lado también están los mixins que sirven para añadir su funcionalidad al componente junto con el que se usa, es decir, no son usados individualmente sino que son son aplicados al uso de algún otro componente. Pueden añadir funcionalidades como autocompletado a un input, hacer que una vez pulsado un botón o al enviar su formulario este se deshabilite (lo que nos puede evitar el problema del Doble envío (o N-envío) de formularios).

Las páginas en mayor parte tienen las mismas propiedades que los componentes pero con unas pocas diferencias:

- No se puede usar una página dentro de otra página, mientras que los componentes pueden ser usados dentro de otros componentes.
- Las páginas tienen URLs, los componentes no.
- Los componentes tienen parámetros, las páginas no.
- Las páginas tienen un contexto de activación, los componentes no.

3.1 Clase del componente

La clase del componente es el código Java asociado a la página, componente o mixin de la aplicación web. Las clases para las páginas, componentes y mixins se crean de idéntica forma. Son simplemente POJO con anotaciones y unas convenciones para los nombres de los métodos. No son abstractas ni necesitan extender una clase base o implementar una determinada interfaz.

Figura 3.1: Modelo Vista Controlador de Tapestry

En la mayoría de casos, cada componente tendrá una plantilla. Sin embargo, es posible que un componente emita sus etiquetas sin necesidad de una plantilla. La clase Java es el único archivo imprescindible de código fuente que tendremos que escribir para hacer un nuevo componente el resto son opcionales.

Tapestry sigue el patrón MVC pero su aproximación es diferente a lo que podemos encontrar en muchos de los frameworks basados en acciones. La clase del componente (controlador) y su plantilla (vista), en caso de tenerla, siguen siendo dos archivos diferentes pero íntimamente relacionados, la plantilla puede acceder a las propiedades e invocar los métodos que necesite de su clase controlador que a su vez posiblemente accedan a una base de datos para obtener los datos (modelo). Esto hace que las plantillas tengan muy poca lógica y que esta se ubique en su clase asociada, usando el lenguaje Java el compilador nos avisará de errores de compilación y podremos aprovecharnos en mayor medida de las utilidades de refactorización de los IDE. Cada plantilla tiene su clase de componente asociada y esta se conoce de forma inequívoca ya que la plantilla y la clase Java tienen el mismo nombre y se ubican en el mismo paquete. Todo ello hace que no tengamos la responsabilidad de unir el controlador (la clase java) con la vista (la plantilla) y su modelo de datos (obtenido a través de su controlador) que en el momento de hacer modificaciones y refactor puede suponer una dificultad en otros frameworks.

Creando un componente trivial

Crear un componente en Tapestry es muy sencillo. Estas son las restricciones:

- Tiene que ser una clase pública.
- La clase tiene que estar en el paquete correcto.
- Y la clase tienen que tener un constructor público sin argumentos (el constructor que proporciona el compilador es suficiente, no deberíamos necesitar un constructor con parámetros).

Este es un componente mínimo que emite un mensaje usando una plantilla:

Listado 3.1: HolaMundoTemplate.java

```
1 package io.github.picodotdev.plugintapestry.components;
```

```
public class HolaMundoTemplate {
}
```

Y una plantilla tml asociada:

Listado 3.2: HolaMundoTemplate.tml

```
<!DOCTYPE html>
<t:container xmlns="http://www.w3.org/1999/xhtml" xmlns:t="http://tapestry.apache.org/
    schema/tapestry_5_4.xsd" xmlns:p="tapestry:parameter">
    ¡Hola mundo! (template)
</t:container>
```

A continuación los mismo pero sin la necesidad de una plantilla:

Listado 3.3: HolaMundo.java

```
package io.github.picodotdev.plugintapestry.components;

import org.apache.tapestry5.MarkupWriter;

public class HolaMundo {

  void beginRender(MarkupWriter writer) {
    writer.write("¡Hola mundo! (java)");
  }
}
```

En este ejemplo, al igual que el primero, la única misión del componente es emitir un mensaje fijo. El método beginRender de la fase de renderizado sigue una convención en su nombre y es invocado en un determinado momento por Tapestry. Estos métodos no es necesario que sean públicos, pueden tener cualquier nivel de accesibilidad que queramos, por convención suelen tener nivel de paquete.

Paquetes del componente

Las clases de los componentes han de colocarse en el paquete apropiado, esto es necesario para que se les añada el código en tiempo de ejecución y la recarga de las clases funcione. Estos son los paquetes que han de existir en el paquete raíz de la aplicación:

- Para las páginas: el paquete donde se han de colocar es [root].pages. Los nombres de las páginas se corresponden con el de las clases de este paquete.
- Para los componentes: el paquete donde se han de colocar es [root].components. Los tipos de los componentes se corresponden con el de las clases de este paquete.
- Para los mixins: el paquete donde se han de colocar es [root].mixins. Los tipos de los mixins se corresponden con el de las clases de este paquete.

En caso de que tengamos una clase base de la que heredan las páginas, componentes o mixins estas no han de colocarse en los anteriores paquetes ya que no son válidas para realizar las transformaciones. Se suelen colocar en el paquete [root].base.

Subpaquetes

Las clases no tienen que ir directamente dentro de su paquete, es válido crear subpaquetes. El nombre del subpaquete se convierte parte del nombre de la página o del tipo del componente. De esta forma, puedes crear una página en el paquete io.github.picodotdev.plugintapestry.pages.admin.ProductoAdmin y el nombre lógico de la página será admin/Producto.

Tapestry realiza algunas optimizaciones al nombre lógico de la página, componente o mixin, comprueba si el nombre del paquete es un prefijo o sufijo de nombre de la clase y lo elimina del nombre lógico si es así. El resultado es que si una página tiene el siguiente paquete io.github.picodotdev.plugintapestry.pages.admin.ProductoAdmin tendrá el nombre lógico de admin/Producto y no admin/ProductoAdmin. El objetivo es que las URL sean más cortas y naturales.

Páginas índice

Otra simplificación son las páginas índice, si el nombre lógico de una página es Index después de eliminar el nombre de paquete se corresponderá con la raíz de esa carpeta. Una clase con el nombre io.github.picodotdev.plugintapestry o io.github.picodotdev.plugintapestry.pages.usuario.IndexUsuario tendrá la URL usuario/. Esto también se aplica para la página raíz de la aplicación io.github.picodotdev.plugintapestry.pages.Index que se corresponderá con /.

Páginas contra componentes

La distinción entre páginas y componentes es muy, muy pequeña. La única diferencia real es el nombre del paquete y que conceptualmente las páginas son el contenedor raíz del árbol de componentes.

Transformación de clase

Tapestry usa las clases como punto de partida que transforma en tiempo de ejecución. Estas transformaciones son invisibles. Dado que la transformación no ocurre hasta en tiempo de ejecución, la fase de construcción no se ve afectada por el hecho de que estés creando una aplicación Tapestry. Es más, las clases son absolutamente POJO también durante el tiempo de pruebas unitarias.

Recarga en vivo de clases

Las clases de los componentes son monitorizadas por el framework en busca de cambios y son recargadas cuando se modifican. Esto significa que puedes desarrollar con la velocidad de un entorno de scripting sin sacrificar la potencia de la plataforma Java.

El resultado es un aumento de la productividad. Sin embargo, la recarga de clases solo se aplica para las clases de los componentes y las implementaciones de los servicios. Otras clases como las interfaces de los servicios, las clases de entidad del modelo y otras clases quedan fuera de la recarga en vivo de las clases.

Variables de instancia

Las clases de los componentes pueden tener propiedades y pueden estar en el ámbito protected o private. Con la anotación @Property se añadirán los métodos get y set de esa propiedad en la transformación de la clase.

A menos que las propiedades sean decoradas con una anotación de persistencia se considerarán propiedades transitorias. Esto significa que al final de la petición perderán su valor.

Constructores

Las clases de los componentes se instanciarán usando el constructor sin argumentos por defecto. El resto de constructores serán ignorados.

Inyección

La inyección de dependencias ocurren a nivel de propiedades a través de anotaciones. En tiempo de ejecución las propiedades con anotaciones de inyección se convierten en solo lectura.

```
1  // Inyectar un asset
   @Inject
   @Path("context:images/logo.png")
   private Asset logo;
5
   // Inyectar un servicio
   @Inject
   private ProductoDAO dao;

10 // Inyectar un componente embebido de la plantilla
   @Component
   private Form form;

   // Inyectar un bloque de la plantilla
15 @Inject
```

```
private Block foo;

// Inyectar un recurso
@Inject
private ComponentResources componentResources;
```

Parámetros

Los parámetros del componente son propiedades privadas anotadas con @Parameter. Son bidireccionales lo que significa que su valor además de ser leído puede ser modificado y se crea un enlace entre la propiedad del componente y la propiedad del componente que lo contiene.

Propiedades persistentes

La mayoría de propiedades pierden su valor al finalizar la petición. Sin embargo, pueden anotarse con @Persist para que mantengan su valor entre peticiones.

Componente embebidos

Es común que los componentes contengan otros componentes. A los componentes contenidos se les denomina embebidos. La plantilla del componente contendrá elementos especiales que identificarán en que partes de la página irán.

Los componentes se pueden definir dentro de la plantilla o mediante propiedades de instancia mediante la anotación @Component que definirá el tipo y los parámetros.

Listado 3.4: Mensaje.java

```java
package io.github.picodotdev.plugintapestry.components;

...

public class Mensaje {
  @Component(parameters = { "value=m" })
  private TextOutput output;

  @Parameter(defaultPrefix = BindingConstants.LITERAL)
  private String m;
}
```

Este componente podría ser usando en una plantilla de la siguiente forma:

```
<t:mensaje m="¡Hola mundo!"/>
```

Anotaciones

Tapestry hace un uso extensivo de anotaciones, esto hace que no sean necesarios archivos de configuración XML, además en algunos casos siguiendo convenciones no hace falta usar anotaciones.

Las anotaciones están agrupadas en función de su finalidad: Para usarse en páginas, componentes y mixins. Para usarse en los servicios y IoC, para Hibernate o JPA, para usarse con los componentes BeanEdit y Grid, etc... Algunas anotaciones destacadas son:

- BeginRender: hace que el método anotado sea llamado cuando el componente comience su renderizado.
- Cached: el resultado de un método es cacheado para que en futuras llamadas no se tenga que volver a calcular.
- Component: permite inyectar un componente embebido en la plantilla.
- OnEvent: el método es el manejador de un evento.
- Parameter: la propiedad es un parámetro del componente.
- Persist: guarda el valor de la propiedad para que esté accesible en futuras peticiones.
- Property: crea los métodos get y set para la propiedad en tiempo de ejecución.
- Service: inyecta un servicio en el componente por nombre de servicio, se usa junto con la anotación Service.
- Inject: inyecta un servicio por nombre de interfaz.
- Symbol: inyecta el valor de un símbolo dado su nombre.
- CommitAfter: al finalizar el método sin una excepción unchecked se hace un commit de la transacción.

3.2 Plantillas

La plantilla de un componente es un archivo que contiene el lenguaje de marcas (html) que generará. Las plantillas son documentos XML bien formados, esto significa que cada etiqueta debe tener su correspondiente de cierre, cada atributo debe estar entrecomillado y el resto de reglas que se aplican a los documentos XML. En tiempo de ejecución se comprueba que el documento esté bien formado sin comrpobar que sea válido aunque incluya DTD o esquemas.

Estas plantillas en su mayor parte son html o xhtml estándar, las extensiones son proporcionadas por Tapestry al lenguaje de marcas con un nuevo espacio de nombres. Una plantilla para un página podría ser:

```
<!DOCTYPE html>
<html t:type="layout" xmlns:t="http://tapestry.apache.org/schema/tapestry_5_4.xsd">
  <h1>Hola desde el componente HolaMundo.</h1>

  <t:mensaje m="¡Hola mundo!"/>
</html>
```

Localización

En este apartado la localización de las plantillas tiene relación con la ubicación y nombre de la clase del componente. Tendrá el mismo nombre del archivo Java asociado pero con la extensión .tml (Tapestry Markup Language) y almacenadas en el mismo paquete, siguiendo la estructura estándar de Gradle los archivos para un componente podría ser:

- Java class: src/main/java/io/github/picodotdev/plugintapestry/components/HolaMundo.java
- Template: src/main/resources/io/github/picodotdev/plugintapestry/components/HolaMundo.tml

De la misma forma para una página sería:

- Java class: src/main/java/io/github/picodotdev/plugintapestry/pages/Index.java
- Template: src/main/resources/io/github/picodotdev/plugintapestry/pages/Index.tml

Doctypes

Como las plantillas son documentos XML bien formados para usar entidades html como &, <, >, © se debe usar un doctype html o xhtml. Este será pasado al cliente en el (x)html resultante. Si una página está compuesta de múltiples componentes, cada uno con una plantilla que posee una declaración doctype se usará el primer doctype encontrado. Los siguientes son los doctypes más comunes:

```
<!DOCTYPE html>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD
    /xhtml1-strict.dtd">

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/
    xhtml1/DTD/xhtml1-transitional.dtd">

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/html4/strict.dtd"
    >

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/
    html4/loose.dtd">
```

El primero es para html5 y es el recomendado y se usará en caso de que una plantilla no lo tenga.

Espacios de nombres

Las plantillas de componentes deben incluir el espacio de nombres de Tapestry en el elemento raíz de la plantilla. En el siguiente ejemplo se usa el prefijo estándar t:

```
1  <!DOCTYPE html>
   <html xmlns:t="http://tapestry.apache.org/schema/tapestry_5_4.xsd" xmlns:l="tapestry-
      library:libreria">
   <head>
     <title>Página</title>
   </head>
6  <body>
     <t:libreria.mensaje texto="¡Hola mundo!"/>
     <span t:type="libreria/mensaje" texto="¡Hola mundo!"/>
     <l:mensaje texto="¡Hola mundo!"/>
   </body>
11 </html>
```

Elementos

En algunos casos un componente es diseñado para que su plantilla se integre alrededor del componente contenido. Los componentes tiene control en que lugar es incluido el cuerpo. Mediante el elemento <t:body/> se identifica en que lugar de la plantilla debe ser incluido lo generado por el componente contenido. El siguiente ejemplo muestra un componente que podría actuar como layout de las páginas de la aplicación:

```
1  <!DOCTYPE html>
   <html xmlns:t="http://tapestry.apache.org/schema/tapestry_5_4.xsd">
   <head>
4    <title>PlugIn Tapestry</title>
   </head>
   <body>
     <t:body/>
   </body>
9  </html>
```

Una página usaría el componente anterior de la siguiente manera:

```
1  <!DOCTYPE html>
   <html t:type="layout" xmlns:t="http://tapestry.apache.org/schema/tapestry_5_4.xsd">
     Contenido específico de la página
   </html>
```

Cuando se genere la página, la plantilla del componente layout y la plantilla de la página son fusionadas generando lo siguiente:

```
1  <!DOCTYPE html>
   <html>
   <head>
     <title>PlugIn Tapestry</title>
   </head>
6  <body>
```

```
  Contenido específico de la página
</body>
</html>
```

Un elemento <t:container> no es considerado parte de la plantilla y es útil para componentes que generan varios elementos de nivel raíz. Por ejemplo, un componente que genera las columnas de una tabla:

```
1 <!DOCTYPE html>
  <t:container xmlns:t="http://tapestry.apache.org/schema/tapestry_5_4.xsd">
    <td>${label}</td>
    <td>${value}</td>
  </t:container>
```

Sin el elemento <t:container> el XML de la plantilla no sería XML válido ya que los documentos XML deben tener siempre un único elemento raíz.

El elemento <t:block> es un contenedor de una porción de la plantilla del componente. Un bloque no se renderiza en la salida por si solo, sin embargo, puede inyectarse y controlar cuando es necesario mostrar su contenido. Un componente puede ser anónimo o tener un id (especificado mediante el atributo id). Solo los bloques con id pueden ser inyectados en la clase Java del componente.

El elemento <t:content> marca la porción de la plantilla como el contenido a usar, cualesquiera marcas fuera de este elemento será ignoradas. Esto es útil para eliminar pociones de la plantilla que solo existen para soportar la previsualización de herramientas WYSIWYG (What You See Is What You Get, lo que ves es lo que obtienes).

El elemento <t:remove> marca porciones de la plantilla que serán eliminadas, es como si lo contenido en ella no fuera parte de la plantilla. Esto es usado para incluir comentarios en el código fuente o para eliminar temporalmente porciones de la plantilla de la salida pero no de los archivos de código fuente.

El elemento <p:parameter> es un bloque especial debiéndose definir su espacio de nombres.

```
1 <html xmlns:t="http://tapestry.apache.org/schema/tapestry_5_4.xsd" xmlns:p="tapestry:
      parameter">
```

Definido el espacio de nombres se puede pasar un bloque usando el espacio de nombre p: y un elemento que corresponda al nombre del parámetro.

```
1 <t:if test="loggedIn">
    Hola, ${usuario}!
    <p:else>
4     Haga clic para <a t:type="actionlink" t:id="login">iniciar sesión</a>.
    </p:else>
  </t:if>
```

Este ejemplo pasa un bloque de la plantilla (conteniendo un componente ActionLink y algo de texto) al componente If como un parámetro de nombre else. En la página de documentación del componente If verás que else es un parámetro de tipo Block.

Los elementos del espacio de nombres parámetro no está permitido que tengan atributos. El nombre del elemento indica el nombre del componente al que asiciarse.

Expansiones

Las expansiones son unas cadenas especiales en el cuerpo de las plantillas y tienen una sintaxis similar a las expresiones de Ant.

```
Bienvenido, ¡${usuario}!

${message:Hola_mundo}
```

Aquí ${usuario} es la expansión y ${message:Hola_mundo} otra usando un binding. En este ejemplo el valor de la propiedad usuario del componente es extraído convertido a String y enviado como resultado de la salida. Las expansiones está permitidas dentro de texto y dentro de elementos ordinarios y de elementos de componentes. Por ejemplo:

```
<img src="${request.contextPath}/images/catalog/product_${id}.png"/>
```

En este hipotético ejemplo, la clase del componente proporciona la propiedad request e id, y ambos son usados para construir el atributo src de la etiqueta <img>. Internamente las expansiones son lo mismo que los bindings de parámetros. El binding por defecto de una expansión es prop: (esto es, el nombre de una propiedad del componente) pero se pueden utilizar otros bindings. Especialmente útil es el binding message: para acceder a los mensajes localizados del catálogo de mensajes del componente. No uses expansiones en los parámetros de componentes si el binding por defecto del parámetro es prop: o var: ya que las expansiones convierten el valor a una cadena inmutable que produce una excepción en tiempo de ejecución si el componente trata de actualizar el valor del parámetro. Incluso para parámetros de solo lectura las expansiones no son deseables ya que siempre convierten a un String y a partir de él al tipo declarado por el parámetro. En los parámetros de los componentes es mejor usar la expresión de un binding literal:, prop:, message:, ... pero no ${...}.

Ten en cuenta que las expansiones escapan cualquier caracter reservado de HTML. Específicamente, el menor que (<), el mayor que (>) y el ampersand (&) sustituyéndose por sus entidades respectivamente <, > y &. Esto es lo que normalmente se quiere, sin embargo, si la propiedad contiene HTML que quieres emitir como contenido de marcado puede usar el componente OutpuRaw de la siguiente forma donde contenido es una propiedad de componente que almacena HTML:

```
<t:outputRaw value="contenido"/>
```

Hay que tener cuidado con esto si el contenido proviene de una fuente no confiable, asegurate de filtrarlo antes de usarlo en el componente OutputRaw, de otra manera tendrás una poetencial vulnerabilidad de cross-site scripting.

Componentes embebidos

Un componente embebido se identifica en la plantilla por el espacio de nombre t: del elemento.

```
Tienes ${carrito.size()} elementos en el carrito.
<t:actionlink t:id="clear">Vaciar</t:actionlink>.
```

El nombre del elemento, t:actionlink, es usado para identificar el tipo del componente, ActionLink. El nombre del elemento es insensible a mayúsculas y minúsculas, podría ser también t:actionlink o t:ActionLink. Los componentes puede tener dos parámetros específicos de Tapestry:

- id: Un identificativo único para el componente dentro de su contenedor.
- mixins: Una lista de mixins separada por comas para el componente.

Estos atributos son especificados dentro del espacio de nombre t:, por ejemplo t:id="clear". Si el atributo id es omitido se le asignará automáticamente uno único. Para los componentes no triviales se debería asignar uno id siempre, ya que las URL serán más cortas y legibles y el código será más fácil de depurar ya que será más obvio como las URL se corresponden con las páginas y componentes. Esto es muy recomendable para los controles de formulario. Los ids debe ser identificadores Java válidos (empezar con una letra y contener solo letras, números y barras bajas).

Cualquier otro atributo es usado como parámetros del componente. Estos pueden ser parámetros formales o informales. Los parámetros formales son aquellos que el componente declara que puede recibir, tienen un binding por defecto de prop:. Los parámetros informales son usados para incluirse tal cual como atributos de una etiqueta del componente, tienen un binding por defecto de literal:.

La apertura y cierre de las etiquetas de un componente definen el cuerpo del componente. Es habitual que componentes adicionales sean incluidos en el cuerpo de otro componente:

```
<t:form>
  <t:errors/>

  <div class="control-group">
    <t:label for="nombre" />
    <div class="controls">
      <input t:type="textfield" t:id="nombre" value="producto.nombre" size="100" label="
    Nombre" />
    </div>
  </div>
</t:form>
```

En este ejemplo, el elemento <t:form> en su cuerpo contiene otros componentes. Todos estos componentes (form, errors, label, ...) son hijos de la página. Algunos componentes requieren estar contenidos en un determinado componente, como por ejemplo todos los campos de formulario (como TextField) que deben estar dentro de un componente form y lanzarán una excepción si no lo están.

Es posible colocar los componentes en subpaquetes. Por ejemplo, si tu aplicación tiene un paquete como io.github.picodotdev su nombre será ajax/Dialog. Para que este nombre pueda ser incluido en la plantilla XML se han de usar puntos en vez de barras, en este caso se debería usar <t:ajax.dialog>.

Espacio de nombre de una librería

Si usas muchos componentes de una librería puedes usar un espacio de nombre para simplificar las referencias a esos componentes. En el siguiente ejemplo se usa un componente de tres formas distintas pero equivalentes:

```
1  <!DOCTYPE html>
   <html xmlns:t="http://tapestry.apache.org/schema/tapestry_5_4.xsd" xmlns:l="tapestry-
       library:libreria">
   <head>
     <title>Página</title>
5  </head>
   <body>
     <t:libreria.mensaje texto="¡Hola mundo!"/>
     <span t:type="libreria/mensaje" texto="¡Hola mundo!"/>
     <l:mensaje texto="¡Hola mundo!"/>
10 </body>
   </html>
```

Instrumentación invisible

La instrumentación invisible es un buena característica para que el equipo de diseñadores y de desarrolladores puedan trabajar a la vez. Esta características permite a los desarrolladores marcar elementos html ordinarios como componentes que los diseñadores simplemente pueden ignorar. Esto hace que las plantillas sean también más concisas y legibles. La instrumentación invisible necesita usar el atributo id o type con el espacio de nombres t:. Por ejemplo:

```
1  <!DOCTYPE html>
   <html xmlns:t="http://tapestry.apache.org/schema/tapestry_5_.xsd">
     <p>¡Feliz navidad!,
4    <span t:id="count" t:type="Count" end="3">
       Ho!
     </span>
   </p>
```

El atributo t:type marca el elemento span como un componente. Cuando se renderice, el elemento span será reemplazado por la salida del componente Count. Los atributos id, type y mixins pueden ser colocados con el espacio de nombres de Tapestry (casi siempre t:id, t:type y t:mixins). Usar el espacio de nombres de Tapestry para un atributo es útil cuando el elemento a instrumentalizar no lo tiene definido, de esta manera podemos evitar las advertencias que el IDE que usemos puede que proporcione. Un componente invisiblemente instrumentalizado debe tener un atributo type identificado con una de las siguientes dos formas:

- Como t:type visto anteriormente.
- En la clase Java del componente contenedor usando la anotación Component. Donde el valor de type es determinado por el tipo de la propiedad o el atributo type de la anotación.

```
@Component
private Count count;
```

Se puede elegir cualquiera de las dos formas de instrumentación. Sin embargo, en algunos casos el comportamiento del componente es influenciado por la decisión. Por ejemplo, cuando la plantilla incluye el componente Loop usando la instrumentación invisible, el tag original (y sus parámetros informales) se renderizará repetidamente alrededor del cuerpo del mensaje. Así por ejemplo si tenemos:

```
<table>
  <tr t:type="loop" source="elementos" value="elemento" class="prop:fila">
    <td>${elemento.id}</td>
    <td>${elemento.nombre}</td>
    <td>${elemento.cantidad}</td>
  </tr>
</tabel>
```

El componente Loop se fusiona en el elemento <tr> renderizando un elemento <tr> por cada elemento en la lista de elementos, cada elemento <tr> incluirá tres elementos <td>. También escribirá el atributo informal class en cada <tr>.

Espacio de nombre de parámetros

Son una forma de pasar como parámetros bloques de componentes. Puedes definir un espacio de nombres especial p:. Con el espacio de nombres tapestry:parameter se puede pasar un bloque usando el prefijo p: con el nombre del elemento coincidiendo con el nombre del parámetro:

```
<!DOCTYPE html>
<html xmlns:t="http://tapestry.apache.org/schema/tapestry_5_4.xsd" xmlns:p="tapestry:
    parameter">
  <t:if test="sesiónIniciada">
    Hola, ¡${usuario}!
    <p:else>
      <a t:type="actionlink" t:id="login">Haz clic aquí para iniciar sesión</a>.
    </p:else>
  </t:if>
</html>
```

Este ejemplo pasa un bloque de la plantilla (conteniendo el componente ActionLink y algo de texto) al componente If en un parámetro de nombre else. En la documentación de referencia encontrarás que el parámetro else es de tipo Block. Los elementos en el espacio de nombres p: no está permitido que tengan atributos, el nombre del elemento es usado para identificar el parámetro del componente.

Espacios en las plantillas

Tapestry elimina los espacios en blanco innecesarios de las plantillas. Dentro de un bloque de texto, los espacios en blanco repetidos son reducidos a un único espacio. Los bloques con solo espacios son eliminados por completo. Si ves el código fuente de la salida (en el modo producción) verás que la página por completo es una única linea sin apenas retornos de carro.

Esto tiene ciertas ventajas de eficiencia tanto en el servidor (al procesar menos datos) como en el cliente (menos caracteres que parsear). Herramientas como FireBug y Chrome Developer Tool son útiles para ver la salida en el cliente de forma más legible.

En raras ocasiones que los espacios en la plantilla son significativos son conservados. Al generar un elemento <pre> con texto preformateado o al interaccionar con la hojas de estilos para conseguir un efecto. Para ello se puede usar el atributo estándar xml:space para indicar que los espacios deberían ser comprimidos (xml:space="default") o preservados (xml:space="preserve"). Estos atributos son eliminados de la plantilla y no generados en la salida. Por ejemplo:

```
1 <ul class="menu" xml:space="preserve">
    <li t:type="loop" t:source="paginas" t:value="var:pagina">
      <t:pagelink page="var:pagina">${var:pagina}</t:pagelink>
    </li>
  </ul>
```

Esto preservará los espacios entre los elementos <ul> y <li> y entre los elementos <li> y los elementos <a> anidados. La salida será la siguiente:

```
1 <ul>
    <li><a href="carrito>Mostrar carrito</a></li>
    <li><a href="cuenta">Ver cuenta</a></li>
  </ul>
```

Con la compresión normal, la salida sería la siguiente:

```
1 <ul><li><a href="carrito">Mostrar carrito</a></li><li><a href="cuenta">Ver cuenta</li></
      ul>
```

Puedes incluso colocar los atributos xml:space dentro de los elementos anidados para controlar detalladamente que es preservado y que es comprimido.

Herencia de plantillas

Si un componente no tiene plantilla pero extiende de una clase de un componente que tiene una plantilla, entonces la plantilla de la clase padre será usada por el componente hijo. Esto permite extender la clase base sin duplicar la plantilla. La plantilla padre puede marcar secciones reemplazables con <:extension-points> y los subcomponentes reemplazar esas secciones con <t:remplace>. Esto funciona a varios niveles de herencia aunque no es recomendable abusar de esta característica. En general es mejor la composición que la herencia ya que será más fácil de entender y mantener.

<t:extension-point>

Marca el punto de la plantilla que puede ser reemplazado. Un id único (insensible a mayúsculas) es usado en la plantilla y subplantillas para unir los puntos de extensión con los posibles reemplazos.

```
1 <t:extension-point id="titulo">
    <h1>${tituloPorDefecto}</h1>
  </t:extension-point>
```

<t:extend>

Es el elemento raíz de una plantilla hija que hereda de se plantilla padre. El atributo <t:extend> solo puede aparecer como raíz y solo puede contener elementos <t:replace>.

<t:replace>

Reemplaza un punto de extensión de la plantilla padre por su contenido. Solo puede aparecer como hijo de un elemento raíz <t:extend>.

```
1 <t:extend xmlns:t="http://tapestry.apache.org/schema/tapestry_5_3.xsd">
2   <t:replace id="titulo">
      <h1><img src="${context:images/icon.jpg}"/>Carrito</h1>
    </t:replace>
  </t:extend>
```

3.2.1 Content Type y markup

Tapestry necesita que las plantillas XML de los componentes estén bien formadas y renderizar su salida como XML, con algunas cosas a tener en cuenta:

- La declaración <?xml?> XML es omitida.
- Los elementos se renderizan con una etiqueta de apertura y de cierre, incluso si están vacías.
- Algunos elementos serán abreviados a únicamente la etiqueta de apertura como <br>, <hr> e <img>.
- Las secciones <![CDATA[...]> serán emitidas tal cual están definidas.

Esto es para asegurar que el lenguaje de marcas, casi bien formado, sea entendido apropiadamente por los navegadores que esperan html ordinario. En realidad, Tapestry puede decidir renderizar un documento XML puro, depende del content type de la respuesta. Cuando se renderiza una página, el content type y el mapa de caracteres es obtenido de los metadatos de la misma página. Los metadatos son especificados usando la anotación @Meta.

Content type

Por defecto tiene el valor «text/html» lo que desencadena una renderización especial de XML. Una página puede declarar su content type usando la anotación de clase @ContentType. Los content types distintos de «text/html» renderizará documentos XML bien formados, incluyendo la declaración XML y un comportamiento más estándar para los elementos vacíos.

Mapa de caracteres

El mapa de caracteres o character map (esto es la codificación de caracteres) usado al escribir en la salida es normalmente UTF-8. UTF-8 es una versión de Unicode donde los caracteres individuales son codificados con uno o más bytes. La mayoría de caracteres de los lenguajes western son codificados en un solo byte. Los caracteres acentuados o no-western (como los japoneses, árabes, etc.) pueden ser codificados con dos o más bytes. Puede especificarse que todas las páginas usen la misma codificación mediante el símbolo tapestry.charset.

3.3 Parámetros del los componentes

Los parámetros de los componentes permiten que el componente embebido y el contenedor se puedan comunicar. En el siguiente ejemplo el parámetro page es un parámetro del componente pagelink. El parámetro page le indica al componente pagelink que página mostrar cuando el usuario haga clic en el enlace:

```
<!DOCTYPE html>
<html xmlns:t="http://tapestry.apache.org/schema/tapestry_5_4.xsd">
  <t:pagelink page="Index">Ir a la página de inicio</t:pagelink>
</html>
```

Un componente puede tener cualquier número de parámetros. Cada parámetro tiene un nombre específico, un tipo de Java (que puede ser primitivo) y pueden ser opcionales o requeridos.

En la clase de un componente los parámetros son declarados usando la anotación @Parameter en una propiedad privada. En Tapestry un parámetro no es un dato que solo es enviado, es una conexión denominada binding entre la propiedad del componente marcada con la anotación @Parameter y la propiedad del componente contenedor. El binding es de dos sentidos, el componente puede leer el valor de la propiedad asociada a su parámetro y puede actualizar su parámetro con la misma propiedad.

El siguiente componente es un bucle que renderiza su cuerpo varias veces en función de sus parámetros de inicio y fin que establecen los límites del bucle. El componente puede actualizar un parámetro asociado a una propiedad de su contenedor:

```
package io.github.picodotdev.plugintapestry.components;

...

public class Count {

  @Parameter (value="1")
  private int start;

  @Parameter(required = true)
  private int end;

  @Parameter
  private int result;

  @SetupRender
  void initializeValues() {
    result = start;
  }

  @AfterRender
  boolean next() {
    int newResult = value + 1;

    if (newResult <= end) {
      result = newResult; return false;
    }
```

```
    return true;
  }
31 }
```

El nombre del parámetro es el mismo que el nombre de la propiedad. Aquí los parámetros son start, end y result.

3.3.1 Bindings de parámetros

El componente anterior puede ser referenciado en la plantilla de otro componente o página con sus parámetros asociados:

```
1 <!DOCTYPE html>
  <html t:type="layout" xmlns:t="http://tapestry.apache.org/schema/tapestry_5_4.xsd">
    <p>
4     Feliz navidad:
      <t:count end="literal:3">Ho!</t:count>
    </p>
  </html>
```

El literal indicado es asociado al parámetro end del componente Count. Aquí, está siendo asociado con el valor del String 3, que es automáticamente convertido por Tapestry al tipo de la propiedad del componente Count. Cualquier número de parámetros puede ser asociados de esta manera.

Expresiones de binding

El valor dentro de una plantilla, «3» en el anterior ejemplo, es una expresión de binding. Colocando un prefijo delante del valor puedes cambiar como se interpreta el resto de la expresión (lo siguiente de los dos puntos). Los bindings disponibles son:

- asset: El path relativo a un archivo de asset que debe existir (en el classpath).
- context: asset localizado la ruta a partir del contexto de la aplicación.
- block: el id de un bloque dentro de la plantilla.
- component: el id de otro componente dentro de la misma plantilla.
- literal: una cadena interpretada como un literal.
- nullfieldstrategy: usado para localizar un NullFieldStrategy predefinido.
- message: obtiene una cadena del catálogo de mensajes de componente (o aplicación).
- prop: una expresión de una propiedad de la que leer y actualizar.

- symbol: usado para leer uno de tus símbolos.
- translate: el nombre de una translator configurado.
- validate: una especificación de validación usada para crear un número de validadores de propiedades de forma declarativa.
- var: una variable de la que leer y actualizar.

La mayoría de estos prefijos de binding permiten a los parámetros asociarse con valores de solo lectura. Por ejemplo, un parámetro asociado a «message:alguna-clave» obtendrá el mensaje asociado a la clave «alguna-clave» del catálogo de mensajes del contenedor, si el componente intenta actualizar el parámetro (asignando un valor a la propiedad) se lanzará una excepción en tiempo de ejecución para indicar que el valor es de solo lectura. Solo los prefijos de binding prop: y var: son actualizables (pero sin ser usados como una expresión ${...}). Cada parámetro tienen un prefijo por defecto definido por el componente, ese prefijo será el usado cuando no sea proporcionado. El prefijo por defecto cuando no se indica en la anotación @Parameter es prop:. Los más comunes son literal: y prop:. Hay otro prefijo especial, inherit: usado para heredar parámetros de binding.

Variables de renderizado (var:)

Los componentes puede tener cualquier número de variables de renderizado. Las variables son valores con nombre sin un tipo específico (en último término son almacenados en un Map). Las variables son útiles para almacenar valores simples, como los índices en los bucles que se necesitan pasar de un componente a otro. Por ejemplo, la siguiente plantilla:

```
<ul>
  <li t:type="loop" source="1..10" value="indice">${indice}</li>
</ul>
```

Y el siguiente código Java

```
@Property
private int indice;
```

Podría reescribirse así:

```
<ul>
  <li t:type="loop" source="1..10" value="var:indice">${var:indice}</li>
</ul>
```

En otras palabras, en este caso no es necesario definir una propiedad en el código Java. La desventaja es que las variables de renderizado no funcionan con la sintaxis de expresiones de propiedades, de modo que puedes pasar los valores de las variables pero no puedes referenciar ninguno de las propiedades del valor. Las variables de renderizado son automáticamente limpiadas cuando el componente termina de renderizarse. Son insensibles a mayúsculas.

Propiedad (prop:)

El prefijo prop: indica una expresión de propiedad. Las expresiones de propiedades son usadas para enlazar un parámetro de un componente a una propiedad de su contenedor. Las expresiones de propiedades pueden navegar por una serie de propiedades y/o invocar métodos. El binding por defecto de un parámetro es prop: por lo que es habitual que esté omitido. Algunas ejemplos de expresiones de propiedad son:

- this
- null
- userName
- user.address.city
- user?.name groupList.size()
- members.findById(user.id)?.name
- 1..10
- 1..groupList.size()
- 'Beer is proof that God loves us and wants us to be happy.'
- [user.name, user.email, user.phone]
- !user.deleted
- !user.middleName
- { 'framework' : 'Tapestry', 'version' : version }

Validate (validate:)

Este prefijo está altamente especializado en convertir una cadena corta usada para crear y configurar los objetos que realizan la validación para los controles de componente de formulario como TextField y Checkbox. La cadena es una lista separada por comas de tipos de validadores. Estos son alias para los objetos que realizan la validación. En muchos casos la validación es configurable de alguna forma. Un validador que asegura una longitud mínima necesita conocer cual es la longitud mínima. Esos valores se especifican detrás del signo igual. Por ejemplo, «validate:required,minLength=5» asegura que la propiedad tiene un valor de al menos 5 caracteres.

Translate (translate:)

Este prefijo también está asociado con la validación de datos. Es el nombre de un Translator configurado responsable de convertir el valor entre el servidor y el valor en el cliente que siempre será un String (por ejemplo entre el id de una entidad y su objeto). Se pueden añadir nuevos translators usando el servicio TranslatorSource.

Asset (asset:)

Son usadas para especificar bindings de assets (contenido estático servido por Tapestry). Por defecto, los assets están localizados relativamente a la clase del componente donde está empaquetado. Esto puede ser redefinido usando el prefijo «context:», en cuyo caso la ruta es relativa al contexto de la aplicación. Dado que acceder a recursos del contexto es muy común existe el prefijo context:.

Context (context:)

Son como los bindings de assets pero su ruta es siempre relativa al contexto de la aplicación web. Se indica de la siguiente manera en las plantillas:

```
1 <img src="${context:images/icon.png}"/>
```

Tapestry ajustará la URL de la imagen para que sea procesada por Tapestry y no por el contenedor de servlets. La URL tendrá un hash obtenido a partir del contenido del asset, una expiración lejana y si el cliente lo soporta su contenido será comprimido con gzip antes de ser enviado al cliente.

3.4 La anotación @Parameter

3.4.1 Parámetros requeridos

Los parámetros que son requeridos deben de indicarse. Ocurrirá una excepción en tiempo de ejecución si el componente tiene parámetros requeridos y no le son proporcionados al usarlo.

```
1 public class Componente {

    @Parameter(required = true)
4   private String parámetro;
  }
```

Algunas veces el parámetro está marcado como requerido pero aún así puede ser omitido si el valor es proporcionado de alguna otra forma. Este es el caso del parámetro valor del componente Select que puede ser proporcionada por un ValueEncoderSource (lee la documentación de los parámetros del componente Select detenidamente). El ser requerido simplemente comprueba que el parámetro está asociado no significa que se ha de proporcionar en la plantilla (o con la anotación @Component).

3.4.2 Parámetros opcionales

Los parámetros son opcionales a no ser que se marque como requeridos. Se puede especificar un valor por defecto usando el atributo value de la anotación @Parameter. En el componente Count anterior el parámetro start tiene el valor por defecto 1. Ese valor es usado a menos que el parámetro start esté asociado, el valor asociado se superpone al valor por defecto.

Valores por defecto de bindings

El valor de la anotación @Parameter puede ser usado para especificar una expresión de binding por defecto. Normalmente, es el nombre de una propiedad que calculará el valor al vuelo:

```
@Parameter(value="defaultMessage")
private String mensaje;

@Parameter(required=true)
private int longitudMaxima;

public String getDefaultMessage() {
  return String.format("La longitud máxima del campo es %d.", longitudMaxima);
}
```

Como en cualquier otro lugar, puedes usar cualquier prefijo para el valor. Un prefijo común es usar message: para acceder al mensaje localizado.

Cacheo de parámetros

Leer un parámetro puede ser algo marginalmente costoso (si hay conversión). Por eso el parámetro se cachea mientras el componente está activo renderizandose. En algunos casos raros es deseable desactivar el cacheo que puede hacerse estableciendo el atributo cache de la anotación @Parameter a false.

No uses la sintaxis de las expansiones ${...}

Generalmente no deberías usar la sintaxis de las expansiones al asociar los bindings de los parámetros. Hacerlo produce que el valor contenido en la expresión sea convertido a un String inmutable y por tanto se producirá una excepción si el componente necesita actualizar el valor. Esto es correcto:

```
1 <t:textfield t:id="color" value="color"/>
  <img src="${context:images/banner.png}"/>
```

Esto es incorrecto en ambas lineas:

```
1 <t:textfield t:id="color" value="${color}"/>
  <img src="context:images/banner.png"/>
```

La regla general es usar solo la sintaxis ${...} en lugares no controlados por Tapestry de la plantilla, estos son en atributos de elementos html ordinarios y lugares de texto plano de la plantilla.

3.4.3 Parámetros informales

Varios componentes soportan parámetros informales, que son parámetros adicionales no entre los definidos. Los parámetros informales serán renderizados en la salida como atributos adicionales de la etiqueta que renderiza el componente. Generalmente los componentes que tienen una relación de 1:1 con una etiqueta html particular (como entre TextField y una etiqueta input) soportan parámetros informales.

Solo los componentes que son anotados con @SupportsInformalParameters soportarán parámetros informales. Tapestry eliminará silenciosamente los parámetros informales en los componentes que no tienen esta anotación.

Son usados a menudo para establecer el atributo class de un elemento o para especificar manejadores de evento javascript para el cliente. El binding por defecto de los parámetros informales depende en donde se especifiquen. Si el parámetro es especificado en la clase Java con la anotación @Component, entonces el binding por defecto es prop:. Si el parámetro es asociado en la plantilla entonces el binding por defecto es literal:.

Los parámetros informales si están soportados son siempre renderizados en la salida a menos que estén asociados con una propiedad cuyo valor sea null. Si la propiedad asociada es null entonces el parámetro no estará presente en la salida. Si uno de tus componentes debe renderizar parámetros informales simplemente inyecta ComponentResources en el componente e invoca el método renderInformalParameters().

```
@SupportsInformalParameters
public class Image {

  @Parameter(required=true, allowNull=false, defaultPrefix=BindingConstants.ASSET)
  private Asset src;

  @Inject private ComponentResources resources;

  boolean beginRender(MarkupWriter writer) {
    writer.element("img", "src", src);
    resources.renderInformalParameters(writer);
    writer.end();
    return false;
  }
}
```

En este caso los parámetros informales serán emitidos como atributos de la etiqueta img. En el siguiente los parámetros informales serán emitidos en un componente embebido.

```
@SupportsInformalParameters
public class Bloque {

  @Component(inheritInformalParameters = true)
  private Any capa;
}
```

También, en la plantilla de un componente puede usarse el mixin RenderInformals para emitir los parámetros informales en una etiqueta determinada.

Los parámetros son bidireccionales

Los parámetros no son simplemente variables, cada parámetro representa una conexión o binding entre un componente y una propiedad de su contenedor. Cuando es usado con el prefijo prop: el componente puede forzar cambios en una propiedad de su contenedor simplemente asignando un valor a su propiedad.

```
<!DOCTYPE html>
<t:layout xmlns:t="http://tapestry.apache.org/schema/tapestry_5_4.xsd">
  <p>
    Cuenta adelante: <t:count start="1" end="5" result="indice"> ${indice} ... </t:count>
  </p>
</t:layout>
```

Dado que el componente Count actualiza su parámetro result, la propiedad índice del componente contenedor es actualizado. Dentro del cuerpo del componente Count se emite el valor de la propiedad índice usando la expansión ${indice}. El resultado sería el siguiente:

```
<p>Cuenta adelante: 1 ... 2 ... 3 ... 4 ... 5 ... </p>
```

La parte relevante es que los componentes pueden leer valores fijos o propiedades vivas de su contenedor y del mismo modo pueden cambiar las propiedades de su contenedor.

Parámetros no asociados

Si un parámetro no es asociado (porque es opcional) entonces el valor puede ser leído o actualizado en cualquier momento. Las actualizaciones sobre parámetros no asociados no causan excepciones (aunque puede que sí si son leídos y tienen un valor nulo).

3.4.4 Conversiones de tipo en parámetros

Tapestry proporciona un mecanismo para convertir tipos de forma automática. Habitualmente esto es usado para convertir literales String en valores apropiados pero en otros casos pueden ocurrir conversiones más complejas.

Nombres de parámetros

Por defecto el nombre del parámetro es obtenido del nombre de la propiedad eliminado los caracteres $ y _. Otro nombre del parámetro puede ser especificado con el atributo name de la anotación @Parameter.

Determinando si está asociado

En casos raros puedes necesitar diferentes comportamientos en base a si el parámetro está asociado o no. Esto puede ser llevado a cabo preguntado al ComponenteResources que puede ser inyectado en el componente usando la anotación @Inject:

```
public class MiComponente {

  @Parameter
  private int param;

  @Inject
```

```
  private ComponentResources resources;

  void beginRender() {
    if (resources.isBound("param")) {
      ...
    }
  }
}
```

El ejemplo anterior ilustra la aproximación, dado que el tipo es primitivo es difícil distinguir entre no asociado y estar asociado con el valor 0. La anotación @Inject inyectará el ComponenteResources para el componente.

Aunque no lo explicaré en este libro los parámetros pueden heredarse, se pueden proporcionar valores por defecto calculados y se pueden publicar parámetros de componentes embebidos pero dado que son conceptos medianamente avanzados los he dejado sin explicar.

3.5 La anotación @Cached

En el modelo pull que sigue Tapestry es la vista la que pide los datos al controlador y no el controlador el que proporciona los datos a la vista como se hace en el modelo push. Un problema que puede plantear el que la vista pida los datos al controlador es que si la devolución de los datos solicitados son costosos en tiempo del cálculo, carga para el sistema en CPU o memoria, o intensivos en entrada/salida de disco o red y se piden varias veces puede suponer como resultado que el tiempo empleado para generar la página sea elevado o la aplicación consuma recursos innecesarios.

La anotación Cached permite cachear el resultado de un método a nivel de componente y página durante la generación de la misma de modo que un método costoso solo se evalúe una vez. Su uso sería el siguiente:

Listado 3.5: Label.java

```
package io.github.picodotdev.tapestry.components;

...

public class Label {

    @Parameter
    private Label label;

    @Parameter
    private Integer page;

    @Inject
    private MainService service;

    void setupRender() {
```

```
        page = (page == null) ? 0 : page;
    }

    /**
21   * Método que devuelve los articulos publicados o actualizados más recientemente de
    una etiqueta.
     */
    @Cached(watch = "label")
    public List<Post> getPosts() {
        List<Sort> sorts = new ArrayList<>();
26      sorts.add(new Sort("date", Direction.DESCENDING));
        Pagination pagination = new Pagination(Globals.NUMBER_POSTS_PAGE * page, Globals.
    NUMBER_POSTS_PAGE * (page + 1), sorts);
        return service.getPostDAO().findAllByLabel(label, pagination);
    }

31  @Cached(watch = "label")
    public Long getPostsCount() {
        return service.getPostDAO().countBy(label);
    }
}
```

En este ejemplo cada vez que se llama a los métodos getPosts, getPostsCount se accede a una base de datos (o sistema externo) que lanza una consulta, supongamos, costosa de calcular o que simplemente es innecesaria hacerla varias veces. Usando la anotación Cached podemos hacer la aplicación más eficiente evitando las segundas llamadas a los métodos. Si el componente Label del ejemplo se usa dentro de un bucle de un componente loop y como parámetros se le van pasando varios labels las llamadas a los métodos getPosts y getPostCount se realizarán solo para cada valor diferente.

Algunas veces puede interesarnos que el cacheo dependa de un dato, es decir, que para cada valor de un dato la anotación Cached devuelva diferentes resultados. Y esto es lo que se hace en el ejemplo con el parámetro watch de la anotación, por cada valor de la propiedad label el resultado probablemente sea diferente pero nos interesa que el método solo se ejecute una vez por cada diferente valor, dado que los artículos y el número de ellos únicamente variarán en función de esta propiedad. Esto también puede ser usado para que solo se evalúe los métodos una vez por iteración de un bucle estableciendo la expresión watch al índice del bucle.

Listado 3.6: Label.tml

```
1  <!DOCTYPE html>
   <t:container xmlns="http://www.w3.org/1999/xhtml" xmlns:t="http://tapestry.apache.org/
       schema/tapestry_5_4.xsd" xmlns:p="tapestry:parameter">

   <t:loop source="posts" value="post">
5    <t:postcomponent post="post" excerpt="true"/>
   </t:loop>

   </t:container>
```

Aún así, la anotación Cached funciona a nivel de petición, cada vez que que se haga una petición a la aplicación y se llame al método anotado por primera vez y por cada valor de la expresión watch se ejecutará el método. Si tenemos muchas peticiones o un determinado componente tarda mucho en generar su contenido, por ejemplo, porque depende de un sistema externo lento (base de datos, http, ...) quizá lo que debamos hacer es un componente que almacene durante un tiempo el contenido que genera y sea devuelto en múltiples peticiones, de modo que evitemos emplear un tiempo costoso en cada petición. Para ello, podríamos desarrollar un componente que usase una librería de cache como por ejemplo EHCache.

3.6 Conversiones de tipos

Tapestry realiza conversiones de tipo o type coercions automáticas para los parámetros de los componentes. Se produce una conversión cuando el tipo del parámetro pasado no se corresponde con el tipo del parámetro que espera el componente. Por ejemplo considerando el componente Count:

```
1  public class Count {
2
     @Parameter
     private int start = 1;

     @Parameter(required = true)
7    private int end;

     @Parameter
     private int value;

12   ...
   }
```

Aquí, el tipo de los tres parámetros es un int. Sin embargo, el componente puede ser usado de la siguiente manera:

```
1  ¡Feliz navidad!: <t:count end="3"> Ho! </t:count>
```

Una cadena formada por números es interpretada por el prefijo de binding prop como un long. Tapestry convertirá el tipo de ese valor automáticamente, un long, al tipo del parámetro, un int. De modo que el valor int pasado como parámetro es 3. Esta puede ser una conversión con pérdida si el valor almacenado en el long es mayor de lo que es capaz de almacenarse en un int.

Estas conversiones nos facilitarán el desarrollo ya que no tendremos que estar realizando las conversiones nosotros mismos sino que se encargará Tapestry de hacerlas de forma automática y transparente para nosotros, solo deberemos proporcionarle las clases que hagan la conversión entre dos tipos haciendo una contribución al servicio TypeCoercer. Tapestry ya proporciona built-in la mayoría de las conversiones que podamos necesitar (ver Conversiones de tipos del capítulo Páginas y componentes).

Servicio TypeCoercer

Este servicio es responsable de realizar las conversiones y parte del módulo tapestry-ioc. Es extensible permitiendo añadirle fácilmente nuevos tipos y conversiones. El módulo TapestryIOCModule de tapestry-ioc (ver su código fuente) contribuye con una pocas conversiones adicionales al servicio.

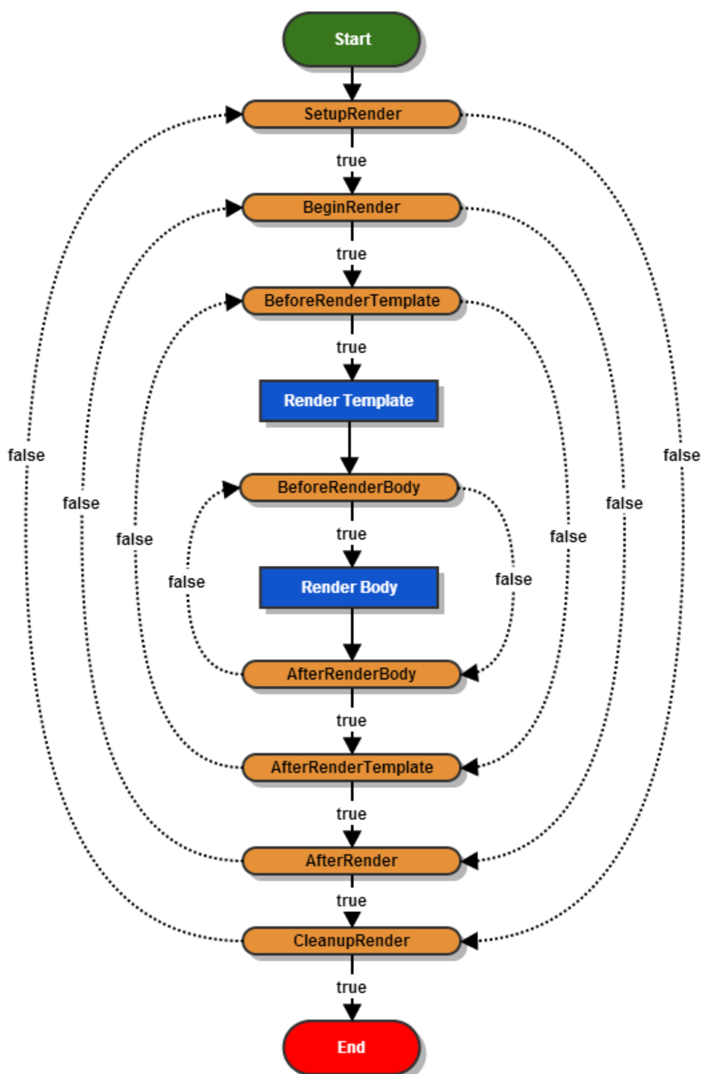
```
@Contribute(TypeCoercer.class)
public static void provideBasicTypeCoercions(Configuration<CoercionTuple> configuration)
       {
  ...
  // String to BigDecimal is important, as String->Double->BigDecimal would lose
  // precision.
  add(configuration, String.class, BigDecimal.class, new Coercion<String, BigDecimal>() {
    public BigDecimal coerce(String input) {
      return new BigDecimal(input);
    }
  });
  ...
}
```

3.7 Renderizado de los componentes

El render de los componente en Tapestry 5 se basa en una máquina de estados y una cola (en vez de recursividad como era en Tapestry 4). Esto divide el proceso de renderizado en pequeñas piezas que pueden ser fácilmente implementadas y sobrescritas. No te preocupes, en la práctica escribir un componente requiere escribir muy poco código.

3.7.1 Fases de renderizado

El renderizado de cada componente se divide en las siguiente fases.



Cada una de las fases naranjas y redondeadas en los bordes (SetupRender, BeginRender, BeforeRenderBody, etc.) tienen una anotación que puedes colocar en uno o más métodos de la clase del componente. Estas anotaciones hacen que Tapestry invoque los métodos como parte de esa fase. Los métodos marcados con estas anotaciones se llaman métodos de fase de render o render phase methods. Estos métodos pueden retornar un void o retornar un valor boolean. Dependiendo del valor retornado se puede forzar a omitir fases o ser revisitadas. En el diagrama, las lineas sólidas muestran el camino normal de procesado y las lineas punteadas son flujos alternativos que pueden lanzarse cuando los métodos de fase de renderizado retornan false en vez de true (o void). Dedica un poco de tiempo a comprender este párrafo y diagrama.

Los métodos de fase de renderizado no deben tener parámetros o un único parámetro de tipo MarkupWriter. Los métodos puede tener cualquier visibilidad, típicamente se usa la de paquete, dado que esta visibilidad permite a las pruebas unitarias probar el código (desde el mismo paquete) sin hacer que los métodos formen parte de la API pública. Estos métodos son opcionales y se asocia con un comportamiento por defecto.

El amplio número de fases refleja el uso de los mixins de componentes que también se unen a las fases de render. Varias de estas fases existen exclusivamente para los mixins. Generalmente, tu código usará una o dos de las fases SetupRender, BeginRender, AfterRender, CleanupRender...

En el siguiente código fuente de un componente de bucle que cuenta hacia arriba o abajo entre dos valores y renderiza su cuerpo un número de veces almacenando el valor del índice actual en el parámetro:

```
package io.github.picodotdev.plugintapestry.components;

...

public class Count {

  @Parameter
  private int start = 1;

  @Parameter(required = true)
  private int end;

  @Parameter
  private int value;

  private boolean increment;

  @SetupRender
  void setup() {
    value = start;
    increment = start < end;
  }

  @AfterRender
  boolean next() {
    if (increment) {
      int newValue = value + 1;
      if (newValue <= end) {
        value = newValue;
        return false;
      }
    } else {
      int newValue = value - 1;
      if (newValue >= end) {
        value = newValue;
        return false;
      }
    }
    return true;
  }
}
```

Retornar falso en el método next() provoca que Tapestry reejecute la fase BeginRender y desde ahí, rerenderice el cuerpo del componente (este componente no tiene una plantilla). Retornar true en este método lleva

hacia la transición de la fase CleanupRender. Nota como Tapestry se adapta a tus métodos marcados con las anotaciones. También se adapta en términos de parámetros, los dos métodos anotados no realizan ninguna salida de modo que no necesitan el parámetro MarkupWriter. Lo que es realmente interesante es que la plantilla y el cuerpo del componente tendrá a menudo más componentes. Esto significa que diferentes componentes estarán en diferentes fases en su propia máquina de estados.

SetupRender

La fase SetupRender (@SetupRender) es donde puedes realizar cualquier configuración de una sola vez para el componente. Este es un buen lugar para leer los parámetros del componente y usarlos para establecer las variables temporales.

BeginRender

La fase BeginRender (@BeginRender) ocurre al inicio de la renderización del componente. Para componentes que renderizan una etiqueta, el inicio de la etiqueta debería renderizarse aquí (la etiqueta de cierre debería renderizarse en la fase AfterRender). El componente puede prevenir que la plantilla y/o el cuerpo se renderice devolviendo false. Los componentes pueden o no tener una plantilla. Si un componente tiene una plantilla, y la plantilla donde se usa incluyen un cuerpo, entonces la fase BeforeRenderBody será lanzada (dando la oportunidad al componente de renderizar su cuerpo o no). Si un componente no tiene un cuerpo en su plantilla, entonces la fase BeforeRenderBody no es lanzada. Si el componente no tiene plantilla pero tiene un cuerpo, entonces la fase BeforeRenderBody es aún así lanzada. Si no hay métodos anotados con @BeginRender, entonces no se emite nada en esta fase pero la plantilla (si existe) o el cuerpo (si no hay plantilla, pero el componente tiene cuerpo) será renderizado.

BeforeRenderTemplate

La fase BeforeRenderTemplate (@BeforeRenderTemplate) existe para permitir a un componente decorar su plantilla (creando etiquetas alrededor de las generadas por la plantilla) o para permitir a un componente prevenir el renderizado de su plantilla.

BeforeRenderBody

La fase BeforeRenderBody (@BeforeRenderBody) está asociada con el cuerpo de un componente (la porción de la plantilla contenida por el componente). Permite a un componente evitar el renderizado del cuerpo mientras permite renderizar el resto de la plantilla del componente (si tiene). Si no hay métodos anotados con BeforeRenderBody entonces el cuerpo se renderizará por defecto. De nuevo, esto ocurre cuando el cuerpo de la plantilla del componente se procese o automáticamente si el componente no tiene plantilla (pero el componente tiene cuerpo).

AfterRenderBody

La fase AfterRenderBody (@AfterRenderBody) es ejecutada después de que el cuerpo sea renderizado, esto solo ocurre si el componente tiene cuerpo.

AfterRender

La fase AfterRender (@AfterRender) complementa la fase BeginRender y es usada habitualmente para renderizar la etiqueta de cierre que corresponde con la etiqueta de inicio emitida en la fase BeginRender. En cualquier caso la fase AfterRender puede continuar en la fase CleanupRender o volver a la fase BeginRender (como ocurre en el ejemplo de componente Count de arriba). Si no hay métodos anotados con AfterRender, entonces no se produce ninguna salida en esta fase y la fase CleanupRender es lanzada.

CleanupRender

La fase CleanupRender (@CleanupRender) complementa la fase SetupRender permitiendo una limpieza.

Usando nombre de métodos en vez de anotaciones

Si prefieres evitar usar anotaciones en tus métodos, puedes hacerlo dándoles a los métodos unos nombres específicos. El nombre del método requerido es el del nombre de la anotación con la primera letra descapitalizada: setupRender(), beginRender(), etc.

Usando esta forma, el ejemplo anterior puede ser reescrito de la siguiente forma sin anotaciones:

```
package io.github.picodotdev.plugintapestry.components;

import org.apache.tapestry5.annotations.Parameter;

public class Count {

  @Parameter
  private int start = 1;

  @Parameter(required = true)
  private int end;

  @Parameter
  private int value;

  private boolean increment;
```

```
19  void setupRender() {
      value = start;
      increment = start < end;
    }

    boolean afterRender() {
24    if (increment) {
        int newValue = value + 1;
        if (newValue <= end) {
          value = newValue;
          return false;
29      }
      } else {
        int newValue = value - 1;
        if (newValue >= end) {
          value = newValue;
34        return false;
      }
      return true;
    }
}
```

Con este estilo las ventajas son que el código es más simple y corto y los nombres de los métodos serán más consistentes de una clase a otra. La desventajas es que los nombres son muy genéricos y pueden en algunos casos ser menos descriptivos que usando las anotaciones. Los métodos initializeValue() y next() son más descriptivos para algunas personas. Por supuesto, puedes tener una mezcla, nombres de métodos de fase para unos casos y anotaciones para otros métodos de fase en otros casos.

Componentes de renderizado

En vez de devolver verdadero o falso, un método de fase de renderizado puede retornar un componente. El componente puede haber sido inyectado por la anotación @Component o puede haber sido pasado por el componente que lo contiene como un parámetro. En cualquier caso, retornar el componente pondrá en la cola ese componente para renderizarse antes de que el componente activo continúe renderizandose. Este componente puede renderizar una página completamente diferente de la aplicación. La renderización recursiva de los componentes no está permitida. Esta técnica permite que el renderizado de las páginas sea altamente dinámico. Retornar un componente no corta la invocación de los métodos del modo que retornar un booleano podría. Es posible que múltiples métodos retornen componentes aunque es desaconsejado.

Tipos adicionales de retorno

Los métodos de fase pueden retornar también componentes Block, Renderable o RenderCommand. El siguiente componente retorna un Renderable en la fase BeginRender y evita la fase BeforeRenderTemplate:

```java
public class OutputValueComponent {

  @Parameter
  private String value;

  Object beginRender() {
    return new Renderable() {
      public void render(MarkupWriter writer) {
        writer.write(value);
      }
    };
  }
}
```

3.7.2 Conflictos y ordenes de métodos

Es posible tener múltiples métodos anotados con al misma anotación de fase. Esto puede incluir métodos en la misma clase o una mezcla de métodos definidos en una clase y herencia de otras clases.

Mixins antes de componente

Cuando un componente tienen mixins, entonces los métodos de fase de los mixins se ejecutan antes que los métodos de fase de renderizado del componente. Si un mixin extiende de una clase base, entonces los métodos de la clase padre se ejecutan antes que los métodos de fase de la clase hija. Excepción: Los mixins cuya clase es anotada con @MixinAfter son ordenados después del componente en vez de antes.

El orden en que los mixins de una clase son ejecutados es determinado por las restricciones de orden especificadas para los mixins. Si no se proporcionan restricciones el orden es indefinido.

Padres antes que los hijos

El orden es siempre los padres primero. Los métodos definidos en la clase padre siempre son invocados antes que los métodos definidos en la clase hija. Cuando una clase sobrescribe un método de fase de una clase base, el método solo es invocado una vez , de la misma forma que cualquier otro método de la clase base. La subclase puede cambiar la implementación de la clase base mediante una sobrescritura, pero no puede cambiar el momento que en el que el método es invocado.

Orden inverso para AfterXXX y CleanupRender

Las fases AfterXXX existen para balancear las fases BeginXXX. A menudo los elementos empiezan en la fase en la fase anterior BeginXXX y son finalizados en la fase AfterXXX correspondiente (con el cuerpo y la plantilla

del componente renderizandose en medio). Para garantizar que las operaciones ocurren en el orden correcto y natural las fases de renderizado de estos dos estados ocurren en el orden inverso.

Orden de los métodos BeginXXX:

- Métodos de la clase padre del mixin
- Métodos del subclase del mixin
- Métodos de clase padre
- Métodos de subclase

Orden de los métodos AfterXXX:

- Métodos de subclase
- Métodos de clase padre
- Métodos del subclase del mixin
- Métodos de la clase padre del mixin

Dentro de una misma clase

Actualmente, los métodos de renderizado marcados con la misma anotación son ejecutados alfabéticamente según el nombre del método. Los métodos con el mismo nombre son ordenados por el número de parámetros. Aún así, anotar múltiples métodos con la misma anotación no es una buena idea. En vez de eso, define un solo método y llama en él a los métodos en el orden que desees.

Cortocircuito

Si un método retorna un valor true o false esto cortocircuitará el procesado. Otros métodos en la fase que serían llamados normalmente no serán invocados. La mayoría de los métodos de fase de renderizado deberían retornar void para evitar cortocircuitos no intencionados sobre otros métodos de la misma fase.

3.8 Navegación entre páginas

En esencia, una aplicación Tapestry es un número de páginas relacionadas trabajando juntas. De cierta forma, cada página es como una aplicación en si misma. Cualquier petición hará referencia a una sola página. Las peticiones llegan en de dos formas:

- Como peticiones de eventos de componentes, que tienen como objetivo un componente específico de una página produciendo un evento en ese componente.

- Peticiones de renderizado de una página específica que producen el lenguaje de marcas que será enviado al cliente.

Esta dicotomía entre peticiones de eventos de componentes y peticiones renderizado de páginas es nuevo en Tapestry 5. En algunas formas basado en la especificación de los Portlets, diferenciando entre los dos tipos de peticiones alivia un número de problemas tradicionales de las aplicaciones web relacionadas con el botón atrás de los navegadores o al pulsar el botón refrescar en el navegador.

Nombre de página lógico

En ciertos casos, Tapestry acortará el nombre lógico de una página. Por ejemplo, la clase de la página io.github.picodotdev.plu
se le dará un nombre de página lógico de admin/Producto (el sufijo Admin redundante es eliminado). Sin embargo, esto solo afecta a como la página es referenciada en las URL, la plantilla seguirá siendo ProductoAdmin.tml independientemente de si está en el classpath o en la raíz del contexto.

En realidad se crean múltiples nombres para el nombre de la página: admin/Producto y admin/ProductoAdmin son sinónimos, puedes usar cualquiera de ellos en el código Java para referirte a una página por nombre o como la página de un parámetro del componente PageLink.

3.9 Peticiones de eventos de componente y respuestas

Los eventos de componente pueden tomar la forma de de enlaces (EventLink o ActionLink) o como envíos de formularios (Form). El valor retornado desde un método manejador de evento controla la respuesta enviada al navegador del cliente. La URL de una petición de un evento de componente identifica el nombre de la página, el id del componente anidado y el nombre del evento a producir en el componente (que normalmente suele ser action). Es más, una petición de evento de componente puede contener información adicional de contexto, que será proporcionada al método manejador de evento. Estas URL exponen un poco de la estructura interna de la aplicación. A medida que la aplicación sigue su desarrollo y la aplicación crece y se mantiene los ids de los componentes pueden cambiar, esto significa que una URL de un evento de componente no debería ser usada por ejemplo como marcador. Afortunadamente, los usuarios raramente tendrán oportunidad de hacer esto por las redirecciones que ocurren en un breve espacio de tiempo.

Respuesta nula

Si el método manejador de evento no retorna ningún valor, o retorna null, entonces la página actual (la página que contiene el componente) se renderizará de nuevo como respuesta. Un enlace de renderizado de página para la página actual es creado y enviado al cliente como una redirección de cliente. El navegador del cliente automáticamente enviará una nueva petición para generar la página. El usuario verá el nuevo contenido generado en su navegador. Adicionalmente, la URL en la barra de direcciones del navegador cambiará a la URL de renderizado. Las URL de petición de renderizado son más cortas y contienen menos estructura de la aplicación (por ejemplo, no incluyen ids de componentes o tipos de eventos). Las URL de petición de renderizado son lo que

los usuarios pueden guardar como marcadores. Las URL de petición de evento de componente son transitorias, con significado solo en la aplicación actual y no significa que puedan usarse para sesiones posteriores.

```
public Object onAction() {
  return null;
}

public void onActionFromEnlace() {
}
```

Repuesta de cadena

Cuando se devuelve una cadena, se espera que sea el nombre lógico de una página (en contraposición de el nombre cualificado completo de la clase). Como en cualquier otra parte, el nombre de la página no es sensible a mayúsculas. De nuevo, una URL de petición de renderizado se construirá y se enviará al cliente como una redirección.

```
public String onAction() {
  return "Index";
}
```

Repuesta de clase

Cuando se devuelve una clase, se espera que sea una clase de una página. Retornar una clase de página de un manejador de evento es más seguro al refactorizar que retornar el nombre de la página como una cadena. Como en otro tipo de respuestas, una URL de renderizado de página será construido y enviado al cliente como una redirección.

```
public Object onAction() {
  return Index.class
}
```

Repuesta de página

También puedes retornar una instancia de una página en vez del nombre o la clase de la página. Una página podría ser inyectada vía la anotación InjectPage. A menudo, puedes querer configurar la página de alguna forma antes de retornar la página. También puedes retornar un componente dentro de esa página, pero esto generará

una advertencia en tiempo de ejecución (a menos que estés haciendo una actualización parcial de una página vía Ajax).

```
@InjectPage
private Index index;

public Object onAction() {
  index.setTitulo("Título de la página");
  return index;
}
```

Repuesta de error http

Un método manejador de evento puede retornar una instancia de HttpError para enviar un error como respuesta al cliente.

```
import javax.servlet.http.HttpServletResponse;

public Object onAction() {
  return new HttpError(HttpServletResponse.SC_INTERNAL_SERVER_ERROR , "Se ha producido un
     error");
}
```

Repuesta de enlace

Un método manejador de evento puede retornar una instancia de Link directamente. El enlace es convertido a una URL y el cliente es redirigido a esa URL. El objeto ComponentResources que es inyectado en las páginas (y componentes) tiene métodos para crear eventos de componente y enlaces de renderizado de página.

Respuesta de stream

Un manejador de evento puede retornar también un objeto StreamResponse que encapsula un flujo de datos a ser enviado directamente al cliente. Esto es útil para componentes que quieren generar una imagen, un PDF para el cliente o en definitiva un archivo como resultado.

Repuesta de URL

Una respuesta de un java.net.URL es tratada como una redirección a una URL externa. Esto funciona también para las peticiones Ajax.

Repuesta de cualquier otro objeto

Cualquier otro tipo de objeto retornado de un manejador de evento es tratado como un error.

3.10 Peticiones de renderizado de página

Las peticiones de renderizado son más simples en estructura y comportamiento que las peticiones de evento. En el caso más simple, la URL es solamente el nombre lógico de la página. Las páginas pueden tener un contexto de activación, no todas lo tienen. El contexto de activación representa la información persistente del estado de la página. En términos prácticos, el contexto de activación es usado en ocasiones para el id de una entidad persistente en la base de datos. Cuando una página tiene un contexto de activación, los valores del contexto son añadidos a la ruta de la URL. El contexto de activación puede establecerse explícitamente cuando se crea el enlace de renderizado de la página (el componente PageLink tiene un parámetro context para este propósito). Cuando no se proporciona un contexto explícito, la página misma es preguntada por su contexto de activación. Esta pregunta toma la forma de evento. El nombre del evento es passivate (como veremos también hay un evento activate). El valor retornado por el método es usado como el contexto. Por ejemplo:

```
1 public class DetallesProducto {

    @Property
    private Producto producto;
5
    long onPassivate() {
      return producto.getId();
    }
  }
```

El contexto de activación puede consistir en una serie de valores, en cuyo caso el valor retornado debería ser un array o un objeto List.

Nota: si estás usando la librería de integración con hibernate (tapestry-hibernate) y tu contexto de activación es una entidad de Hibernate, entonces puedes simplemente retornar la entidad misma. Tapestry automáticamente extraerá el id de la entidad y lo recuperará de nuevo en el método de evento de activación.

```
1 public class DetallesProducto {

    @Property
    private Producto producto;

6   Producto onPassivate() {
      return producto;
    }
  }
```

Activación de página

Cuando un evento de renderizado de página llega, la página es activada antes de que realice su renderizado. La activación sirve para dos propósitos:

- Permite a la página restaurar su estado interno a partir de los datos codificados en la URL (con el contexto de activación comentado anteriormente).
- Proporciona un mecanismo para validar el acceso a la página.

En el último caso, la validación normalmente implica validación a partir de la identidad del usuario, si posees páginas que solo pueden ser accedidas por ciertos usuarios, puedes usar el evento manejador de activación para verificar el acceso. Un manejador de evento de activación de contexto es similar al manejador de passivate:

```
1  public class DetallesProducto {

     private Producto producto;

     @Inject
6    ProductoDAO dao;

     void onActivate(long id) {
       producto = dao.getById(id);
     }
11 }
```

La parte relevante es que cuando la página se renderiza, es probable que incluya URL de manejadores de eventos de componente (enlaces y formularios). La petición de evento de componente de esos enlaces y formularios cuando sean activados comenzarán también por activar la página antes de hacer cualquier otro trabajo. Esto forma una cadena de peticiones que incluyen el mismo contexto de activación. De alguna forma, el mismo efecto puede conseguirse usando datos persistentes en sesión pero eso requiere activar una sesión y hace que el estado no se conserve al añadir la URL a los marcadores. El manejador de evento de activación también puede retornar un valor, el cual es tratado de forma idéntica al valor retornado por un manejador de evento.

3.11 Patrones de navegación de páginas

Esta combinación de enlaces de acciones, contextos y contextos de página pueden ser usados conjuntamente de cualquier número de formas. Consideremos el ejemplo de la relación de maestro/detalle de un producto de un catálogo. En este ejemplo, la página de listado (ListadoProductos) es una lista de productos y el detalles de producto (DetallesProducto) debe mostrar los detalles de un producto específico.

Patrón 1: Peticiones de evento de componente y datos persistentes

En este patrón, la página de listado de productos (ListadoProductos) usa los eventos de acción y propiedades con datos persistentes en la página de detalle (DetallesProducto).

Listado 3.7: ListadoProductos.tml

```
<t:loop source="productos" value="producto">
  <a t:type="actionlink" t:id="select" context="producto.id">${producto.name}</a>
</t:loop>
```

Listado 3.8: ListadoProductos.java

```java
public class ListadoProductos {

  @InjectPage
  private DetallesProducto detalles;

  Object onActionFromSelect(long id) {
    details.setProductoId(id);
    return detalles;
  }
}
```

Listado 3.9: DetallesProducto.java

```java
public class DetallesProducto {

  @Inject
  private ProductDAO dao;

  @Persist
  private long id;

  private Product producto;

  public void setProductId(long id) {
    this.id = id;
  }

  void onActivate() {
    producto = dao.getById(id);
  }
}
```

Este es la aproximación mínima, tal vez útil para un prototipo. Cuando el usuario hace clic en un enlace, la URL de petición de evento inicialmente será algo así http://.../listadoproductos.select/99 y la URL final de renderizado que recibe el cliente con una redirección es http://.../detallesproducto. Nótese que el id del producto (99) no aparece en la URL de renderizado. Esto tiene varios pequeños defectos:

- Requiere una sesión para almacenar el id del producto entre peticiones.
- Puede fallar si la página DetallesProducto es accedida y no se le proporciona un id.
- La URL no identifica el producto, si el usuario guarda en sus marcadores la URL y la usa luego, se producirá un error (el caso anterior) porque no hay un id válido (aunque podría controlarse).

Patrón 2: Petición de evento de componente sin datos persistentes

Podemos mejorar el ejemplo anterior sin cambiar la página ListadoProductos usando un contexto de activación y pasivación para evitar la sesión y hacer que los enlaces sean guardables en los marcadores.

Listado 3.10: DetallesProducto.java

```
public class DetallesProducto {

  @Inject
  private ProductDAO dao;

  private Product producto;

  private long id;

  public void setProductoId(long id) {
    this.id = id;
  }

  void onActivate(long id) {
    this.id = id;
    producto = dao.getById(id);
  }

  long onPassivate() {
    return id;
  }
}
```

Este cambio asegura que la URL petición de renderizado incluya el id del producto, http://.../detallesproducto/99. Tiene la ventaja que la conexión de página a página ocurre en código donde el compilador comprueba los tipos, dentro del método onActionFromSelect de ListadoProductos. Tiene la desventaja de que haciendo clic en el enlace requiere dos peticiones e idas y venidas del servidor (una para procesar el evento y otra para renderizar la página final).

Patrón 3: Petición de renderizado únicamente

Este es la versión más común de esta relación de maestro/detalle.

Listado 3.11: ListadoProductos.html

```
<t:loop source="productos" value="producto">
  <a t:type="pagelink" page="detallesproducto" context="product.id">${producto.nombre}</a
    >
</t:loop>
```

En este patrón no es necesario ningún código para el enlace como se hacía en los anteriores patrones en ListadoProductos pasando el id del producto (el método setProductoId() no es necesitado ahora).

Listado 3.12: DetallesProducto.java

```
public class DetallesProducto {

  @Inject
  private ProductDAO dao;

  private Producto producto;

  private long id;

  void onActivate(long id) {
    this.id = id;
    product = dao.getById(id);
  }

  long onPassivate() {
    return id;
  }
}
```

Limitaciones

A medida que el flujo entre páginas se expande, puedes encontrarte de que no hay una manera razonable de evitar datos de forma persistente entre peticiones fuera del contexto de activación. Por ejemplo, si en la página DetallesProducto se le permite al usuario navegar a páginas relacionadas y volver a los DetallesProducto entonces empieza a ser necesarios pasar el id del producto de página en página. En algún momento, los valores persistentes tienen sentido para evitar que los datos naveguen de página en página. Tapestry posee varias estrategias de persistencia disponibles, incluyendo una que almacena los datos en los parámetros de la URL.

3.12 Eventos de componente

Los eventos de componentes es la forma de Tapestry de llevar a cabo las interacciones del usuario, tales como hacer clics en enlaces y enviar formularios, y asociarlas a los métodos designados de tu clase de página y componente. Cuando un evento de componente ocurre, Tapestry llama al método manejador de evento que proporcionaste, si proporcionaste alguno, contenido en la clase del componente.

Vamos a revisar un ejemplo simple. Aquí hay una porción de una plantilla de una página que permite al usuario elegir un número entre 1 y 10:

Listado 3.13: Selector.tml

```
<p>
  Elige entre un número de 1 a 10:
  <t:count start="1" end="10" value="index">
    <a t:id="select" t:type="actionlink" context="index">${index}</a>
  </t:count>
</p>
```

Nota que el Selector.tml contiene un componente ActionLink. Cuando es renderizado en la página, el componente ActionLink crea una URL con una petición de evento de componente con el tipo de evento establecido a action. En este caso la URL puede tener la siguiente forma: http://localhost:8080/selector.select/3. Esta URL identifica la página que contiene el componente (Selector), el componente que produce el evento (select), además el valor adicional de contexto (3). Los valores adicionales de contexto, si hay alguno, son añadidos al path de la URL.

No hay una correspondencia directa de una URL a una pieza de código. En vez de ello, cuando el usuario hace clic en el enlace, el componente ActionLink emite un evento. Y entonces Tapestry asegura que el código correcto (tu manejador de evento) sea invocado para ese evento. Esto es una diferencia crítica entre Tapestry y los más tradicionales frameworks orientados a acciones. La URL no dice que sucede cuando en un enlace se hace clic, identifica que componente es responsable de procesar la acción cuando en el enlace se hace clic.

A menudo, una petición de navegación (originada por el usuario) podrá generar varios eventos en diferentes componentes. Por ejemplo, un evento de acción puede ser lanzado por un componente de formulario, que a su vez emitirá un evento de notificación para anunciar cuando el envió del formulario está apunto de ser procesado, que podrá ser exitoso o no, y ese evento puede ser manejado adicionalmente por la página del componente.

3.12.1 Métodos manejadores de evento

Cuando un evento de componente ocurre, Tapestry invoca cualesquiera manejadores de evento que hayas identificado para ese evento. Puedes identificar tus métodos manejadores de evento mediante una convención de nombres o mediante la anotación @OnEvent.

```
1 @OnEvent(component = "select")
  void seleccionValor(int value) {
    this.value = value;
4 }
```

Tapestry hace dos cosas aquí:

- La anotación identifica el método seleccionValor() como el método a invocar.
- Cuando se hace clic en el enlace convierte el valor de contexto de un String a un int y se lo pasa al método.

También se validará si el componente identificado por el manejador del evento existe en la plantilla del componente que lo contiene. Esto ayuda con los errores de escritura en las anotaciones.

En el ejemplo anterior, el método seleccionValor será invocado cuando el evento por defecto, action, ocurra en el componente select (y tiene al menos un valor de contexto). Algunos componentes puede producir varios eventos, en cuyo caso puedes querer ser más específico en el evento a manejar:

```
1 @OnEvent(component = "select", value = "action")
  void seleccionValor(int value) {
    this.value = value;
  }
```

El atributo value de la anotación OnEvent es el nombre del evento a manejar. El tipo del evento por defecto es action; los componentes ActionLink y Form usa cada uno este tipo. De forma alternativa podemos usar el componente EventLink, en cuyo caso el nombre del evento es determinado por el atributo event del elemento en vez de ser action. Si omites el atributo component de la anotación OnEvent, entonces recibirás notificación de todos los componentes contenidos, posiblemente incluyendo componentes anidados (dado el burbujeo de los eventos).

Normalmente especificarás exactamente de que componente quieres recibir eventos. Usando @OnEvent en un método y no especificando un id de un componente específico significa que el método será invocado para los eventos de cualquier componente. Los métodos manejadores de evento tendrán normalmente una visibilidad de paquete, para el soporte de las pruebas unitarias, aunque pueden tener cualquier visibilidad (incluso privada). Un solo método manejador de eventos puede recibir notificaciones de diferentes componentes. Como en otros lugares, la comparación entre el nombre del evento y el id del componente es insensible a mayúsculas.

Convenciones de nombre de métodos

Como alternativa al uso de anotaciones puedes seguir ciertas convenciones en los nombres a los métodos manejadores de eventos y Tapestry los encontrará e invocará de la misma forma que si estuviesen anotados. Este estilo empieza con el prefijo on, seguido del nombre de la acción o evento. Puedes continuar añadiendo From y

un id de componente capitalizado. Así que, si existe un método de nombre onActionFromSelect() es invocado cuando sea emitido un evento action por el componente select. El ejemplo anterior puede ser reescrito como:

```
1 void onActionFromSelect(int value) {
    this.value = value;
  }
```

Valores de retorno de métodos

Para los eventos de navegación de página (originados por componentes como EventLink, ActionLink y Form) el valor retornado por el método manejador de evento determina como Tapestry renderizará la respuesta.

- Null: Para métodos sin valor de retorno (void) o null, se renderizará la página actual (la página que contiene el componente).
- Página: Para un nombre de página, la clase de una página o una instancia de una clase de página se construirá una URL de petición de renderizado de página y será enviada al cliente como una redirección.
- URL: Para un objeto java.net.URL se enviará un redirect al cliente (incluido en peticiones Ajax).
- Zone body: En el caso de una petición Ajax para actualizar una zona, el manejador de evento retornará el cuerpo de la zona, normalmente habiendo inyectado el componente o bloque.
- HttpError: Para un error HttpError se enviará una respuesta de error al cliente.
- Link: Para un Link se enviará una redirección.
- Stream: Para un StreamResponse se enviará los datos del flujo al cliente (un PDF, imagen, ...).

(Ver Navegación entre páginas para más detalles).

Coincidencias de varios métodos

En algunos casos, puede que varios métodos manejadores de evento coincidan con un único evento. En ese caso, Tapestry los invoca en el siguiente orden:

- Los métodos de las clases base antes que las subclases.
- Los métodos coincidentes de la misma clase en orden alfabético.
- Para un método sobrecargado por número de parámetros en orden descendente.

Por supuesto, normalmente no tienes por que crear más de un único método para manejar un evento. Cuando una subclase redefine un manejador de evento de una clase base, el método manejador de evento solo es invocado una vez, del mismo modo que cualquier otro método de la clase base redefinido. La subclase puede cambiar la implementación del método de la clase base mediante una sobrescritura pero no puedes cambiar el momento de cuando ese método es invocado.

Contexto de evento

Los valores de contexto (el parámetro de contexto para el componente del EventLink o ActionLink) puede ser cualquier objeto. Sin embargo, solo ocurre una sola conversión a String. De nuevo independiente de lo que sea el valor (una cadena, número o fecha), es convertido a un String. Esto resulta en una URL más legible. Si tienes múltiples valores de contexto (mediante una lista o array de objetos para el parámetro de contexto del EventLink o ActionLink), entonces se añadirá a la URL cada uno en orden.

Cuando se invoca un manejador de evento, las cadenas son convertidas de nuevo a sus valores u objetos de evento. Se usa un ValueEncoder para convertir entre las cadenas para el cliente y los objetos del servidor. El servicio ValueEncoderSource proporciona los codificadores de valores necesarios. Como se ha mostrado en el ejemplo, la mayoría de los parámetros pasados al método manejador de evento son obtenidos a partir de los valores proporcionados por el contexto del evento. Cada parámetro del evento coincide con un valor proporcionado por el contexto del evento (mediante el parámetro de context del componente ActionLink, varios componentes tienen un parámetro de contexto similar). En algunos casos, es deseable tener acceso directo al contexto (por ejemplo, para adaptarse a casos donde hay un número variable de valores de contexto). El contexto puede ser pasado a un manejador de evento como un parámetro de los siguientes tipos:

- EventContext
- Object[]
- List<Object>

Los últimos dos deberían ser evitados ya que pueden ser eliminados en futuras versiones. En todos estos casos, el parámetro de contexto actual es libre, no coincide con un único valor de contexto dado que representa todos los valores de contexto.

Accediendo a los parámetro de consulta de la petición

Un parámetro puede ser anotado con @RequestParameter, esto permite que un parámetro de consulta (query parameter, ?parametro=valor) se extraiga de la petición, se convierta al tipo correcto y se pase al método. De nuevo, esto no cuenta para los valores de contexto del evento.

```
1 void onActionFromSelect(int value, @RequestParameter int parameter) {
2   this.value = value;
    this.parameter = parameter;
    ...
  }
```

Coincidencia de método

Un método manejador de evento solo será invocado si el contexto contiene al menos tantos valores como parámetros tenga. Los métodos con más parámetros serán silenciosamente ignorados. Tapestry silenciosamente ignorará un método si no hay suficientes valores en el contexto para satisfacer el número de parámetros. Los parámetros EventContext y los parámetros anotados con @RequestParameter no cuentan para este límite.

Ordenación de métodos

Cuando coinciden múltiples métodos en la misma clase, Tapestry los invocará en orden alfabético ascendente. Cuando hay múltiples sobrescrituras de un mismo método con el mismo nombre, Tapestry los invoca en orden descendente según el número de parámetros. En general, estas situaciones no suceden... en la mayoría de casos, solo un único método es requerido para manejar un evento especifico de un componente específico. Un método manejador de evento puede retornar el valor true para indicar que el evento ha sido procesado, esto para inmediatamente la búsqueda de métodos adicionales en la misma clase (o en la clase base) o en los componentes contenedores.

Burbujeo de evento

El evento burbujeará en la jerarquía hasta que sea abortado. El evento es abortado cuando un manejador de evento retorna un valor no nulo. Retornar un valor booleano para un método manejador de evento es tratado de forma especial. Retornar true abortará el burbujeo, usa este valor cuando el evento haya sido procesado de forma completa y no se haya de invocar más manejadores de evento (en el mismo componente o en los componentes contenedores). Retornar false es lo mismo que retornar null, el procesado del evento continuará buscando más manejadores de evento, en el mismo componente o en su padre. Cuando un evento burbujea hacia arriba de un componente a su padre, el origen del evento es cambiado para que coincida con el componente. Por ejemplo, un componente Form dentro de un componente BeanEditForm puede lanzar un evento success. La página que contenga el BeanEditForm puede escuchar por ese evento, pero procederá del componente BeanEditForm (tiene sentido, porque el id del Form dentro del BeanEditFrom es parte de la implementación de BeanEditForm, no de su interfaz pública).

Excepciones en los métodos de evento

A los métodos de evento se les permite lanzar cualquier excepción (no solo excepciones runtime). Si un evento lanza una excepción, Tapestry la capturará y en último término mostrará la página de informe de excepción. Para hacer esto necesitas hacer:

```
1 void onActionFromRunQuery() {
    try {
      dao.executeQuery();
    } catch (JDBCException ex) {
```

```
5       throw new RuntimeException(ex);
    }
  }
```

O más simplemente:

```
1 void onActionFromRunQuery() throws JDBCException {
    dao.executeQuery();
3 }
```

Tu manejador de evento puede declarar incluso que lanza Exception si es más conveniente.

Interceptando excepciones de eventos

Cuando un método manejador de evento lanza una excepción (cheked o runtime, Tapestry da la opción al componente y a su página contenedora la oportunidad de tratar la excepción, antes de continuar con el informe de error. Tapestry emite un nuevo evento del tipo exception pasando la excepción lanzada como contexto. En realidad, la excepción es envuelta dentro de un ComponentEventException del cual puedes extraer el tipo del evento y contexto.

```
1 Object onException(Throwable cause) {
2   message = cause.getMessage();
    return this;
  }
```

El valor de retorno del manejador de evento reemplaza el valor de retorno del método manejador de evento original. Para el caso típico (una excepción lanzada por un evento activate o action) la acción será una navegación de página devolviendo una instancia de página o nombre de página. Esto es útil para manejar casos en los que los datos de la URL están incorrectamente formateados. En el ejemplo anterior la página de navegación es la misma. Si no hay un manejador de evento de excepción o el manejador de evento retorna nulo o es void entonces la excepción será pasada al servicio RequestExceptionHandler que en su configuración por defecto renderizará la página de excepción.

3.13 Componentes disponibles

Tapestry incluye más de 65 componentes y mixins listos para usar. Además de estos, hay otros proporcionados libremente por otras partes. Por supuesto, en Tapestry es trivial crear tus propios componentes personalizados, por lo que si no ves lo que necesitas puedes desarrollarlo tu mismo. Los mixins permiten añadir algún comportamiento a los componentes existentes y se encuentran en el paquete org.apache.tapestry5.corelib.mixins.

Tapestry proporciona varias páginas especiales que proporcionan información de estado, la mayoría se encuentran en el paquete org.apache.tapestry5.corelib.pages. Los componentes base del paquete org.apache.tapestry5.corelib.base tiene la intención de ser extendidos por otros componentes en vez de ser usados directamente en las plantillas.

Los componentes proporcionados por Tapestry pueden dividirse en las siguiente categorías:

- Componentes específicos para Ajax (AjaxFormLoop, AddRowLink, RemoveRowLink, ProgressiveDisplay, Zone).
- Mostrado y edición de beans (BeanDisplay, BeanEditForm, BeanEditor, PropertyDisplay, PropertyEditor).
- Condicionales y de bucle (If, Case, Loop, Unless, Delegate).
- Controles de formulario (Checkbox, Checklist, DateField, Form, FormFragment, FormInjector, Hidden, Label, KaptchaField, KaptchaImage, Palette, PasswordField, Radio, RadioGroup, Select, SubmitNotifier, TextArea, TextField, Upload).
- Grid, tablas y árboles (Grid, GridCell, GridColumns, GridPager, GridRows, GridRows).
- Enlaces y botones (ActionLink, EventLink, LinkSubmit, Submit, PageLink).
- De salida y mensajes (Alerts, Dynamic, Error, Errors, ExceptionDisplay, Output, OutputRaw, TextOutput).
- Mixins (Autocomplete, DiscardBody, FormFieldFocus, NotEmpty, OverrideFieldFocus, RenderClientId, RenderDisabled, RenderInformals, RenderNotification, TriggerFragment, ZoneRefresh).
- Páginas de Tapestry (ExceptionReport, PageCatalog, PropertyDisplayBlocks, PropertyEditBlocks, ServiceStatus).
- Componentes base (AbstractComponentEventLink, AbstractConditional, AbstractField, AbstractLink, AbstractPropertyOutput, AbstractTextField, BaseMessages)
- Diversos (Any, Doctype, RenderObject, Trigger).
- Otras componentes de librerías proporcionadas por terceros.

Con los componentes proporcionados por Tapestry pueden resoverse la mayoría de problemas planteados en una aplicación, de todos ellos los más usados quizá sean:

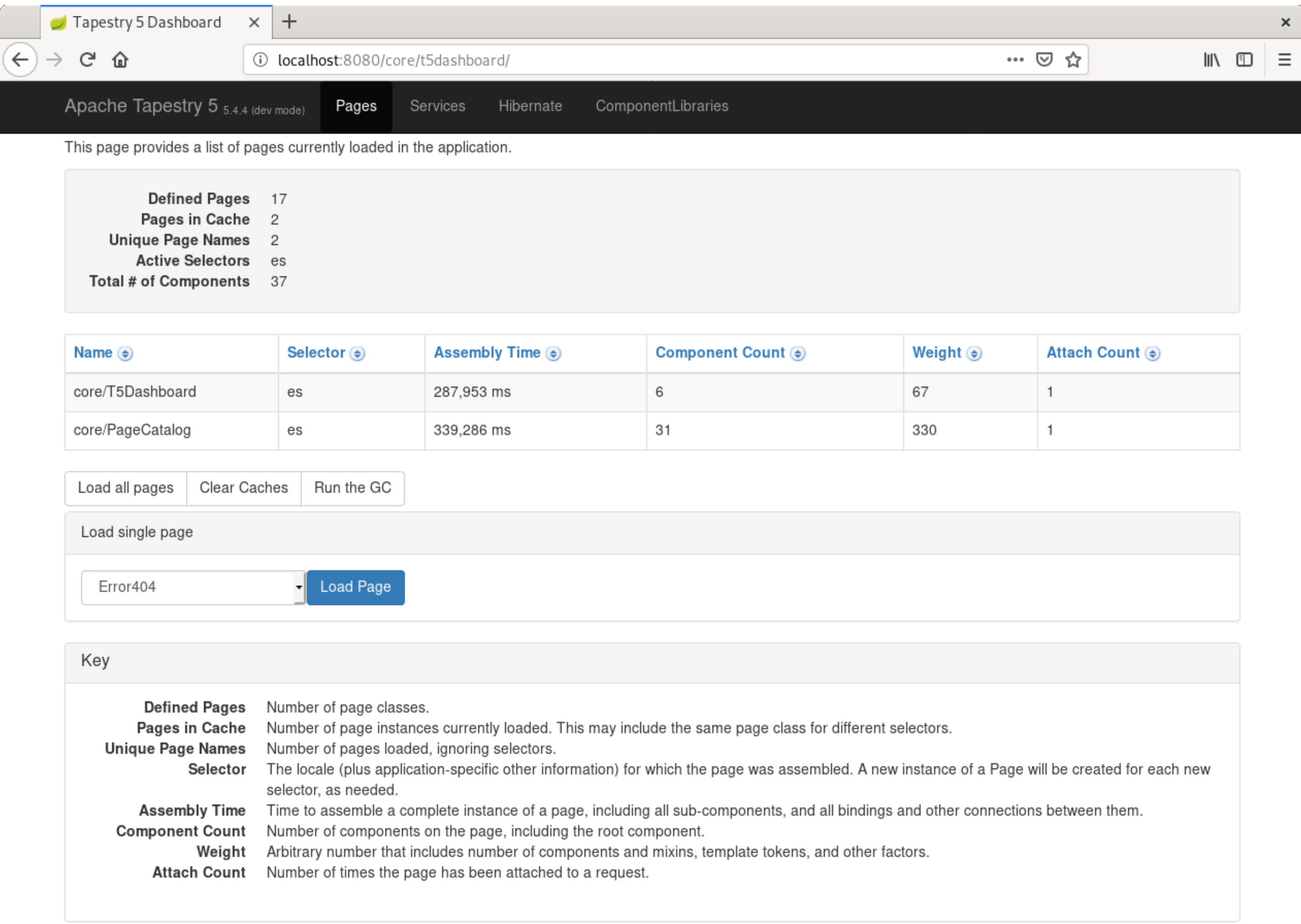
- Zone: una región de una página que puede ser actualizada por Ajax u otros efectos de cliente.
- If: renderiza condicionalmente su cuerpo. Puede renderizar su tag y cualquier parámetro informal.
- Loop: itera sobre una lista de elementos, renderizando su cuerpo por cada uno de ellos.
- Delegate: no proporciona renderizado por si mismo sino que lo delega en otro objeto. Habitualmente un Block.
- Checkbox: renderiza un elemento <input type="checkbox">.
- Form: un formulario html, que incluye otros componentes para renderizar varios tipos de campos de formulario.

- Hidden: usado para una propiedad de la página como un valor del formulario.
- Label: genera un elemento label para un campo particular.
- Radio: un botón radio <input type="radio">. Los radios deben estar incluidos en un RadioContainer, normalmente un componente RadioGroup.
- RadioGroup: agrupa un conjunto de componentes radio que afectan a la misma propiedad.
- Select: renderiza un elemento <select> para seleccionar un elemento de una lista de valores.
- TextArea: renderiza un elemento <textarea> para editar un texto multilinea.
- TextField: renderiza un elemento <input type="text"> para editar una sola linea de texto.
- Grid: Presenta datos usando una etiqueta <table> iterando una lista o array.
- ActionLink: Provoca una acción en el servidor con la consiguiente refresco de página.
- EventLink: Como ActionLink excepto que el evento que se lanza en vez de ser siempre action puede ser especificado.
- LinkSubmit: Genera un enlace que envía el formulario que lo contiene.
- Submit: Se corresponde con un <input type="submit"> o <input type="image"> que puede enviar el formulario que lo contiene.
- PageLink: Genera un enlace de petición de renderizado a otra página de la aplicación.
- Error: Presenta los errores de validación de un solo campo. Debe estar contenido en un Form.
- Errors: Muestra los errores de validación de los campos de un formulario.
- TextOutput: Emite un texto en un párrafo, posiblemente capturado con un componente TextArea. Cada línea es emitida individualmente en un párrafo.
- RenderInformals: renderiza los parámetros informales al finalizar la fase BeginRender.
- PageCatalog: Lista las páginas contenidas en la aplicación con algunas estadísticas.
- Any: Renderiza un elemento arbitrario incluyendo los parámetros informales. Muy útil para permitir a ciertas etiquetas del html tener atributos con expresiones de binding.
- Componentes de terceros.

3.14 Página Dashboard

Con la versión 5.4 de Tapestry las páginas PageCatalog, ServiceStatus e HibernateStaticstis han sido unificadas en la página core/t5dashboard por lo que ahora en una sola página tendremos toda la información. Una de las características más importantes de Tapestry es ser muy informativo proporcionando mucha y descriptiva información, esto se nota con la página de informe de error incluso para las peticiones ajax, los mensajes de logging y con estás páginas de información de estado.

La página T5Dashboard está incluida en el propio core de y disponible en todas las aplicaciones en modo desarrollo y accediendo de forma local al servidor de aplicaciones. Si se incluye en la aplicación la dependencia tapestry-hibernate además en el dashboard podremos ver estadísticas de uso de Hibernate. La página dashboard nos puede resultar muy útil ya que nos proporciona mucha información y alguna acción interesante.

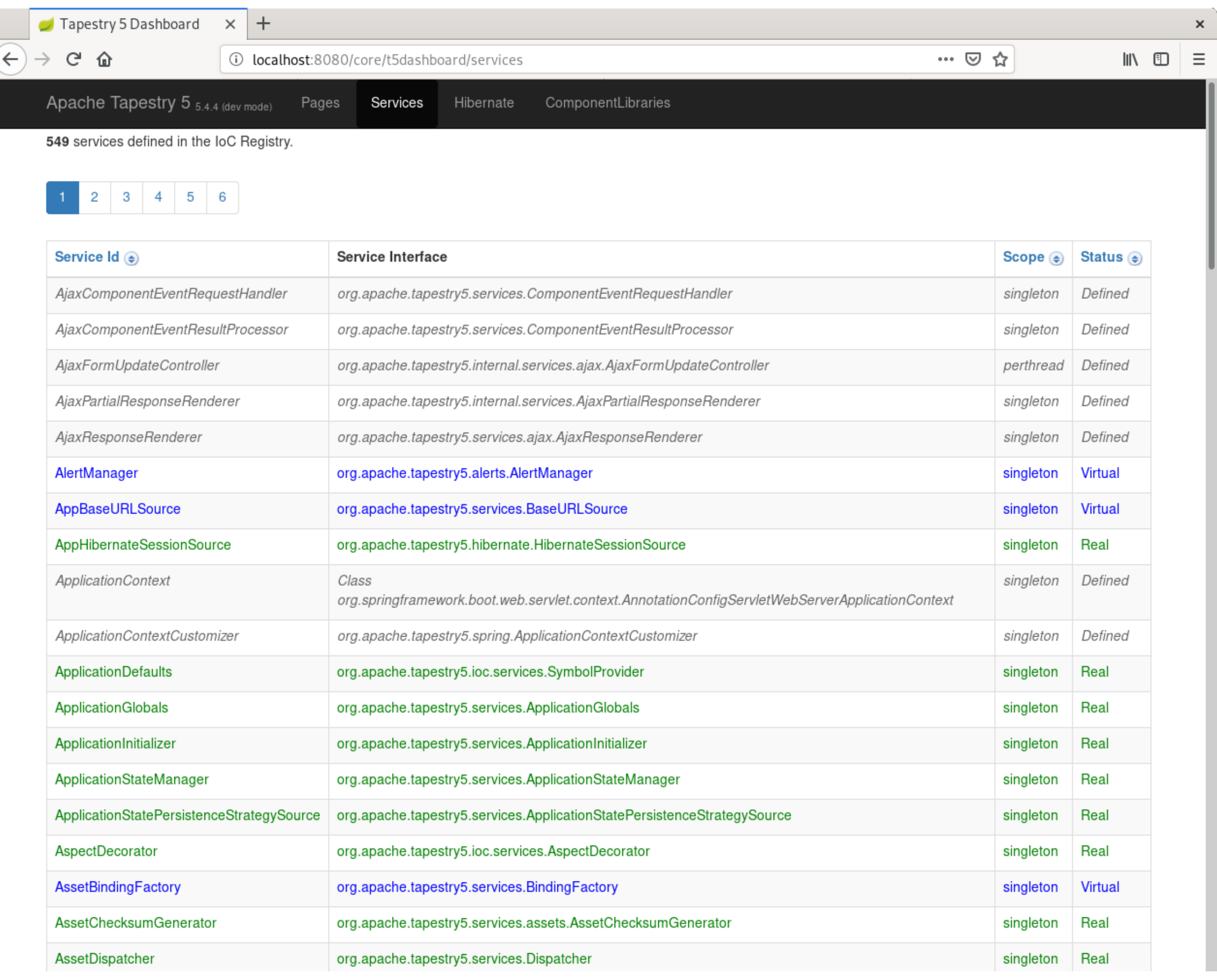


Como se ve en la imagen podemos ver las páginas disponibles, cargadas, cuanto tiempo llevó construirlas, que complejidad y por cuantos componentes están formadas. Y algo que nos resultará muy útil es provocar la acción de cargar todas las páginas quizá después de hacer un despliegue para evitar tiempos de inicialización en las primeras peticiones pero tan o más importante nos permitirá descubrir errores en los archivos tml de los componentes ¿cuantas veces te ha ocurrido que en un php, jsp, gsp, ... hasta que no se usa esa plantilla no descubres un error digamos "de compilación" (variable con nombre que no existe, atributo mal entrecomillado, ...) ? Seguramente como a mi, muchas. Los archivos de plantilla tml son XML válido con lo que si no están bien formados se nos notificará del error o si se hace referencia a una propiedad inexistente de un objeto, nuevamente ¿te ha ocurrido alguna vez tener un php, jsp o gsp que no genera html bien balanceado? Pero también si se está usando un componente que no existe, varios componentes con el mismo nombre, ... Aunque parezca que no estos tipos de errores se pueden producir con relativa facilidad en desarrollo y con mayor peligro si tenemos un

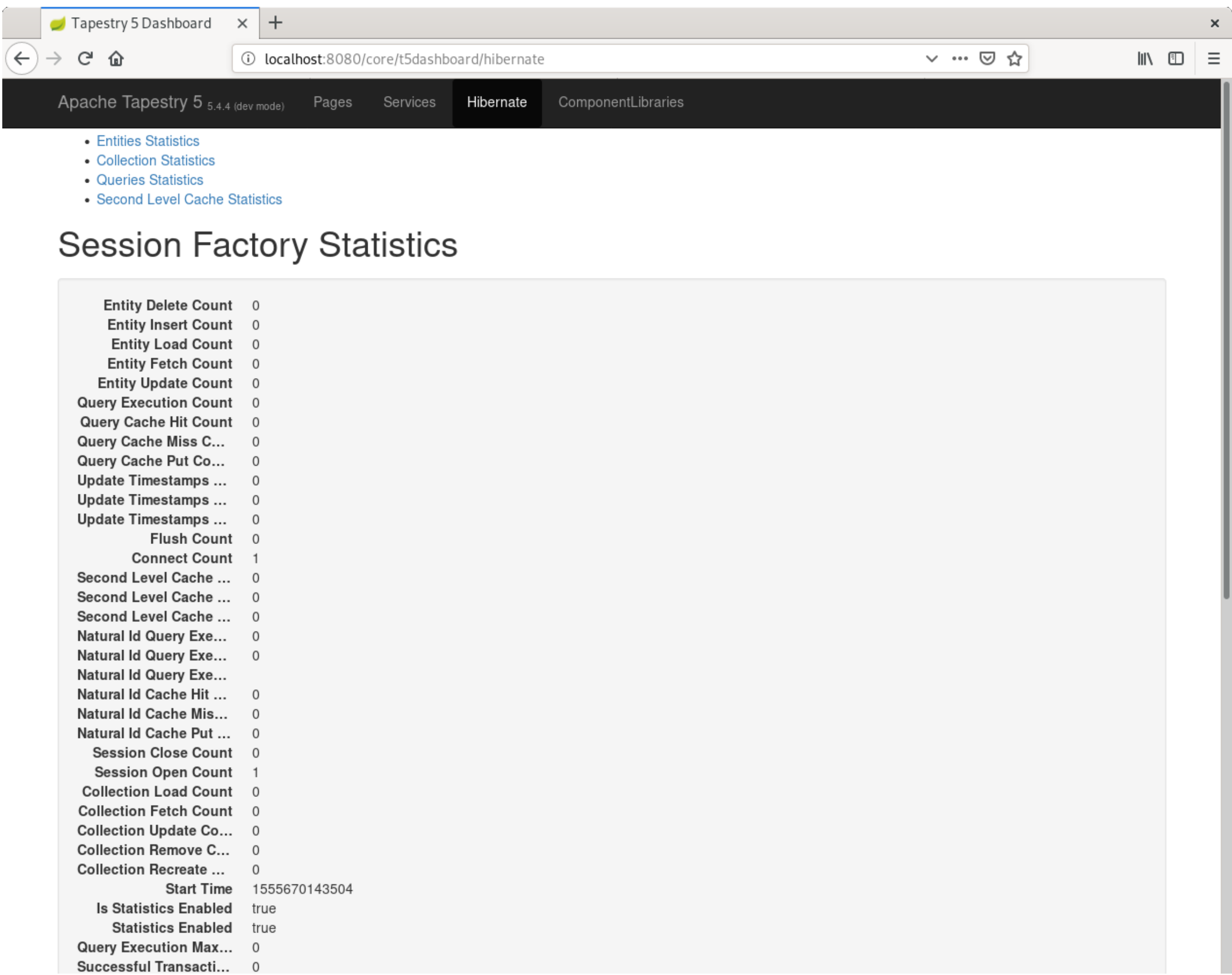
flujo de trabajo con varias ramas donde vamos mergeando los cambios de trunk a la rama que se despliega en producción y nos ocurren conflictos en los merges que tenemos que resolver manualmente con la posibilidad de cometer un error.

En otra sección también podemos ver el estado de los servicios que puede ser:

- Builtin: A servicio fundamentar que existe incluso antes de la creación del registro.
- Defined: El servicio está definido pero aún no ha sido referenciado.
- Virtual: El servicio ha sido referenciado (normalmente como inyección de otro servicio) pero aún no ha sido hecho efectivo con una instancia del servicio. El hacerse efectivo ocurre con la primera invocación en el proxy del servicio.
- Real: El servicio se ha hecho efectivo: se ha instanciado, las dependencias han sido inyectadas, se ha decorado con interceptores y el totalmente operacional.



Finalmente, en la sección HibernateStatistics podemos obtener un montón de datos que nos pueden servir para detectar situaciones anómalas en la aplicación como un gran número de sql que se lanzan en una página como podría ser en un problema de carga N+1 en una relación entre dos entidades, el estado de la cache de segundo nivel que nos permitirá optimizar las caches, la cache de queries, número de transacciones realizadas y otra gran cantidad de información.



Para que hibernate genere estadísticas indicar la propiedad hibernate.generate_statistics al contruir el bean LocalSessionFactoryBean. Y para activar la cache de segundo nivel añadir la propiedad del proveedor de cache (hibernate.cache.provider_class) y usar en las entidades la anotación @Cache, como se indica en la documentación de hibernate.

3.15 Productividad y errores de compilación

Uno de los motivos de la alta productividad es por la alta reutilización de código que se puede conseguir al usar los componentes múltiples veces en un mismo proyecto o en diferentes proyectos creando una librería de

componentes. Otra parte de la productividad es poder detectar de forma rápida errores de compilación no solo en el código Java a través del IDE sino porque con Tapestry es posible detectar errores de compilación en todas las plantillas tml que generan el html fácil y rápidamente con un botón sin tener que probar manualmente toda la funcionalidad. Esta sección muestra en detalle como detectar los errores de compilación en las vistas con este framework.

Por «errores de compilación» me refiero a ese tipo de errores que hace el código ni siquiera pueda ser interpretado correctamente por el computador, puede ser porque falta un import, un nombre de variable, propiedad o método mal puesto y que no existe... Poder detectar errores de compilación fácilmente en toda la aplicación es tremendamente útil y evitará que lleguen a producción con las consiguientes molestias para los usuarios y que posteriormente tengamos que dedicar tiempo a corregirlos cuando hemos perdido el contexto de las modificaciones hechas. También tendremos más seguridad de que no introducimos errores al hacer refactorizaciones importantes en el código. Los errores de compilación suelen ser fáciles y rápidos de corregir pero igualmente pueden impedir totalmente el uso de la aplicación. Cuando antes detectemos los errores más fácilmente los corregiremos y más productivos seremos ya que evitaremos corregirlos en un momento posterior en que costará más tiempo y esfuerzo, además de tener que realizar un nuevo despliegue con los cambios corregidos que dependiendo del tiempo que nos lleve puede suponer otro problema.

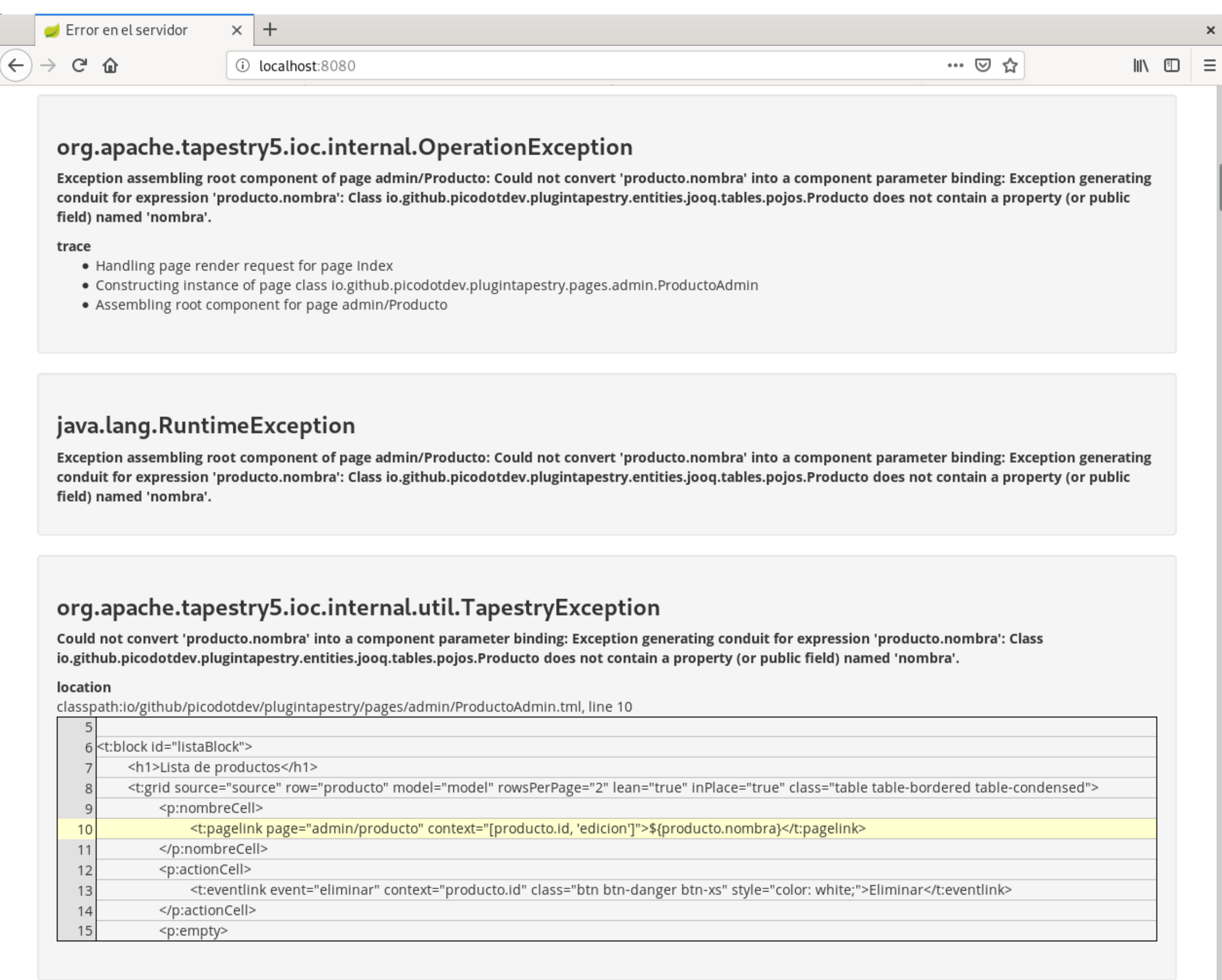
La errores de compilación no dependen de escribir pocas lineas de código o ahorrarnos pulsar unas cuantas teclas, mi experiencia con los lenguajes dinámicos como Groovy y el framework Grails es que se producen ocasionales pero constantes errores de compilación en el código Groovy y en las plantillas de vistas gsp. En parte estos errores se pueden detectar teniendo teses pero la realidad es que en pocos proyectos hay una cobertura del 100% del código sobre todo para esas partes en las que «seguro no se dan errores» o poco relevantes que no merece la pena hacerlos, tener una cobertura completa del código exige tener muchos teses y datos de prueba que los ejerciten para pasar por todas las combinaciones de condiciones y bucles, para detectar errores en las vistas también deberíamos hacer teses para ellas y esto ya no suele ser tan habitual hacerlo. Y de forma similar esto se puede extender a algunas otras combinaciones de lenguajes y frameworks web por sus características similares. Si en el proyecto solo participa una persona o el proyecto es pequeño como suele ocurrir en las pruebas de concepto con las que solemos divertirnos el número de errores no debería ser muy elevado ya que el código estará bajo control por esa persona pero cuando en un proyecto real en el que participan unos cuantos programadores haciendo continuamente modificaciones el número de errores de compilación pueden producirse, y se producirán, en producción. También hay que tener mucho cuidado en merges con conflictos, reemplazos grandes o proyectos grandes con muchos archivos ya que en uno complicado es fácil dejar un código que produce errores de compilación, en un lenguaje dinámico más por no tener la ayuda del compilador que nos avise de los mismos, también hay que resistir la tentación de hacer cambios sin probarlos de forma completa confiando en que no introduciremos errores.

Con Java y un IDE podremos detectar los errores de compilación que en un lenguaje dinámico solo observaremos en tiempo de ejecución. En Tapestry además podemos detectar los errores de compilación en las plantillas tml que generan el contenido html con un botón en la página Dashboard que ofrece incorporada Tapestry.

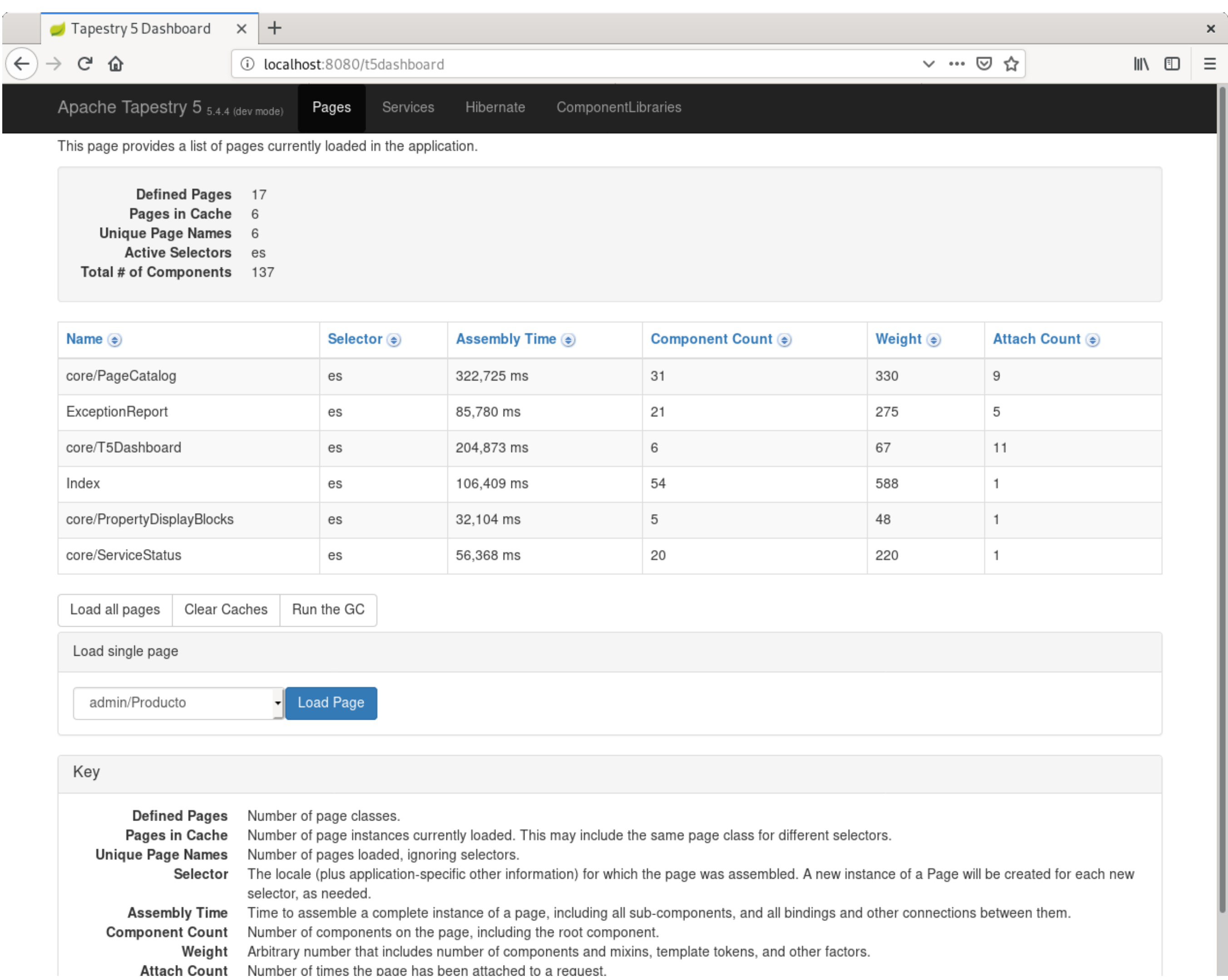
De forma intencionada introduciré un error en la página que muestra el detalle de un producto en el mantenimiento CRUD de un ejemplo. En vez de producto.nombre introduciré el error de compilación poniendo producto.nombra, nombra es una propiedad que no existe en la clase Producto, error que solo detectaremos después de crear un producto en otros frameworks al ejercitar el código pero que en Tapestry detectaremos también desde la página Dashboard. Por otra parte dado que en Tapestry las plantillas tml son xml válido si una etiqueta está mal balanceada también nos avisará.

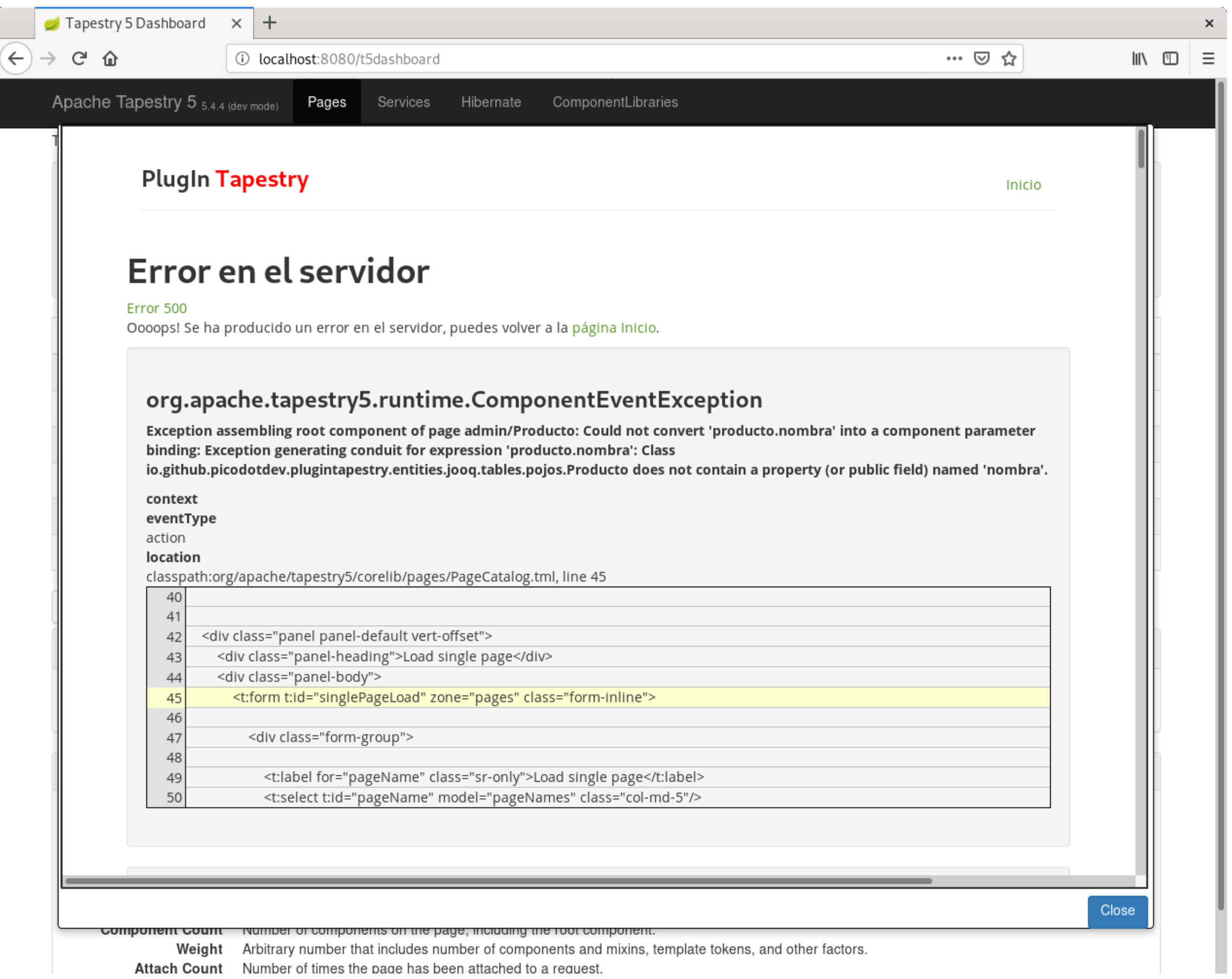
Listado 3.14: ProductoAdmin.tml

```
...
<t:form t:id="form" context="producto.id" validate="producto" clientValidation="none"
    class="well" role="form">
    <t:errors class="literal:alert alert-danger" />

    <t:delegate to="botonesEdicionBlock"/>

    <div style="margin-top: 10px;">
        <div class="form-group">
            <t:label for="nombre" />
            <div class="controls">
                <input t:type="textfield" t:id="nombre" value="producto.nombra" size="100
    " label="Nombre" />
            </div>
        </div>
...
```



Entrando a la página Dashboard y pulsando el botón Load all pages detectaremos el error sin necesidad de crear un producto. El error es el siguiente que nos indicará claramente en que página o componente se ha producido el error y una descripción bastante clara de la causa del problema.





En la imagen con el mensaje del error se puede ver de forma muy detallada cual es la causa, nos indica que el error está en la página admin/Producto y que la clase io.github.picodotdev.plugintapestry.entities.hibernate.Producto no tiene una propiedad llamada nombra, con este mensaje rápidamente nos damos cuenta del error de escritura que hemos cometido, corregirlo basta con sustituir nombra por nombre y pulsando de nuevo el botón Load all pages comprobamos que no hay más errores en esa misma página o ninguna otra de la aplicación.

Los errores en producción son un problema para los usuarios de la aplicación que no podrán trabajar normalmente y para la productividad de los desarrolladores ya que habremos perdido el contexto de los cambios causantes del fallo y nos costará más corregirlos. En caso de que se nos escape algún error la página de excepción incluso para las peticiones Ajax nos dará información detallada y un mensaje que suele ser bastante descriptivo por si solo para descubrir donde está el bug. Otro aspecto que ayuda a la productividad y que ya incorporan varios frameworks es la recarga de clases, en Tapestry es posible para los artefactos del framework (páginas, componentes y servicios, recursos i18n, imágenes, estilos css), sí, incluido código Java, con lo que tenemos las ventajas de los lenguajes de scripting y la ayuda del compilador para detectar errores inmediatamente, lo mejor de ambas opciones sin sus debilidades.

Por supuesto, no evitaremos tener otro tipo de errores en la aplicación pero al menos los de compilación si podremos detectarlos, un error habitual que se puede seguir produciendo son los NullPointerException (NPE)

pero que con las novedades introducidas en Java 8 y usando la clase Optional también deberíamos poder evitarlos. Esto es una gran ayuda tanto para la productividad como para aún mejor evitar que lleguen este tipo de errores a producción.

Capítulo 4

Contenedor de dependencias (IoC)

La estructura interna de Tapestry se basa en la inversión de control (IoC, Inversion of Control), una aproximación de diseño que permite construir un sistema a partir de muchas y pequeñas piezas fácilmente testables. Un beneficio adicional es que el IoC, divide un sistema complejo en piezas pequeñas que son más fáciles de modificar y de extender, sobrescribiéndolas o reemplazando las partes seleccionadas del sistema. El uso de IoC en Tapestry representó una evolución desde la versión 3 a la 4 y de la 4 a la 5. Tapestry 3 no usaba IoC, aunque incluía algunos mecanismos más débiles, como las extensiones que servían para el mismo propósito. Para hacer cambios importantes al comportamiento de Tapestry 3 requería heredar de clases clave y sobrescribir métodos. Tapestry 4 introdujo el uso de un contenedor IoC Hivemind. En realidad, el proyecto Hivemind fue específicamente creado para ser usado como contenedor IoC de Tapestry 4. Tapestry consiguió sus meta de extensibilidad y configuración debido a la flexibilidad de Hivemind. Tapestry 5 se basa en esto, reemplazando Hivemind por un nuevo contenedor específicamente construido para Tapestry 5, diseñado para ser más fácilmente usado, ser más expresivo y de alto rendimiento. T5 IoC es considerado un Hivemind simplificado y mejorado y que también puede ser usado separadamente del resto de Tapestry.

¿Por que no Spring?

Spring es el proyecto de IoC más exitoso. Spring combina un buen contenedor IoC, integrado con soporte AspectJ y una larga lista de librerías sobre el contenedor. Spring es un contenedor excelente pero no tiene algunas características necesarias según las necesidades de Tapestry:

- A pesar de que los beans de Spring permiten ser interceptados lo hacen en la forma de un nuevo bean, dejando el bean sin interceptar visible (y posiblemente mal usado). El IoC de Tapestry envuelve el servicio dentro de interceptores previniendo accesos sin interceptar a la implementación del servicio.
- Spring tiene un esquema simple de configuración map/list/value pero no es distribuido, es parte de una sola definición de beans. El IoC de T5 permite que la configuración de un servicio sea construida desde múltiples módulos. Esto es muy importante para una extensibilidad fácil del framework sin necesidad de configuración (simplemente dejar el módulo en el classpath y todo se enlaza entre sí).

¿Por que no Hivemind?

La dificultad de manejar los calendarios de dos frameworks complejos demostró ser un problema. El uso de Hivemind estuvo relacionado con uno de las críticas de T4: el tiempo de arranque. El tiempo que tomaba parsear y organizar todo el XML tomaba varios segundos del tiempo de arranque. Crear un contenedor IoC simplificado que no estuviese dirigido por XML alivió estos problemas. Con la llegada de nuevas tecnologías (en particular con las anotaciones de JDK 1.5 y la generación de clases mediante Javassist) algunos de los preceptos de HiveMind se debilitaron. Eso es para decir, que ese XML de Hivemind (como en Spring) era una forma incómoda de describir unas pocas operaciones Java: instanciar clases e invocar métodos en esas clases (para inyectar las dependencias en esas instancias). El concepto central del IoC de T5 es eliminar el XML y construir un sistema equivalente alrededor de objetos simples y métodos. El IoC de Tapestry también representa varias simplificaciones de Hivemind tomando las lecciones aprendidas en él.

¿Por que no Guice?

Google Guice es relativamente nuevo en el espacio de IoC. Guice y T5 IoC están muy cercanos y en realidad T5 IoC toma prestados expresamente varias grandes e innovadoras ideas de Guice. Guice no solo abandona el XML sino también el concepto de id de servicio... para la inyección, los servicios son emparejados por tipo y tal vez filtrados en base a anotaciones. Aún así a Guice todavía le faltan algunas ideas básicas necesitadas como IoC de T5. No existe el concepto de configuración o algo similar y hay limitaciones en la inyección basada en el ámbito (un valor de ámbito de petición no puede ser inyectado en un servicio de ámbito global).

4.1 Objetivos

Como en T5 en general, el objetivo de Tapestry IoC es conseguir mayor simplicidad, más poder y evitar el XML. Los contenedores IoC existentes como Hivemind y Spring típicamente contienen grandes cantidades de configuración XML que describen como y cuando instanciar un JavaBean particular y como proporcionar a un Bean sus dependencias (ya sea por inyección en el constructor o mediante inyección de propiedades). Otro XML es usado para enganchar objetos en alguna forma de ciclo de vida... típicamente métodos de llamadas de vuelta invocadas cuando el objeto es instanciado y configurado o cuando va a ser descartado.

El concepto central de Tapestry IoC es que el propio lenguaje Java es la forma más fácil y breve para describir la creación de un objeto e invocación de un método. Cualquier aproximación en XML es en última instancia más verboso y difícil de manejar. Como muestran los ejemplos, una pequeña cantidad de código Java y un puñado de convenciones de nombres y anotaciones es de lejos más simple y fácil que un gran trozo de XML. Además, cambiar de XML a código Java anima a hacer pruebas, puedes hacer pruebas unitarias el los métodos de construcción de servicios de tus clases de módulo, ya que en realidad no puedes probar unitariamente un descriptor XML. Los módulos de Tapestry IoC son fácilmente empaquetados en archivos JAR que para usarlos no requieren configuración, simplemente incluirlos en el classpath.

Otro objetivo es afinidad con el desarrollador. El framework IoC de Tapestry está diseñado para ser fácilmente usado y entendido. Es más, cuando las cosas van mal, intenta activamente ayudarte mediante comprobaciones

entendibles y mensajes de error compuestos con cuidado. Y aún más, todos los objetos visibles por el usuario implementan un método toString() razonable para ayudarte a entender que está yendo mal cuando inevitablemente intentes averiguar cosas en el depurador.

En términos de construcción de servicios usando Tapestry IoC el objetivo es ligereza. En el desarrollo de software estamos intentando crear sistemas complejos de piezas simples pero la restricción es balancear la necesidad de probar código existente y mantener código existente. Demasiado a menudo en el mundo del desarrollo de software necesitas añadir una funcionalidad que supera a todo lo demás, y las pruebas y el mantenimiento es aplazado hasta que es demasiado tarde. Los contenedores IoC en general, y T5 IoC específicamente, existen para resolver este problema proporcionando las bases de necesidad de rapidez y funcionalidad contra la necesidad de probar nueva funcionalidad y mantenimiento de funcionalidad existente. Los contenedores IoC proporcionan los medios para dividir sistemas grandes, complejos y monolíticos en piezas ligeras, pequeñas y probables.

Cuando se construyen registros de servicios, la ligereza se refiere a una división adecuada de responsabilidad, separación de conceptos y limitar las dependencias entre diferentes partes del sistema. Este estilo es habitualmente llamado Ley de Demeter. Usando un contenedor IoC hace fácil seguir esta aproximación, dado que una preocupación que es la responsabilidad de instanciar a otros es gestionado por el contenedor. Con esta preocupación del ciclo de vida resuelto se hace más fácil reducir complejos bloques de código en servicios pequeños, testables y reusables.

Ligereza (ligth) significa:

- Interfaces pequeñas de dos o tres métodos.
- Métodos pequeños, con dos o tres parámetros (dado que las dependencias son inyectadas detrás de la escena en vez de pasado al método).
- Comunicación anónima vía eventos, en vez de invocaciones explícitas de métodos. La implementación del servicio puede implementar una interfaz de evento.

4.2 Terminología

La unidad básica de Tapestry IoC es un servicio. Un servicio consisten en una interfaz y una implementación. La interfaz del servicio es una interfaz ordinaria de Java. La implementación del servicio es un objeto Java que implementa la interfaz del servicio. A menudo habrá solo un servicio por interfaz, pero en algunas situaciones, puede haber diferentes servicios e implementaciones de servicios todas compartiendo la misma interfaz. Los servicios son identificados por un id único. Típicamente, un id de servicio coincide con un nombre no cualificado de la interfaz del servicio, pero esto es simplemente una convención. La dirección de evolución de Tapestry IoC es eliminar eventualmente los id de los servicios y trabajar únicamente en términos de interfaces de servicios y anotaciones de etiquetado.

Un módulo es definido en una clase de módulo, una clase específica que contiene una mezcla de métodos estáticos y de instancia usados para definir, decorar o contribuir a configuraciones de servicio. Los métodos de

la clase del módulo define los servicios proporcionados por el módulo y los mismos métodos son responsables de instanciar las implementaciones de los servicios. Los métodos que definen y construyen servicios son denominados métodos constructores.

El registro es una vista exterior de los módulos y servicios. A partir del registro, es posible obtener un servicio, mediante su id único o mediante su interfaz de servicio. El acceso por id único es insensible a mayúsculas. Los servicios pueden ser decorados por métodos de decoración. Estos métodos crean objetos interceptores que envuelven las implementaciones de los servicios, añadiendo comportamiento como trazas, seguridad o transaccionalidad. Las implementaciones de los interceptores implementan la misma interfaz de servicio que el servicio que interceptan. Un servicio puede tener una configuración que puede ser un mapa, una colección o una lista ordenada. El servicio define el tipo de objeto permitido a contribuir. La configuración es construida a partir de las contribuciones proporcionadas por uno o más módulos y los métodos de contribución de servicio son invocados por Tapestry para contribuir objetos a configuraciones.

Los servicios son instanciados cuando se necesitan. En este caso, necesitado se traduce cuando un método del servicio es invocado. Un servicio se representa (al mundo exterior o a otros servicios) como un proxy que implementa la interfaz del servicio. La primera vez que un método es invocado en un proxy, el servicio completo (que consistente en el servicio y los interceptores) es construido. Esto ocurre de forma segura para los threads. La instanciación justo a tiempo permite una red de servicios más complejos y mejora los tiempos de inicio. Instanciar un servicio, inyectar dependencias y decorar el servicio son todo partes de la realización del servicio, el punto en que el servicio pasa de virtual (solo un proxy) a real (completamente instanciado y listo para operar). Los servicios definen un ámbito que controla cuando el servicio se construye así como su visibilidad. El ámbito por defecto es una única instancia (singleton), que significa que una instancia global se crear cuando se necesita. Otros ámbitos permiten a las implementaciones de los servicios ser asociadas al thread actual (esto es, a la petición actual en una aplicación de servlet).

Las dependencias son otros servicios (u otros objetos) que son necesarios por una implementación de servicio. Estas dependencias pueden ser inyectadas en el método constructor de servicio y proporcionado desde ahí a las implementaciones de los servicios. También pueden referirse como colaboradores, especialmente en el contexto de las pruebas unitarias. El punto de inyección es una propiedad, parámetro de método o parámetro de constructor que recibe el valor a inyectar. El tipo del servicio (y otras dependencias) es determinado por el tipo de la propiedad o parámetro. A menudo, las anotaciones identifican que es inyectado o en caso de la inyección de propiedades que una inyección es requerida.

4.3 Inversión de control (IoC)

La inversión del control se refiere al hecho de que el contenedor, esto es el registro de Tapestry IoC, instancia tus clases y decide cuando. La inyección de dependencias es como a un servicio se le proporcionan otros servicios que necesita para operar. Por ejemplo, a un servicio objeto de acceso a datos (DAO) puede inyectarsele un servicio ConnectionPool que proporcione las conexiones a la base de datos.

En Tapestry, la inyección ocurre a través de los constructores, a través de métodos constructores de servicio o mediante la inyección directa de propiedades. Tapestry prefiere la inyección en el constructor, dado que esto enfatiza que las dependencias debería ser almacenadas en variables finales. Este es el mejor camino para garantizar hilos seguros. En cualquier caso, la inyección simplemente ocurre. Tapestry encuentra el constructor

de tu clase y analiza los parámetros para determinar que pasarles. En algunos casos, solamente se usa el tipo para encontrar una coincidencia, en otros se usan además las anotaciones sobre los parámetros. También busca en las propiedades la clase de la implementación del servicio para identificar que valores deberían ser inyectados en ellos.

Módulos de Tapesty IoC

Puedes informar a Tapestry acerca de tus servicios y contribuciones proporcionando una clase de módulo. La clase de módulo es una clase Java normal. Un sistema de anotaciones y convenciones de nombres permiten a Tapestry determinar que servicios son proporcionados por el módulo. Una clase de módulo existe por las siguientes razones:

- Para asociar las interfaces a implementaciones.
- Para contribuir configuraciones a los servicios.
- Para decorar servicios proporcionando interceptores alrededor de ellos.
- Para proporcionar código explícito para construir el servicio.
- Para establecer un marcador por defecto para todos los servicios en el módulo

Todos los métodos públicos de una clase de módulo debe ser significativos para Tapestry (ser de una de las categorías anteriores). Cualquier método público extra produce excepciones en el inicio.

Métodos de construcción de servicio

Los métodos de construcción de servicio son la forma original de definir un servicio y proporcionar la lógica para construirlo, sin embargo, estos es más comúnmente (y simplemente) llevado a cabo usando el método bind(), aún así hay algunos casos en que los métodos de construcción de servicios son útiles. Los métodos de construcción de servicio son métodos públicos y a menudo son estáticos. Este es un ejemplo trivial:

```
1 package io.github.picodotdev.plugintapestry.services;

  public class AppModule {

    public static Indexador build() {
6     return new IndexadorImpl();
    }
  }
```

Cualquier método publico (estático o de instancia) cuyo nombre comience con build es un método constructor de servicio que define un servicio en el módulo. Aquí estamos definiendo un servicio que implementa la interfaz Indexador (presumiblemente también en el paquete io.github.picodotdev.plugintapestry.services). Cada servicio tienen un id único usado para identificarlo en el registro de servicios (el registro es la suma combinada de todos los servicios de todos los módulos). Si no proporcionas un id de servicio explícito, como en este ejemplo, el id del servicio es obtenido del tipo del retorno, este servicio tiene un id Indexador. Puedes dar al servicio un id explícito añadiéndolo al nombre del método: buildIndexador(). Esto es útil cuando no quieres que el id del servicio coincida con el nombre de la interfaz (por ejemplo, cuando tienes diferentes servicios que implementar la misma interfaz) o cuando necesitas evitar colisiones de nombre en los nombres de los métodos (Java solo permite un único método con un nombre y conjunto de parámetros, independiente de si el tipo de retorno es diferente, de modo que si tienes dos métodos de constructor de servicio diferentes que toman los mismos parámetros, deberías darles un id de forma explicita en el nombre del método). Tapestry IoC es insensible a mayúsculas, por lo que nos podemos referir al servicio como indexador, INDEXADOR o cualquier otra variación. Los ids de servicio deben ser únicos, si otro módulo contribuye un servicio con el id Indexador se producirá una excepción en tiempo de ejecución cuando el registro sea creado.

Extenderemos este ejemplo añadiendo métodos de construcción de servicio adicionales o mostrando como inyectar dependencias.

Autoconstruyendo servicios

Una forma alternativa, y usualmente preferida, de definir un servicio es mediante el método de módulo bind(). El ejemplo anterior puede ser reescrito como:

```
1  package io.github.picodotdev.plugintapestry.services;
2
   import org.apache.tapestry5.ioc.ServiceBinder;

   public class AppModule {

7    public static void bind(ServiceBinder binder) {
       binder.bind(Indexador.class, IndexadorImpl.class);
     }
   }
```

Generalmente, deberías hacer un bind y autobuild (inyección automática de dependencias) de tus servicios. Las únicas excepciones son cuando:

- Deseas hacer algo más que solamente instanciar la clase; por ejemplo, registrar la clase como un escuchador de eventos de otro servicio.
- No hay implementación de la clase; en algunos casos, puedes crear la implementación al vuelo usando proxys dinámicos o con generación de bytecode.

El método bind() debe ser estático y se lanzará una excepción si el método existe pero es de instancia.

Cacheo de servicios

Ocasionalmente te encontrarás con la necesidad de inyectar el mismo servicio repetidamente en los constructores de servicio o en los decoradores de servicio y puede ser repetitivo (esto ocurre menos a menudo desde la introducción del autobuilding de servicios). Menos código es mejor código, a modo de alternativa puedes definir un constructor para el módulo que acepta parámetros. Esto da la oportunidad de almacenar servicios comunes como propiedades de instancia para ser usados más tarde en los métodos constructores de servicio.

```
public class AppModule {

  private final JobScheduler scheduler;
  private final FileSystem fileSystem;

  public AppModule(JobScheduler scheduler, FileSystem fileSystem) {
    this.scheduler = scheduler;
    this.fileSystem = fileSystem;
  }

  public Indexador buildIndexador() {
    IndexadorImpl indexador = new IndexadorImpl(fileSystem);
    scheduler.scheduleDailyJob(indexador);
    return indexador;
  }
}
```

Fíjate que hemos cambiado de métodos estáticos a métodos de instancia. Dado que los métodos constructores no son estáticos, la clase del módulo será instanciada de forma que los métodos puedan ser invocados. El constructor recibe dos dependencias, que son almacenadas como propiedades de instancia para ser usadas más tarde en los métodos constructores de servicio como buildIndexador(). Esto es así si quieres, todos los métodos de tu módulo pueden ser estáticos si deseas. Es usado cuando tienes varias dependencias comunes y deseas evitar definir esas dependencias como parámetros en múltiples métodos.

Tapestry IoC automáticamente resuelve los tipos de los parámetros a los correspondientes servicios que implementan esos tipos. Cuando hay más de un servicio que implementa la interfaz del servicio, se producirá un error (con anotaciones adicionales y configuración puede ser inyectado el servicio correcto).

Nota que los campos son finales esto es para que los valores estén disponibles en varios threads. Tapestry IoC es thread-safe de modo que no deberás pensar en esto.

Notas de la implementación de clases de módulo

Las clases de módulo están diseñadas para ser muy simples de implementar. Mantén los métodos muy simples. Usa inyección de parámetros para obtener las dependencias que necesites. Ten cuidado con la herencia, Tapestry verá todos los métodos públicos, incluso aquellos heredados por la clase base. Por convención los

nombres de las clases de módulo terminan en Module y son finales. No necesitas que los métodos sean estáticos, el uso de métodos estáticos solo es absolutamente necesario en pocos casos donde el constructor para un módulo es dependiente de contribuciones del mismo módulo (esto crea el problema de la gallina y el huevo que se resuelve con métodos estáticos, el módulo necesita los servicios y estos las contribuciones, las contribuciones necesitan una instancia del módulo).

4.4 Clase contra servicio

Un servicio de Tapestry es más que solo una clase. Primero, es una combinación de una interfaz que define las operaciones del servicio y una clase de implementación que implementa esa interfaz.

¿Por que esta división? Tener una interfaz del servicio es lo que permite a Tapestry crear proxys y realizar otras operaciones. Además es una buena práctica codificar sobre interfaces en vez de implementaciones. Te sorprenderás de los tipos de cosas que puedes hacer sustituyendo una implementación por otra.

Tapestry es también consciente de que un servicio tendrá dependencias sobre otros servicios o otras necesidades como acceso a Loggers. Tapestry tiene soporte para proporcionar una configuración que puede ser proporcionada cuando se realizan.

Ciclo de vida de los servicios

Cada servicio tiene un ciclo de vida específico.

- Definido: el servicio tiene una definición (en algún módulo) pero no ha sido referenciado.
- Virtual: el servicio ha sido referenciado, de modo que hay un proxy para la clase.
- Realizado: un método del servicio ha sido invocado, de modo que la implementación del servicio ha sido instanciada y cualquier decorador ha sido aplicado.
- Apagado: el registro entero ha sido apagado y con él todos los proxys han sido deshabilitados.

Cuando el registro es creado por primera vez, todos los módulos son revisados y las definiciones para todos los servicios son creadas. Los servicios serán referenciados accediendo a ellos usando el registro o como dependencias de otros servicios realizados. Tapestry IoC espera hasta el último momento posible para realizar el servicio que es cuando un método del servicio es invocado. Tapestry es thread-safe de modo que aún en un entorno altamente concurrente (como un servidor de aplicaciones o contenedor de servlets) las cosas simplemente funcionan.

4.5 Inyección

El contenedor de IoC de Tapestry usa inyección principalmente a través de constructores y mediante parámetros en los métodos constructores de servicio.

Inyección en las clases de componente

Para los componentes, sin embargo, lo hace de forma completamente distinta, la inyección la realiza directamente en propiedades del componente. La notación @Inject es usada para identificar las propiedades que contendrán los servicios inyectados y otros recursos. Se permiten dos tipos de inyecciones:

- Inyección por defecto, donde Tapestry determina el objeto a inyectar en la propiedad basándose en su tipo.
- Inyección explícita, cuando se especifica un servicio particular.

En ambos casos, la propiedad es transformada en un valor de solo lectura e intentar actualizarla resultará en una excepción en tiempo de ejecución. Como en otras partes, esta transformación ocurre en tiempo de ejecución (que es muy importante para ser testable). Además, hay unos pocos casos especiales de inyección que son provocados por algunos tipos especiales o anotaciones adicionales en la propiedad además de la anotación @Inject.

Inyección de bloques

Para una propiedad de tipo Block, el valor a inyectar por la anotación Inject es el id del elemento <t:block> en la plantilla del componente. Normalmente, el id del bloque es determinado por el nombre de la propiedad (después de eliminar cualquier caracter _ y $ al principio).

```
1 @Inject
  private Block foo;
```

Aunque no es lo más apropiado, se puede proporcional la anotación @Id:

```
1 @Inject
  @Id("bar")
3 private Block barBlock;
```

La primera anotación inyectaría el bloque con id foo de la plantilla (como siempre, insensible a mayúsculas). La segunda inyección inyectará el bloque con el id bar.

Inyección de recursos

Para un conjunto particular de tipos de propiedades, Tapestry inyectará un recurso relacionado con el componente, como su Locale. Un ejemplo muy común ocurre cuando un componente necesita acceso a sus recursos. El componente puede definir una propiedad del tipo apropiado y usar la anotación @Inject:

```
@Inject
private ComponentResources resources;
```

Tapestry usa el tipo de la propiedad, ComponentResources, para determinar que inyectar en esta propiedad. Los siguientes tipos está soportados en la inyección de recursos:

- java.lang.String: El id completo, que incorpora el nombre de la clase completo de la página que lo contiene y los id anidados dentro de la página.
- java.util.Locale: El locale para el componente (todos los componentes dentro de una página usan el mismo locale).
- org.slf4j.Logger: Un logger configurado para el componente, basado en el nombre de la clase.
- org.apache.tapestry5.ComponentResources: Los recursos para el componente, usado a menudo para generar enlaces relacionados con el componente.
- org.apache.tapestry5.ioc.Messages: El catálogo de mensajes para el componente a partir de los cuales se pueden generar mensajes localizados.

Inyección de assets

Cuando la anotación @Path también está presente, entonces el valor inyectado será un asset localizado (relativo al componente). Los símbolos en el valor de la anotación son expandidos.

```
@Inject
@Path("context:images/banner.png")
private Asset banner;
```

Inyección de servicios

A continuación se inyecta el servicio personalizado ProductoDAO, cualquier servicio nuestro o propio de Tapestry puede inyectarse de la misma forma.

```
@Inject
private ProductoDAO dao;
```

Tapestry proporciona un largo número de servicios en los siguientes paquetes:

- Core Services

- AJAX Services
- Assets Services
- Dynamic Component Services
- JavaScript Services
- Link Transformation Services
- Message Services
- Component Metadata Services
- Page Loading Services
- Security Services
- Template Services
- Class Transformation Services
- Tapestry IOC Services
- Tapestry IOC Cron
- Services Kaptcha Services
- File Upload Services

Inyección explícita de servicio

Aquí, se solicita inyectar un objeto específico. La anotación @Service se usa para identificar el nombre del servicio.

```
@Inject
@Service("Request")
private Request request;
```

Esto generalmente no es necesario, deberías poder identificar el servicio a inyectar con solo el tipo, no por su id explícito. Los ids explícitos tienen la desventaja de no ser seguros en los refactors: esto no pasará en el servicio Request pero si en tus propios servicios... si renombras la interfaz del servicio y renombras el id para que coincida, tus inyecciones que hagan uso de un id explícito se romperán.

Inyección por defecto

Cuando el tipo y/o otras anotaciones no son suficientes para identificar el objeto o servicio a inyectar, Tapestry se retira con dos pasos restantes. Asume que el tipo de la propiedad será usado para identificar un servicio, por la interfaz del servicio.

Primero, se consulta el proveedor del objeto creado por el servicio Alias. Este objeto proveedor es usado para desambiguar inyecciones cuando hay más de un servicio que implementa la misma interfaz del servicio.

Segundo, se busca un servicio único que implemente la interfaz. Esto fallará si no hay servicios que implementen la interfaz o si hay más de uno. En el último caso, puedes eliminar la desambiguación con una contribución al servicio Alias o explícitamente identificando el servicio con la anotación @Service.

Servicios mutuamente dependientes

Uno de los beneficios de la aproximación basada en proxys de Tapestry IoC con la instanciación justo en el momento es el soporte automático para servicios mutuamente dependientes. Por ejemplo, supón que el servicio Indexer y el FileSystem necesitan hablar directamente uno con el otro. Normalmente, esto causaría el problema del huevo o la gallina ¿cual crear primero?. Con Tapestry IoC, esto no es considerado un problema especial:

```
public static Indexer buildIndexer(JobScheduler scheduler, FileSystem fileSystem) {
  IndexerImpl indexer = new IndexerImpl(fileSystem);
  scheduler.scheduleDailyJob(indexer);
  return indexer;
}

public static FileSystem buildFileSystem(Indexer indexer) {
  return new FileSystemImpl(indexer);
}
```

Aquí, Indexer y FileSystem son mutuamente dependientes. Eventualmente, uno o el otro será creado... digamos que es FileSystem. El método constructor buildFileSystem() será invocado y un proxy de Indexer será pasado. Dentro del constructor de FileSystemImpl (o en algún momento después) un método del servicio Indexer será invocado en cuyo punto el método buildIndexer es invocado. Aún así todavía recibe el proxy del servicio FileSystem.

Si el orden es invertido, de modo que Indexer es construido antes que FileSystem todo funciona exactamente igual. Esta aproximación puede ser muy potente. Por ejemplo, puede ser usada para partir código monolítico no testable en dos mitades mutuamente dependientes, cada una de las cuales puede ser probada individualmente.

La excepción a esta regla es un servicio que depende de si mismo durante la construcción. Esto puede ocurrir cuando (indirectamente, a través de otros servicios) al construir el servicio intenta invocar un método en el servicio que se está construyendo, cuando el constructor de la implementación del servicio invoca métodos en servicios dependientes que le son pasados o cuando el constructor del servicio mismo hace lo mismo. Este es un caso raro.

4.5.1 Configuración en Tapestry IoC

Los servicios de Tapestry, tanto los proporcionados por Tapestry y los escritos por ti, son configurados usando código Java, no XML. Uno de los conceptos clave en Tapestry IoC es la configuración distribuida. La parte distribuida se refiere al hecho de que cualquier módulo puede configurar un servicio. La configuración distribuida es la característica clave de Tapestry IoC que soporta la extensibilidad y modularidad. Los módulos configuran un servicio contribuyendo configuraciones al servicio.

Veamos un ejemplo. Digamos que has escrito un puñado de diferentes servicios, cada uno de los cuales hace algo específico para un tipo particular de archivo (identificado por la extensión del archivo) y que cada uno implementa la misma interfaz que llamaremos FileService. Y ahora digamos que necesitas un servicio central que seleccione aquella implementación de FileService basada en una extensión. Empezarás proporcionando el método constructor del servicio:

```
1 public static FileServiceDispatcher buildFileServicerDispatcher(Map<String,FileService>
    contributions) {
  return new FileServiceDispatcherImpl(contributions);
}
```

Para proporcionar un valor para el parámetro de contribución, Tapestry recolecta las contribuciones de los métodos de contribución de servicio. Asegurará que las claves y los valores corresponden con los tipos genéricos mostrados (String para la clave, FileService para el valor). El mapa se construirá y será pasado al método de construcción de servicio y de ahí al constructor del FileServiceDispatcherImpl. De modo que ¿de donde vienen los valores? De los métodos de contribución. Los métodos de contribución de servicio son aquellos que comienzan con contribute:

Listado 4.1: AppModule.java

```
1 public static void contributeFileServiceDispatcher(MappedConfiguration<String,FileService
    > configuration) {
2   configuration.add("txt", new TextFileService());
  configuration.add("pdf", new PDFFileService());
}
```

O en vez de instanciar esos servicios nosotros mismos podemos inyectarlos:

Listado 4.2: AppModule.java

```
1 public static void contributeFileServicerDispatcher(MappedConfiguration<String,
    FileService> configuration, @InjectService("TextFileService") FileService
    textFileService, @InjectService("PDFFileService") FileService pdfFileService) {
  configuration.add("txt", textFileService);
  configuration.add("pdf", pdfFileService);
}
```

La extensibilidad viene por el hecho de que múltiples módulos pueden contribuir a la configuración del mismo servicio:

Listado 4.3: OffimaticModule.java

```
1 public static void contributeFileServiceDispatcher(MappedConfiguration<String,FileService
    > configuration) {
  configuration.add("doc", new WordFileService());
  configuration.add("ppt", new PowerPointFileService());
}
```

Ahora el constructor de FileServiceDispatcher obtiene un Map con al menos cuatro entradas en él. Dado que Tapestry es altamente dinámico (busca los archivos de manifiesto en los JAR para identificar las clases de los módulos), el servicio FileServiceDispatcher puede estar en un módulo y otros módulos contribuirle, como en el que contribuye con los archivos de ofimáticos. Sin hacer ningún cambio al servicio FileServiceDispatcher o su clase de módulo, los nuevos servicios se conectan a la solución global simplemente por tener su JAR en el classpath.

Convenciones de nombre contra anotaciones

Si prefieres usar anotaciones en vez de convenciones de nombre puedes usar la anotación @Contribute. El valor de la anotación es el tipo de servicio al que contribuir. Las principales razones para usar @Contribute y anotaciones de marcado son:

- No hay unión entre el nombre de método de contribución y el id del servicio lo que es mucho más seguro al refactorizar: si cambias el nombre de la interfaz del servicio o el id del servicio tu método seguirá siéndose invocado.
- Hace mucho más fácil para una sobrescritura del servicio obtener la configuración intencionada para el servicio original.

El siguiente ejemplo es una alternativa basada en anotaciones para el método de contribución anterior.

```
1 @Contribute(FileServiceDispatcher.class)
public static void nombreDeMetodoArbitrario(MappedConfiguration<String,FileService>
    configuration) {
  configuration.add("doc", new WordFileService());
  configuration.add("ppt", new PowerPointFileService());
}
```

Si tienes varias implementaciones de la interfaz del servicio debes desambiguar los servicios. Para este propósito las anotaciones de marcado deberían ser colocadas en el método contribuidor.

```java
@Contribute(FileServiceDispatcher.class)
@Red
@Blue
public static void nombreDeMetodoArbitrario(MappedConfiguration<String,FileService>
    configuration) {
  configuration.add("doc", new WordFileService());
  configuration.add("ppt", new PowerPointFileService());
}
```

En este ejemplo, el método solo será invocado cuando se construya una configuración de servicio donde el servicio mismo tenga las dos anotaciones Red y Blue. Tapestry conoce que anotaciones son anotaciones de marcado y que anotaciones de marcado aplican al servicio mediante la anotación @Marker en la implementación del servicio.

Si la anotación especial @Local está presente entonces la contribución es realizada solo para la configuración de un servicio que sea construido en el mismo módulo. Nota que es posible para el mismo método de contribución ser invocado para contribuir a la configuración del múltiples servicios diferentes.

```java
@Contribute(FileServiceDispatcher.class)
@Local
public static void nombreDeMetodoArbitrario(MappedConfiguration<String,FileService>
    configuration) {
  configuration.add("doc", new WordFileService());
  configuration.add("ppt", new PowerPointFileService());
}
```

Tipos de configuración

Hay tres tipos estilos diferentes de configuraciones (con sus correspondientes contribuciones):

- Colecciones no ordenadas: las contribuciones son simplemente añadidas y el orden no es importante.
- Listas ordenadas: las contribuciones son proporcionadas como una lista ordenada. Las contribuciones deben establecer el orden dando a cada objeto contribuido un id único, estableciendo dependencias entre los valores siguientes y anteriores.
- Mapas: las contribuciones proporcionan claves únicas y correspondientes valores.

Colecciones no ordenadas

Un método constructor de servicio puede recolectar una lista no ordenada de valores definiendo un parámetro de tipo java.util.Collection. Es más, deberías parametrizar el tipo de la colección. Tapestry identificará el tipo

parametrizado y asegurará que todas las contribuciones coincide. Una cosa a recordar es que el orden en que las contribuciones ocurren es indeterminado. Habrá un posible número grande de módulos cada uno teniendo cero o más métodos que contribuyen al servicio. El orden en que estos métodos son invocados es desconocido. Por ejemplo, este es un servicio que necesita algunos objetos Runnable. No importa en que orden los objetos Runnable son ejecutados.

```java
public static Runnable buildStartup(final Collection<Runnable> configuration) {
  return new Runnable() {
    public void run() {
      for (Runnable contribution : configuration)
        contribution.run();
    }
  };
}
```

Aquí no necesitamos ni siquiera una clase separada para la implementación, usamos una clase anónima para la implementación. El punto es que la configuración es proporcionada al método constructor que se lo pasa a la implementación del servicio. En el lado de la contribución, un método de contribución ve un objeto Configuration:

```java
public static void contributeStartup(Configuration<Runnable> configuration) {
  configuration.add(new JMSStartup());
  configuration.add(new FileSystemStartup());
}
```

La interfaz Configuration define solo un método: add(). Esto es así de forma intencionada: la única cosa que puedes hacer es añadir nuevos elementos. Si pasásemos una colección podrías estar tentado de comprobar los valores o eliminarlos. Por legibilidad se ha parametrizado el parámetro de configuración, restringiéndolo a instancias de java.lang.Runnable. Esto es opcional pero a menudo útil. En cualquier caso, intentar contribuir un objeto que no extiende o implementa el tipo (Runnable) resultará en un advertencia de tiempo de ejecución (y el valor será ignorado). Tapestry soporta solo estos tipos simples de parametrización, los generics de Java soportan una forma más amplia (wildcards) que Tapestry no entiende.

Listas ordenadas

Las listas ordenadas son mucho mas comunes. Con una lista ordenada, las contribuciones son almacenadas en un orden apropiado para ser proporcionado al método constructor del servicio. De nuevo, el orden en el que los métodos de contribución de servicio son invocados es desconocido. Por lo tanto, el orden en el que los objetos son añadidos a la configuración no es conocido. En vez de ello, se fuerza un orden a los elementos después de que todas las contribuciones se ha realizado. Como con los servicios decoradores, establecemos el orden dando a cada objeto contribuido un id único e identificando por id cuales elementos deben preceder en la lista y

cuales estar a continuación. De modo que podemos cambiar nuestro servicio para requerir un orden específico de arranque del siguiente modo:

```
public static Runnable buildStartup(final List<Runnable> configuration) {
  return new Runnable() {
    public void run() {
      for (Runnable contribution : configuration)
        contribution.run();
    }
  };
}
```

Nota que el método constructor de servicio está protegido de los detalles de como los elementos están ordenados. No tiene que conocer nada acerca de id y de requisitos pre y post. Usando un parámetro de tipo List habremos recogido la información de ordenación. Para los métodos de contribución de servicio debemos proporcionar un parámetro de tipo OrderedConfiguration:

```
public static void contributeStartup(OrderedConfiguration<Runnable> configuration) {
  configuration.add("JMS", new JMSStartup());
  configuration.add("AppModuleJoinPoint", null);
  configuration.add("FileSystem", new FileSystemStartup(), "after:CacheSetup");
}
```

A menudo, no te preocupas del orden como en el primer add. El algoritmo de ordenación encontrará el punto para el objeto basándose en las restricciones de los objetos contribuidos. Para la contribución FileSystem se especifica una restricción indicando que el FileSystem debería ser ordenado después de alguna otra contribución llamada CacheSetup. Puede especificarse cualquier número de restricciones de orden (el método add acepta un número variable de argumentos). El objeto de la configuración pasado puede ser nulo, esto es válido y es considerado un «join pint»: puntos de referencia en la lista que no tienen actualmente ningún significado por si mismo pero que puede ser usado para ordenar otros elementos. Los valores nulos una vez ordenados son eliminados (la lista pasada el método constructor de servicio no incluye nulos).

Al usar add() sin ninguna restricción se añade una restricción por defecto: después del elemento anterior. Estas restricciones por defecto solo aplican dentro del método de contribución pero hace mucho mas fácil establecer el orden de varias contribuciones relacionadas. Nota que las contribuciones serán ordenadas relativamente entre sí pero es posible que se intercalen entre ellos contribuciones de otro módulo o método.

Contribuciones mapeadas

Como se ha comentado en ejemplos anteriores, se soportan contribuciones mapeadas. Las claves pasadas deben ser únicas. Cuando ocurre un conflicto Tapestry mostrará advertencias (identificando la fuente del conflicto en términos de métodos invocados) e ignorará el valor del conflicto. Ni la clave ni el valor puede ser nulos. Para

las configuraciones mapeadas donde el tipo de la clave es String se usará automáticamente un CaseInsensitiveMap (y será pasado al método constructor de servicio) para asegurar que la insensibilidad a mayúsculas es automática y ubicua.

Inyectando clases

Las tres interfaces de configuración tienen un segundo método, addInstance(). Este método toma una clase y no una instancia. La clase es instanciada y contribuida. Si el constructor del la clase tiene dependencias esas también son inyectadas.

Sobrescrituras de configuración

Las interfaces OrderedConfiguration y MappedConfiguration soportan sobrescrituras, una sobrescritura es un reemplazo para un objeto contribuido normalmente. Una sobrescritura debe coincidir con un objeto contribuido y cada objeto contribuido puede ser sobrescrito una vez como máximo. El nuevo objeto reemplaza al objeto original, alternativamente puedes sobrescribir el objeto original con null. Esto permite ajustar los valores de la configuración que son contribuidos desde los módulos que estás usando en vez de solo los que está escribiendo tú. Esto es poderoso y un poco peligroso. Con el siguiente código reemplazamos la librerías RequireJS, jQuery y underscore proporcionadas por Tapestry por unas más recientes.

```
@Core
@Contribute(JavaScriptStack.class)
public static void contributeJavaScriptStack(OrderedConfiguration<StackExtension>
    configuration) {
    configuration.override("requirejs", StackExtension.library("classpath:/META-INF/
    resources/webjars/requirejs/2.3.5/require.js"));
    configuration.override("jquery-library", StackExtension.library("classpath:/META-INF/
    resources/webjars/jquery/3.3.1-1/jquery.min.js"));
    configuration.override("underscore-library", StackExtension.library("classpath:/META-
    INF/resources/webjars/underscore/1.9.1/underscore-min.js"));
}
```

4.6 Tutores de servicios

Los tutores o advisors de servicios son una potente facilidad de metaprogramación disponible para los servicios. En realidad, es un tipo de programación orientada a aspectos (AOP, Aspect Oriented Programming) limitada.

Los tutores de servicios te permiten interceptar las invocaciones a los métodos de tus servicios. Tienes la posibilidad de ver que métodos son invocados y cuales son los parámetros. Puedes dejar hacer el trabajo normal del método y entonces inspeccionar o incluso ajustar el valor de retorno o cualquier excepción lanzada. Y puedes hacer todo esto en código Java normal.

Un ejemplo común de tutor a nivel de método es sacar trazas a la entrada y salida de los métodos junto con los valores de los parámetros y las excepciones lanzadas. Otras posibilidades son hacer comprobaciones de seguridad, gestión de transacciones y otro tipo necesidades generales.

Empecemos con un ejemplo artificial. Digamos que tienes un conjunto de servicios que tienen métodos que a veces retornan null y quieres que retornen una cadena vacía porque se están produciendo excepciones NullPointerException en cualquier parte de la aplicación. Puedes acceder a la implementación de cada servicio y corregir la lógica que retorna esos valores... o puedes crear tutores para los métodos.

```
1 @Match("*")
  public static void adviseNonNull(MethodAdviceReceiver receiver) {
3   MethodAdvice advice = new MethodAdvice() {
      void advise(Invocation invocation) {
        invocation.proceed();
        if (invocation.getResultType().equals(String.class) && invocation.getResult() ==
     null)
          invocation.overrideResult("");
8     }
    };
    receiver.adviseAllMethods(advice);
  }
```

Este es un método que se coloca en una clase de módulo. Nota la terminología: advise es el verbo y advice es el nombre. El MethodAdviceReceiver es un envoltorio alrededor el servicio a tutorizar: puedes tutorizar algunos o todos los métodos del servicio y puedes también obtener la interfaz del servicio. Se pasa automáticamente a los métodos de tutorización de servicios.

Los métodos de tutorización deben tener un parámetro de tipo MethodAdviceReceiver. Un servicio puede ser tutorizado múltiples veces (cualquier método puede tener cualquier número de objetos de tutor aplicados a él), algunos métodos pueden no tener ningún tutor, todo esto es aceptable. Los métodos de tutor de servicio son siempre métodos de retorno void.

La anotación @Match("*") indica que este tutor se aplica a todos los servicios (tuyos y definidos por Tapestry). Probablemente querrás reducir los servicios objetivo en la mayoría de casos. Nota que algunos servicios, especialmente aquellos propios de Tapestry IoC están marcados para no ser sujetos a tutorización ni decoración. La interfaz de MethodAdvice es muy simple, recibe un objeto Invocation que representa una invocación de método. Invocation tiene métodos para inspeccionar el tipo y valor de los parámetros y para sobrescribir los valores de los parámetros. La llamada a proceed() permite al método ser invocado. Si el método se ha tutorizado múltiples veces, la llamada a proceed() se encadenará con el siguiente objeto MethodAdvice. En cualquier caso después de invocar proceed() puedes inspeccionar y sobrescribir el resultado. Tutorizar es bastante eficiente, pero aún así es mejor aplicarlo solo en métodos que tiene sentido. Podemos mejorar el tutor del servicio de nuestro ejemplo para solo tutorizar métodos que retornan String:

```java
@Match("*")
public static void adviseNonNull(MethodAdviceReceiver receiver) {
  MethodAdvice advice = new MethodAdvice() {
    void advise(Invocation invocation) {
      invocation.proceed();
      if (invocation.getResult().equals(null))
        invocation.overrideResult("");
    }
  };
  // Tutorizar solo métodos que retornan un String
  for (Method m : receiver.getServiceInterface().getMethods()) {
    if (m.getReturnType().equals(String.class))
      receiver.adviseMethod(m, advice);
  }
};
```

Tutores incorporados

Tapestry incluye dos tutores. El tutor de logging que muestra trazas de las llamadas a los métodos y lazy que hace los métodos que retornan interfaces no se ejecuten inmediatamente sino cuando un método de esa interfaz se invoque. Estos tutores se pueden aplicar de la siguiente manera:

```java
@Match("*")
public static void adviseLogging(LoggingAdvisor loggingAdvisor, Logger logger,
    MethodAdviceReceiver receiver) {
  loggingAdvisor.addLoggingAdvice(logger, receiver);
}
```

Orden y coincidencia

Cada método tutorizado del servicio tiene un id único, obtenido quitando el prefijo advise del nombre del método. Los ids de tutorizado deben ser únicos teniendo en cuenta todos los módulos. Si se omite la anotación @Match el tutor coincidirá con el servicio del mismo id. En algunos casos, el orden en el que se proporcionan los tutores es muy importante; por ejemplo, puedes querer mostrar trazas primero, luego realizar las transacciones y posteriormente las comprobaciones de seguridad. La anotación @Order permite establecer el orden de forma explícita.

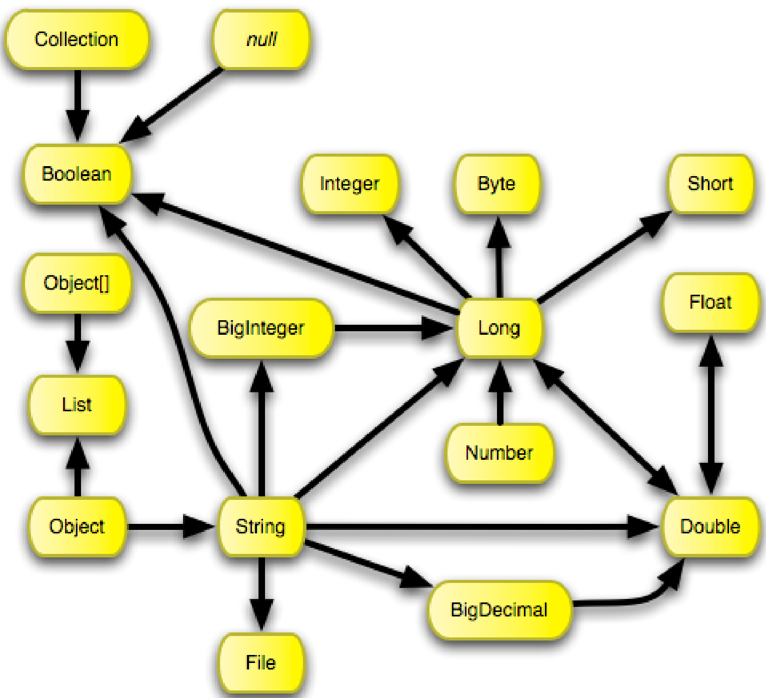
Tutores dirigidos por anotaciones

Tapestry soporta anotar métodos dirigidos por anotaciones. Si está presente la anotación @Advise, el método tutor puede tener cualquier nombre como se muestra en el siguiente ejemplo.

```java
@Advise
@Match("*DAO")
public static void byServiceId(MethodAdviceReceiver receiver) {
  ...
}
```

4.7 Conversiones de tipos

La conversión de tipos o type coercion es la conversión de un tipo de objeto a uno nuevo de diferente tipo con contenido similar. Tapestry frecuentemente debe convertir objetos de un tipo a otro. Un ejemplo común es la conversión de un String a un integer o double. A pesar de que las conversiones ocurren dentro de tapestry-core (incluyendo las conversiones en los parámetros de los componentes), también puede ocurrir en la inyección de dependencias (tapestry-ioc). Como cualquier otra cosa en Tapestry, las conversiones son extensibles. La raíz es el servicio TypeCoercer. Su configuración cosiste en un número de CoercionTuples. Cada tupla define como convertir de un tipo a otro. El conjunto de conversiones incorporadas se centran principalmente en conversiones entre diferentes tipos numéricos:



Tapestry puede interpolar las conversiones necesarias. Por ejemplo, si es necesario convertir de un String a un Integer, el servicio TypeCoercer encadenará una serie de conversiones:

- Object –> String

- String –> Long
- Long –> Integer

Conversión desde null

Hay unas pocas conversiones especiales relacionadas con el valor null, Object –> List envuelve un objeto solitario en una lista de un elemento, pero puede que queramos que el null permanece como null en vez de convertirse una lista con un único elemento null, para ello necesitamos una conversión específica de null –> List. La conversión de null no se expande en conversiones de otros tipo. O hay una conversión del null al tipo deseado o no se produce la conversión y el valor convertido es null. La única conversión desde null es a boolean que es siempre falso.

Lista de conversiones

Esta es la lista completa de conversiones proporcionada por Tapestry:

- Double –> Float
- Float –> Double
- Long –> Boolean
- Long –> Byte
- Long –> Double
- Long –> Integer
- Long –> Short
- Number –> Long
- Object –> Object[]
- Object –> String
- Object –> java.util.List
- Object[] –> java.util.List
- String –> Boolean
- String –> Double
- String –> Long
- String –> java.io.File
- String –> java.math.BigDecimal

- String –> java.math.BigInteger
- String –> java.text.DateFormat
- String –> java.util.regex.Pattern
- String –> org.apache.tapestry5.Renderable
- String –> org.apache.tapestry5.SelectModel
- String –> org.apache.tapestry5.corelib.ClientValidation
- String –> org.apache.tapestry5.corelib.LoopFormState
- String –> org.apache.tapestry5.corelib.SubmitMode
- String –> org.apache.tapestry5.corelib.data.BlankOption
- String –> org.apache.tapestry5.corelib.data.GridPagerPosition
- String –> org.apache.tapestry5.corelib.data.InsertPosition
- String –> org.apache.tapestry5.ioc.Resource
- String –> org.apache.tapestry5.ioc.util.TimeInterval
- boolean[] –> java.util.List
- byte[] –> java.util.List
- char[] –> java.util.List
- double[] –> java.util.List
- float[] –> java.util.List
- int[] –> java.util.List
- java.math.BigDecimal –> Double
- java.util.Collection –> Boolean
- java.util.Collection –> Object[]
- java.util.Collection –> org.apache.tapestry5.grid.GridDataSource
- java.util.Date –> java.util.Calendar
- java.util.List –> org.apache.tapestry5.SelectModel
- java.util.Map –> org.apache.tapestry5.SelectModel
- long[] –> java.util.List
- null –> Boolean
- null –> org.apache.tapestry5.grid.GridDataSource
- org.apache.tapestry5.ComponentResources –> org.apache.tapestry5.PropertyOverrides

- org.apache.tapestry5.PrimaryKeyEncoder –> org.apache.tapestry5.ValueEncoder
- org.apache.tapestry5.Renderable –> org.apache.tapestry5.Block
- org.apache.tapestry5.Renderable –> org.apache.tapestry5.runtime.RenderCommand
- org.apache.tapestry5.ioc.util.TimeInterval –> Long
- org.apache.tapestry5.runtime.ComponentResourcesAware –> org.apache.tapestry5.ComponentResources
- short[] –> java.util.List

Contribuir nuevas conversiones

El servicio TypeCoercer es extensible, puedes añadir las nuevas conversiones que desees. Por ejemplo, digamos que tienes un tipo Dinero que representa una cantidad en alguna moneda y quieres convertir de un BigDecimal a un Dinero. Y asumamos que Dinero tiene un constructor que acepta BigDecimal como parámetro. Usaremos algo de configuración de Tapestry IoC para informar al servicio TypeCoercer de esta conversión.

```
1 public static void contributeTypeCoercer(Configuration<CoercionTuple> configuration) {
    Coercion<BigDecimal, Money> coercion = new Coercion<BigDecimal, Dinero>() {
      public Dinero coerce(BigDecimal input) {
        return new Dinero(input);
5     }
    };
    configuration.add(new CoercionTuple<BigDecimal, Dinero>(BigDecimal.class, Dinero.class,
       coercion));
  }
```

Además, como TypeCoercer conoce como convertir de Double a BigDecimal o incluso Integer (desde Long y Double) a BigDecimal, todas estos tipos de conversiones funcionarán también. Al crear una conversión desde null, usa Void.class como la fuente del tipo. Por ejemplo, la conversión de null a Boolean está implementada como:

```
1 public static void contributeTypeCoercer(Configuration<CoercionTuple> configuration) {
2   Coercion<Void, Boolean> coercion = new Coercion<Void, Boolean>() {
      public Boolean coerce(Void input) {
        return false;
      }
    };
7   configuration.add(new CoercionTuple<Void, Boolean>(Void.class, Boolean.class, coercion)
      );
  }
```

4.8 Símbolos de configuración

Muchos de los servicios integrados de Tapestry, algunos de los cuales son públicos, son configurados mediante símbolos. Estos símbolos pueden ser sobrescritos haciendo contribuciones a la configuración del servicio ApplicationDefaults o colocando un elemento context-param en el archivo web.xml de la aplicación o mediante la linea de comandos definiendo propiedades de la JVM con la opción -D.

Estos símbolos son siempre definidos en términos de Strings y esos Strings son convertidos al tipo apropiado (un número, booleano, etc). En la clase SymbolConstants pueden encontrarse muchos de los símbolos que pueden modificar el comportamiento de la aplicación.

```
public static void contributeApplicationDefaults(MappedConfiguration<String, Object>
    configuration) {
  configuration.add(SymbolConstants.PRODUCTION_MODE, false);
  configuration.add(SymbolConstants.SUPPORTED_LOCALES, "es,en");
  configuration.add(SecuritySymbols.LOGIN_URL, "/login");
  configuration.add(SecuritySymbols.SUCCESS_URL, "/index");
  configuration.add(SecuritySymbols.UNAUTHORIZED_URL, "/unauthorized");
  configuration.add(SecuritySymbols.REDIRECT_TO_SAVED_URL, "true");
  configuration.add(SymbolConstants.APPLICATION_VERSION, "1.0");
  configuration.add(SymbolConstants.JAVASCRIPT_INFRASTRUCTURE_PROVIDER, "jquery");
}
```

Tu aplicación y servicios también puede definir nuevos símbolos y pueden ser usados en diferentes partes de tu aplicación:

- Servicios. En el siguiente ejemplo el servicio inyectado para construir otro es determinado por un símbolo. Modificando el símbolo se puede cambiar la aplicación sin necesidad de cambiar el código.

```
public static MiServicio build(@InjectService("${id-servicio}") Collaborator
    colaborador) {
  return ...;
}
```

- Valores. En este caso se usa el valor de un símbolo para variar el comportamiento dependiendo de su valor.

```
public class MiServicio implements MiServicioInterface {
  public MiServicio(@Symbol(SymbolConstants.PRODUCTION_MODE) boolean productionMode)
     {
    if (productionMode) {
      ...
    }
  }
}
```

Símbolos recursivos

Es posible y válido definir un símbolo en términos de uno o más símbolos.

```
1 public void contributeFactoryDefaults(MappedConfiguration<String, String> configuration)
      {
    configuration.add("report.url", "http://${report.host}:${report.port}/${report.path}");
3   configuration.add("report.host", "www.localhost.com.local");
    configuration.add("report.port", "80");
    configuration.add("report.path", "/report");
  }
```

El valor por defecto de report.url será http://www.localhost.com.local:80/report pero puede ser cambiado haciendo una contribución de sobrescritura a la configuración del servicio ApplicationDefaults.

Tapestry comprueba que ningún símbolo es directamente o indirectamente dependiente de si mismo. Por ejemplo la siguiente contribución es ilegal:

```
1 public void contributeApplicationDefaults(MappedConfiguration<String, String>
      configuration) {
    configuration.add("report.path", "${report.url}/report.cgi");
  }
```

Cuando report.url sea referenciado se producirá un excepción con el mensaje: Symbol 'report.path' is defined in terms of itself (report.path –> report.url –> report.path), más que suficiente para detectar y corregir el problema rápidamente.

Capítulo 5

Assets y módulos RequireJS

En Tapestry los assets son cualquier tipo de contenido estático que puede ser descargado a un navegador web cliente como imágenes, hojas de estilo y archivos javascript. Los assets son normalmente almacenados en la carpeta de contexto de la aplicación web. Además, Tapestry trata algunos de los archivos almacenados en el classpath junto a tus clases como assets visibles para el navegador web. Los assets son expuestos a tu código como instancias de la interfaz Asset.

5.1 Assets en las plantillas

Los assets también pueden ser referenciados directamente en las plantillas. Hay dos prefijos de binding para esto: asset: y context:. El prefijo asset puede obtener assets del classpath o con el prefijo context del contexto de la aplicación (especificando context: de forma explícita, si no se especifica prefijo se usa asset):

```
1 <img src="${asset:context:image/tapestry.png}" alt="Banner"/>
```

Este es un ejemplo de usar una expansión de plantilla dentro de un elemento ordinario (en vez de un componente). Dado que acceder a assets es muy común existe el prefijo context:

```
1 <img src="${context:image/tapestry.png}" alt="Banner"/>
```

5.2 Assets en las clases de componente

Los componentes obtienen referencias a assets mediante una inyección. La anotación @Inject permite inyectar un asset en los componentes como propiedades de solo lectura. La ruta al recurso es especificado usando además la anotación @Path como en el siguiente ejemplo:

```
@Inject
@Path("context:images/tapesty.png")
private Asset banner;
```

Los assets son almacenados en dominios, estos dominios están identificados por un prefijo en el valor de la anotación @Path. Si el prefijo es omitido, el valor será interpretado como una ruta relativa a la clase Java dentro del dominio classpath:. Esto es usado habitualmente al crear librerías de componentes donde los assets usados por los componentes son empaquetados en el jar con los archivos .class de los propios componentes. Al contrario que en todas las otras partes, las mayúsculas importan. Esto es porque Tapestry es dependiente de la Servlet API y el entorno de ejecución de Java para acceder a los archivos, y esas API, al contrario de Tapestry, son sensibles a mayúsculas. Ten encuentra que algunos sistemas operativos (como Windows) son insensibles a mayúsculas lo que puede enmascarar errores que se harán notar en el momento de despliegue (si el sistema operativo de despliegue es sensible a mayúsculas como Linux).

Assets relativos

Puedes usar rutas relativas con dominios (si omites el prefijo). Dado que debes omitir el prefijo, esto solo tiene sentido para componentes empaquetados en una librería para ser reutilizados.

```
@Inject
@Path("../edit.png")
private Asset icon;
```

Símbolos para assets

Los símbolos dentro del valor de la anotación son expandidos. Esto te permite definir un símbolo y referenciarlo como parte de la ruta. Por ejemplo, puedes contribuir un símbolo llamado skin.root como context:skins/basic y referenciar un asset como:

```
@Inject
@Path("${skin.root}/style.css")
private Asset style;
```

El uso de la sintaxis ${...} es aquí una expansión de símbolo porque ocurre en una anotación en código Java en vez ser una expansión de plantilla que solo ocurre en un archivo de plantilla. Una sobrescritura del símbolo skin.root afectaría a todas sus referencias en los assets.

Localización de assets

Los assets son localizados, Tapestry buscará la variación del archivo apropiado al locale efectivo de la petición. En el ejemplo previo, un usuario alemán de la aplicación podría ver el archivo edit_de.gif si ese archivo existiese.

Nuevos dominios de assets

Si quieres crear nuevos dominios para assets, por ejemplo para que puedan ser almacenados en el sistema de archivos, en la base de datos o de un almacén de S3 de Amazon podrías definir una nueva factoría y contribuirla a la configuración del servicio AssetSource.

URL de asset

Tapestry crea una nueva URL para los assets (sea de contexto o de classpath). Esta URL tiene la forma /asset[.gz]/carpeta/hash/ruta.

- carpeta: identifica la librería contenedora del asset, ctx para un asset de contexto o stack cuando se combinan múltiples archivos javascript en un único asset virtual.
- hash: código hash obtenido a partir del contenido del archivo.
- ruta: la ruta debajo del paquete raíz de la librería hasta el archivo de asset especifico.

Notas de rendimiento

Se espera que los assets sean totalmente estáticos (que no cambien mientras la aplicación está desplegada). Esto permite a Tapestry realizar algunas optimizaciones importantes de rendimiento. Tapestry comprime con gzip el contenido de los assets si el asset es comprimible, el cliente lo soporta y tu explícitamente no lo desactiva. Cuando Tapestry genera la URL para el asset, ya sea en el classpath o del contexto, la URL incluye un código hash único del asset. Además, el asset tendrá una cabecera de expiración lejana en el tiempo lo que promoverá que el cliente cachee el asset. Mientras el contenido del asset y su hash no cambie el cliente podrá conservarlo en la cache. Los navegadores de los clientes cachearan de forma agresiva los assets, normalmente no enviarán ni siquiera la petición para ver si el asset ha cambiado una vez que ha sido descargado por primera vez.

Seguridad en los asset

Dado que Tapestry expone directamente archivos del classpath a los clientes, hay que asegurar que clientes maliciosos no pueden descargar assets que no deberían ser visibles a ellos. Primero, hay una limitación de

paquete: los assets de classpath solo son visibles si hay un LibraryMapping para ellos y el mapeo de librería sustituye las carpetas iniciales del classpath. Dado que los assets más seguros como hibernate.cfg.xml están localizados en el paquete anónimo estos está fuera de los límites. Pero también hay que securizar los archivos .class ya que decompilándolo se podrían obtener claves si están en el código fuente. Por fortuna esto no puede ocurrir. Los archivos con la extensión .class son protegidos, las peticiones deben venir acompañadas en la URL con un parámetro de query que es el hash MD5 del contenido del archivo. Si el parámetro no viene o no coincide con el contenido actual del archivo la petición es denegada. Cuando tu código expone un asset la URL automáticamente incluye el parámetro de query si el tipo del archivo está protegido. Por defecto, Tapestry protege los archivos de extensión .class, .tml y .properties. La lista puede ser extendida contribuyendo al servicio ResourceDigestGenerator:

```
1 public static void contributeResourceDigestGenerator(Configuration<String> configuration)
      {
2   configuration.add("doc");
  }
```

5.3 Minimizando assets

Tapestry proporciona un servicio ResourceMinimizer que ayuda a minimizar todos tus los recursos estáticos (principalmente archivos CSS y JavaScript). Para ello basta con incluir una librería.

```
1 org.apache.tapestry:tapestry-webresources
```

Añadiendo esta dependencia, todos tus archivos JavaScript y CSS serán minimizados con el símbolo PRODUCTION_MODE a true. Puedes forzar la minimización específicamente de estos archivos cambiando el valor de la constante SymbolConstants.MINIFICATION_ENABLED en tu clase de módulo:

```
1 @Contribute(SymbolProvider.class)
  @ApplicationDefaults
  public static void contributeApplicationDefaults(MappedConfiguration<String, String>
      configuration) {
4   configuration.add(SymbolConstants.PRODUCTION_MODE, true);
    configuration.add(SymbolConstants.MINIFICATION_ENABLED, true);
  }
```

5.4 Hojas de estilo

La mayoría de aplicaciones web delegan en las hojas de estilo (CSS, Cascading Style Sheets) los detalles de estilo de la página como fuentes, colores, margenes, bordes y alineamiento. Esto ayuda a que el html se mantenga

simple y semántico lo que hace que sea más fácil de mantener y leer. Tapestry incluye un sofisticado soporte para los CSS en forma de enlaces con anotaciones, cabeceras de expiración lejanas en en tiempo, eliminación de duplicados automático y otras características proporcionados por los assets.

Hoja de estilos por defecto

Tapesty incluye varias hojas de estilo al hacer uso del stack core, entre ellas la de bootstrap. Haciendo que las hojas de estilo del stack core sean las primeras se permite sobrescribir esos estilos por los tuyos propios.

Añadiendo tu propio CSS

Una página o componente (por ejemplo un componente de layout) que renderice la etiqueta <head> puede añadir hojas de estilo directamente en el lenguaje de marcas.

```
<head>
  <link href="/css/site.css" rel="stylesheet" type="text/css"/>
  ...
</head>
```

Si quieres usar el soporte de localización de Tapestry deberías usar una expansión y el prefijo de binding asset: o context:

```
<head>
  <link href="${context:css/site.css}" rel="stylesheet" type="text/css"/>
  ...
</head>
```

El prefijo context: significa que el resto de la expansión es una ruta al asset de contexto, un recurso en la raíz de la aplicación web (src/main/webapp en tu espacio de trabajo). Por contrario, el prefijo asset indica a Tapestry que lo busque en el classpath.

Usando la anotación @Import

Otra forma de añadir una hoja de estilos es incluir una anotación @Import en la clase del componente:

```
@Import(stylesheet = "context:css/site.css")
public class MiComponente {
  ...
}
```

Como las librerías de javascript incluidas, cada hoja de estilos solo será añadida una sola vez, independientemente del número de componentes que la incluyan mediante la anotación.

Cargar hojas de estilos condicionalmente

Se puede incluir hojas de estilo condicionalmente, lo que puede ser útil para los navegadores Internet Explorer, de la siguiente forma:

```
<!--[if IE]>
<link type="text/css" rel="stylesheet" href="/asset/ctx/8e7bae2f/layout/ie-only.css"></
    link>
<![endif]-->
```

5.5 JavaScript

Javascript es un concepto de primera clase en Tapestry y se proporciona un sofisticado soporte Javascript listo para usar, incluyendo soporte ajax, optimización de descarga, logging en el cliente y localización. En el modo producción, por defecto, Tapestry fusionará librerías javascript y se establecerá una cabecera http de expiración lejana en el tiempo para promover el cacheo agresivo en el navegador. Tapestry también puede minificar (comprimir) librerías javascript en el modo producción. Además, puede usarse fácilmente librerías como jquery.

5.5.1 Añadiendo JavaScript personalizado

Para añadir tu propio javascript o librerías de terceros solo has de seguir las estrategias de abajo para aprovechar las ventajas de los mecanismos de soporte de javascript. La práctica recomendada en Tapestry es empaquetar cualquier cantidad significativa de javascript como una librería estática de javascript, un archivo .js que puede ser descargado al cliente y cacheado. Mantén tu código javascript de página al mínimo, solo las pocas sentencias para inicializar los objetos y métodos de referencia en la librerías de javascript de modo que el tamaño de las páginas sean más pequeñas y el cliente tenga menos KiB que descargar ello redundará en una aplicación más rápida y un servidor que puede procesar más peticiones por unidad de tiempo.

Enlazando a tus librerías de JavaScript

Tapestry proporciona varias maneras para enlazar a una librería javascript desde tu página o componente. A pesar de que puedes usar etiquetas script directamente deberías usarlas solo para javascript que resida fuera de la aplicación. Para javascript dentro de la aplicación, Tapestry proporciona mejores maneras para hacer lo mismo. La mayoría de los usuarios usan la mas simple, usar la anotación @Import.

Método 1: @Import

Usa la anotación @Import para incluir enlaces a archivos javascript (y css) en tus páginas o componentes. Tapestry asegura que ese archivo solo es referenciado una sola vez en la página.

```
1 @Import(library={"context:js/jquery.js", "context:js/app.js"})
2 public class MiComponente {
    ...
  }
```

La anotación @Import también puede se aplicada a métodos individuales en cuyo caso la operación import solo ocurre cuando el método es invocado. Añadir la misma librería de javascript múltiples veces no crea enlaces duplicados, los siguientes son simplemente ignorados. De esta forma, cada componente puede añadir las librerías que necesite sin preocuparse de conflictos con otros componentes.

Método 2: Usar módulos

Desde la versión 5.4 de Tapestry la forma recomendada para incluir javascript en una página es a través de RequireJS y módulos. Ver el aparado RequireJS y módulos de Javascript.

Inyectando el servicio JavaScriptSupport

JavaScriptSupport es un objeto de entorno, de modo que normalmente lo inyectas mediante la anotación @Environmental:

```
1 @Environmental
  private JavaScriptSupport support;
```

La anotación @Environmental solo funciona dentro de componentes pero ocasionalmente puedes querer inyectar JavaScriptSupport en un servicio. Afortunadamente, se ha configurado un proxy para permitirte usar @Inject en su lugar en una propiedad:

```
1 @Inject
  private JavaScriptSupport support;
```

... o en una implementación mediante su constructor:

```java
public MiServicioImpl(JavaScriptSupport support) {
  ...
}
```

Dentro de un componente deberías usar @Environmental para resaltar el hecho de que RenderSupport (como la mayoría de objetos de entorno) solo esta disponible durante el renderizado no durante peticiones de acción.

5.5.2 Combinando librerías de JavaScript

En el modo producción, Tapestry combina automáticamente las librerías javascript. Una sola petición (para un asset virtual) obtendrá el contenido combinado para todos los archivos de librería javascript si pertenecen a un Stack. Esta es un característica útil ya que reduce el número de peticiones para visualizar una página que ha de hacer el navegador del usuario. Puede ser deshabilitada estableciendo el símbolo de configuración SymbolConstants.COMBINE_SCRIPTS a false en tu clase de módulo de aplicación. Por defecto esta habilitado en el modo de producción y deshabilitado en otro caso. Como en otro lugares, si el navegador soporta compresión gzip el archivo combinado será comprimido.

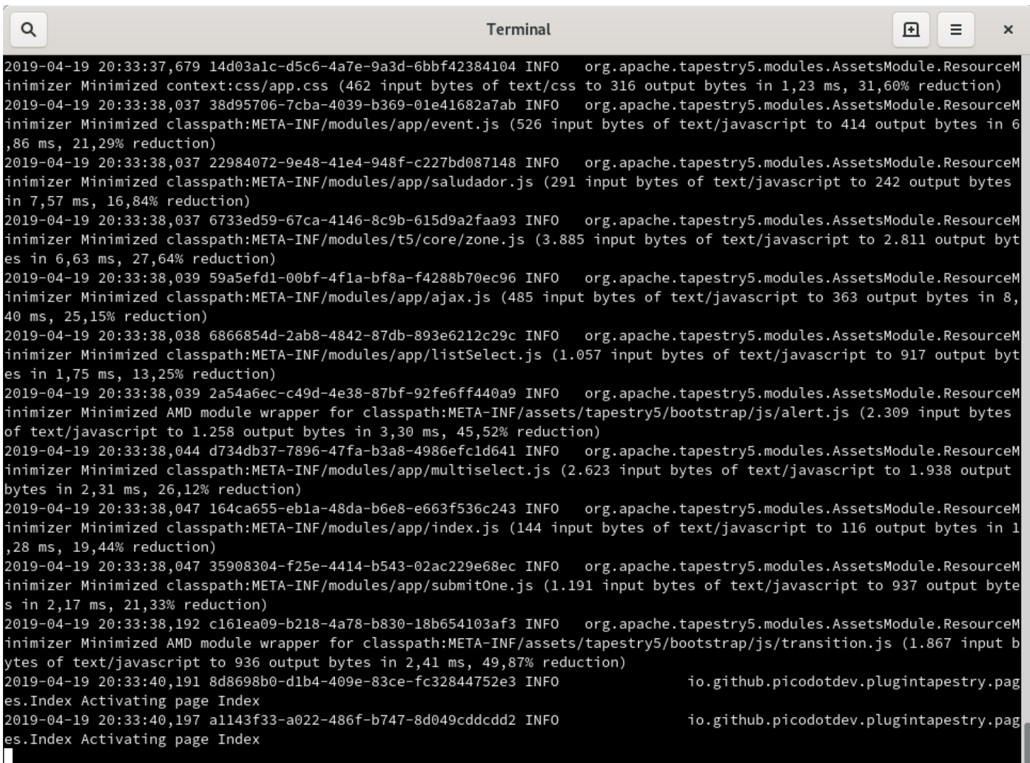
5.5.3 Minificando librerías de JavaScript

En el modo producción, además de combinarlo Tapestry puede minificar (comprimir inteligentemente) las librerías javascript (y CSS) cuando la aplicación se inicia. Esto puede disminuir significativamente el tamaño del contenido estático que el navegador necesita descargar. La minificación es llevada a cabo usando el servicio ResourceMinimizer. La implementación se basa en wro4j.

Nota: el modulo tapestry-core solo proporciona una infraestructura vacía para la minificación, la lógica actual es proporcionada en el modulo tapestry-webresources. Para usarlo, necesitaras actualizar tus dependencias para incluir este modulo.

```
compile "org.apache.tapestry:tapestry-webresources:$versions.tapestry"
```

La minificación puede ser desactivada estableciendo el símbolo de configuración SymbolConstants.MINIFICATION_ENABLE a false en tu clase de módulo de aplicación. Por defecto es habilitado en el modo producción y deshabilitado en otro caso.



Como se puede ver en la captura dependiendo del archivo se puede conseguir una reducción del tamaño notable antes de aplicar la compresión gzip usada al enviarlo al cliente.

5.5.4 Pilas de recursos

Tapestry te permite definir grupos de librerías de javascript, módulos de javascript, código javascript de inicialización y hojas de estilo como pilas o stacks como una unidad. El stack incorporado core es usado para definir las librerías de javascript del núcleo necesitadas por Tapestry. Otras librerías de componente pueden definir stack adicionales para conjuntos de recursos relacionados, por ejemplo, para agrupar junto a algunas porciones de las librerías ExtJS y YUI. Los stacks de assets pueden (si se habilita) ser expuestas al cliente como una única URL (identificando el nombre del stack por nombre). Los assets individuales son combinados en un solo asset virtual que es enviado al cliente. Para agrupar varios recursos estáticos juntos en un solo stack debes crear una nueva implementación de la interfaz JavaScriptStack. Esta interfaz tiene cuatro métodos:

- getStylesheets: este método retorna una lista de archivos de estilo asociados con el stack.
- getJavaScriptLibraries: este método retorna una lista de archivos javascript asociados con este stack.
- getStacks: es posible hacer un stack dependiente de otros stacks. Todos los stacks definidos en este método será cargados antes que el stack actual.
- getInitialization: este método hace posible llamar un javascript de inicialización para el stack. Tapestry automáticamente añadirá esta inicialización en la pagina que importa el stack.

```java
public class PlugInStack implements JavaScriptStack {

  private final AssetSource assetSource;

  public PlugInStack(final AssetSource assetSource) {
    this.assetSource = assetSource;
  }

  @Override
  public String getInitialization() {
    return null;
  }

  @Override
  public List<String> getModules() {
    return Collections.emptyList();
  }

  @Override
  public List<Asset> getJavaScriptLibraries() {
    List<Asset> r = new ArrayList<>();
    r.add(assetSource.getClasspathAsset("META-INF/assets/tapestry5/bootstrap/js/dropdown.
    js"));
    r.add(assetSource.getClasspathAsset("META-INF/resources/webjars/bootstrap-select
    /1.13.8/js/bootstrap-select.min.js"));
    return r;
  }

  @Override
  public JavaScriptAggregationStrategy getJavaScriptAggregationStrategy() {
    return JavaScriptAggregationStrategy.COMBINE_AND_MINIMIZE;
  }

  @Override
  public List<StylesheetLink> getStylesheets() {
    List<StylesheetLink> r = new ArrayList<>();
    r.add(new StylesheetLink(assetSource.getClasspathAsset("META-INF/resources/webjars/
    bootstrap-select/1.13.8/css/bootstrap-select.min.css", null)));
    r.add(new StylesheetLink(assetSource.getContextAsset("css/app.css", null)));
    return r;
  }

  @Override
  public List<String> getStacks() {
    return Collections.emptyList();
  }
}
```

Cuando hayas creado tu nuevo stack puedes definirlo en tu módulo:

```
@Core
@Contribute(JavaScriptStack.class)
public static void contributeJavaScriptStack(OrderedConfiguration<StackExtension>
    configuration) {
  configuration.override("requirejs", StackExtension.library("classpath:/META-INF/
    resources/webjars/requirejs/2.3.5/require.js"));
  configuration.override("jquery-library", StackExtension.library("classpath:/META-INF/
    resources/webjars/jquery/3.3.1-1/jquery.min.js"));
  configuration.override("underscore-library", StackExtension.library("classpath:/META-
    INF/resources/webjars/underscore/1.9.1/underscore-min.js"));
}
```

Y puedes usarlo en tus paginas y componentes usado la anotación @Import o el servicio JavaScriptSupport. Con la anotación @Import:

```
@Import(stack = { "core", "plugin" })
public class Layout {
  ...
}
```

Con JavaScriptSupport:

```
public class MiPagina {

  @Inject
  private JavaScriptSupport js;

  public void setupRender() {
    js.importStack("plugin");
  }
}
```

5.5.5 RequireJS y módulos de Javascript

Una de las novedades que incorpora Tapestry 5.4 siguiendo la evolución que están tomando las aplicaciones web es el uso de módulos mediante RequireJS dado el mayor peso que está tomando javascript.

Las páginas web ha evolucionado mucho desde sus inicios en los que eran simples páginas estáticas hechas con el lenguaje de marcas html, podían contener enlaces e imágenes. Posteriormente adquirieron capacidad de cambiar a través un lenguaje de programación que en el servidor genera el código html de forma dinámica

basándose en la información que el usuario podía enviar en un formulario, también se incorpora cierta programación en el cliente con javascript. Con las nuevas versiones del estándar html, los avances de los navegadores y una explosión de dispositivos móviles de gran capacidad las aplicaciones están evolucionando hacia el cliente, haciéndose cada vez más complejas en el lado del navegador del usuario y adquiriendo responsabilidades que antes tenía la aplicación en el lado del servidor. Cada vez hay más librerías y frameworks javascript que tratan de resolver problemas específicos de las aplicaciones de internet. Entre estas librerías algunas de las más conocidas son, muy resumidamente:

- jQuery: para manejar los elementos de la página.
- Mustache: a partir de una plantilla y unos datos genera un resultado.
- Underscore: proporciona ciertas utilidades bastante comunes que el lenguaje javascript no proporciona.
- Backbone: da un modelo MVC para el desarrollo de las aplicaciones.
- React: capa de vista en el lado cliente que se basa en componentes con similitudes entre los componentes de Tapestry. Se puede combinar Backbone para los modelos y a modo de controlador y React para la vista.

Por supuesto, para cada área hay varias opciones entre las que se puede elegir, estas no son las únicas librerías hay muchas más alternativas (hadlebars, prototype, agularjs, mootools, ...) que en esencia proporcionan la misma funcionalidad que las anteriores. A medida que vamos haciendo uso de más librerías, archivos javascript y que estas pueden tener dependencias unas sobre otras se hace necesario algo que permita gestionar esas relaciones entre las librerías para que el código javascript se cargue en el orden adecuado y funcione correctamente. Aquí surge RequireJS, que además de gestionar esas dependencias también nos proporciona otras ventajas:

- La carga de los archivos javascript se hace de forma asíncrona evitando el resto del contenido de la página se bloque hasta que los js de la página se carguen.
- Se evita contaminar el ámbito global de javascript evitando posibles conflictos entre archivos javascript.

Algunas de estas ventajas hacen que la página cargue más rápido que es algo que el buscador de Google tiene muy en cuenta para el posicionamiento en los resultados de búsqueda.

Con RequireJS los archivos javascript se organizan en módulos y estos pueden tener dependencias sobre otros, todos los archivos javascript necesarios son cargados por RequireJS de forma asíncrona. Haciendo uso de RequireJS en la página web solo será necesario incluir un único javascript, que será el de RequireJS, y Tapestry lo hace ya de forma automática sin que tengamos que hacer nada. El resto de módulos se cargarán cuando alguno de los componentes usados en la página lo requiera.

En el siguiente ejemplo se muestra un componente que carga vía javascript el contenido de una etiqueta, el javascript es definido como un módulo de RequireJS.

Listado 5.1: Javascript.java

```java
package io.github.picodotdev.plugintapestry.components;

...

public class Javascript {

  @Parameter(defaultPrefix = BindingConstants.LITERAL)
  @Property
  private String selector;

  @Parameter(defaultPrefix = BindingConstants.LITERAL)
  @Property
  private String mensaje;

  @Environmental
  private JavaScriptSupport support;

  void setupRender() {
    JSONObject o = new JSONObject();
    o.put("selector", selector);
    o.put("mensaje", mensaje);

    support.require("app/saludador").invoke("init").with(o);
  }
}
```

Listado 5.2: META-INF/modules/app/saludador.js

```javascript
define("app/saludador", ["jquery"], function($) {

  function Saludador(spec) {
    this.spec = spec;
  }

  Saludador.prototype.render = function() {
    $(this.spec.selector).html(this.spec.mensaje);
  }

  function init(spec) {
    new Saludador(spec).render();
  }

  return {
    init: init
  };
});
```

Listado 5.3: Index.tml

```
1 <p>
2   <span id="holaMundoJavascript"></span> (Componente que hace uso del soporte javascript
      con RequireJS)
    <t:javascript selector="literal:#holaMundoJavascript" mensaje="literal:¡Hola mundo! (
      javascript)" />
  </p>
```

En versiones anteriores de Tapestry ya era habitual pero con la adición de los módulos mediante RequireJS se hace sencillo evitar que el html de las páginas lleven código javascript embebido. El hacer que el código javascript esté externalizado del html hace que los archivos js puedan ser cacheados por el navegador y reducen el tamaño de las páginas, esto ayuda a que las aplicaciones sean más eficientes y rápidas entre otras ventajas derivadas de ello.

5.6 Assets con Webjars

Una aplicación web se compone de código de lado de servidor, en el caso de utilizar el lenguaje de programación Java de código Java normalmente utilizando algún de los muchos framework web, por otra parte se compone de código de lado de cliente con una gran variedad de librerías de JavaScript como jQuery, React, Underscore o Bootstrap para los estilos. En las aplicaciones Java las librerías de lado de servidor se gestionan como dependencias del proyecto y con herramientas como Gradle se puede automatizar el descargar la librería de repositorios como Maven Central y la versión que se necesite así como hacer sencillo actualizar a una nueva. En el caso de las librerías de lado del cliente con Webjars se consiguen los mismos beneficios.

Los webjars son librerías de extensión jar con los recursos de lado del cliente empaquetados en ellos que en el momento de ser requeridos pueden ser devueltos como un recurso estático por la aplicación, incluyen los archivos JavaScript sin minimizar y minimizados, los archivos map para depuración si minimizados están ofuscados, recursos de estilos CSS o imágenes. Se gestionan como cualquier otra dependencia del proyecto Java lo que proporciona las mismas ventajas de obtener las dependencias de forma automática y hace fácil actualizar a una nueva versión. Por si fuera poco es muy sencillo utilizar webjars, para los frameworks más populares se ofrece una pequeña guía de uso en la documentación.

Las librerías más populares de JavaScript o CSS están empaquetadas como webjars en las diferentes versiones y han sido publicadas de forma que es posible añadir la dependencia en la versión concreta que necesite la aplicación. Dado que los webjars se gestionan como una dependencia Java si estos a su vez tiene alguna dependencia sobre otra librería está se incluyen en el proyecto de forma transitiva. El contenido del webjar para jQuery es el siguiente.

```
1 $ tree
  .├──
   META-INF│
```

```
    ├── MANIFEST.MF│
    ├── maven│
    │   └── org.webjars│
    │       └── jquery│
    │           ├── pom.properties│
    │           └── pom.xml│
    └── resources│
        └── webjars│
            └── jquery│
                └── 3.3.1-1│
                    ├── jquery.js│
                    ├── jquery.min.js│
                    ├── jquery.min.map│
                    └── webjars-requirejs.js└──
 jquery-3.3.1-1.jar
```

En el caso del framework web Apache Tapestry basado en componentes para el desarrollo de aplicaciones web Java tan solo hay que incluir la dependencia en el proyecto y un poco de configuración en el módulo de la aplicación para el contenedor de dependencias como se indica en la guía de uso con el objetivo que los recursos de los webjars sean servidos.

Listado 5.4: AppModule.java

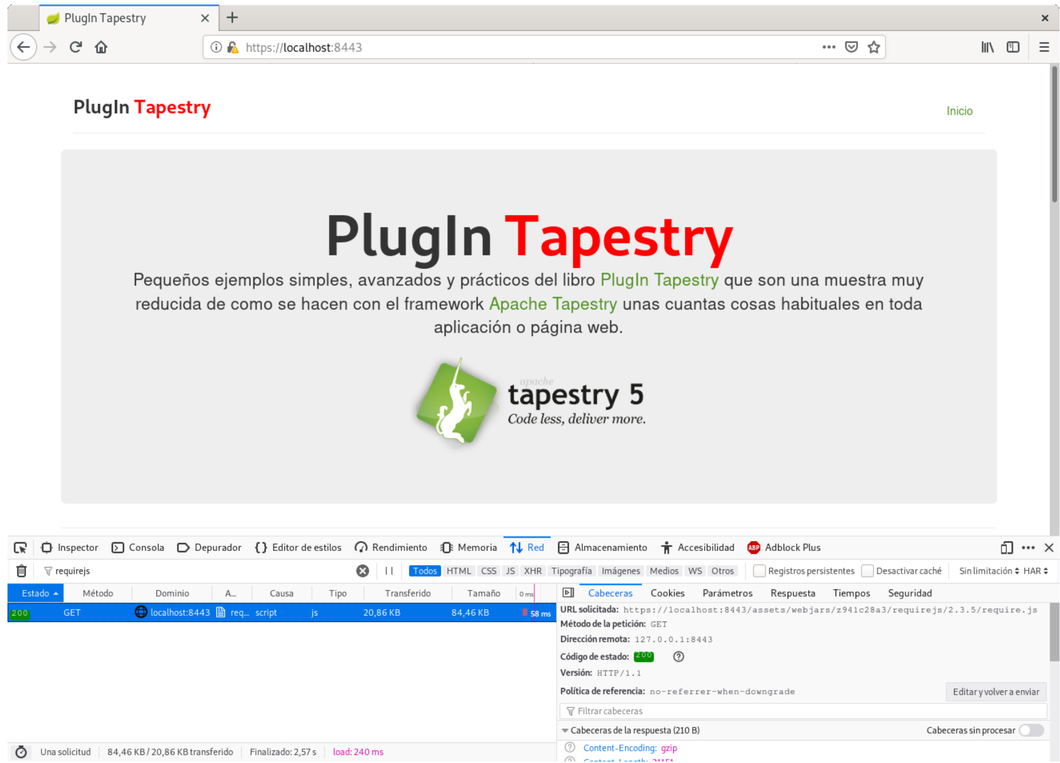
```java
package io.github.picodotdev.plugintapestry.services;

public class AppModule {
    ...

    public static void contributeClasspathAssetAliasManager(MappedConfiguration
    configuration) {
        configuration.add("webjars", "META-INF/resources/webjars");
    }

    ...
}
```

Listado 5.5: build.gradle

```
...

dependencies {

    ...

    // Webjars
    runtime("org.webjars:requirejs:$versions.webjars.requirejs")
```

```
9       runtime("org.webjars:jquery:$versions.webjars.jquery")
        runtime("org.webjars.bower:underscore:$versions.webjars.underscore")
        runtime("org.webjars:bootstrap-select:$versions.webjars.bootstrapSelect")

        ...
14
    }

    ...
```

Inspeccionando el código fuente de la página devuelta se observa que en el caso de Tapestry la URL generada al solicitar en un webjar es del estilo https://localhost:8443/assets/webjars/z941c28a3/requirejs/2.3.5/require.js.



Los webjars muy útiles para gestionar las librerías de lado cliente que hacen innecesario descargar manualmente las dependencias, automatizan la descarga, hacen muy sencillo actualizar a nuevas versiones y es muy fácil de usar al no requerir mucha confiración ni ser invasiva. Además, al estar como una dependencia en el archivo de construcción del proyecto queda indicado de forma explícita que el proyecto utiliza y necesita esa librería. Para mi son una herramienta imprescindible.

5.7 Actualizar versiones assets incorporados por defecto

El framework Apache Tapestry para el desarrollo de aplicaciones web Java basado en componentes aparte de ser un framework para el desarrollo de la capa de presentación del lado del servidor y lógica de negocio también ofrece soporte para el desarrollo de funcionalidad del lado del cliente. Incorpocopora de serie las librerías

RequireJS para la gestión de módulos y dependencias de JavaScript, la popular jQuery para la manipulación de elementos del HTML y Underscore que añade algunas utilidades que no tiene el lenguaje JavaScript y Bootstrap para los estilos además de alguna otra librería JavaScript de menor relevancia que estas.

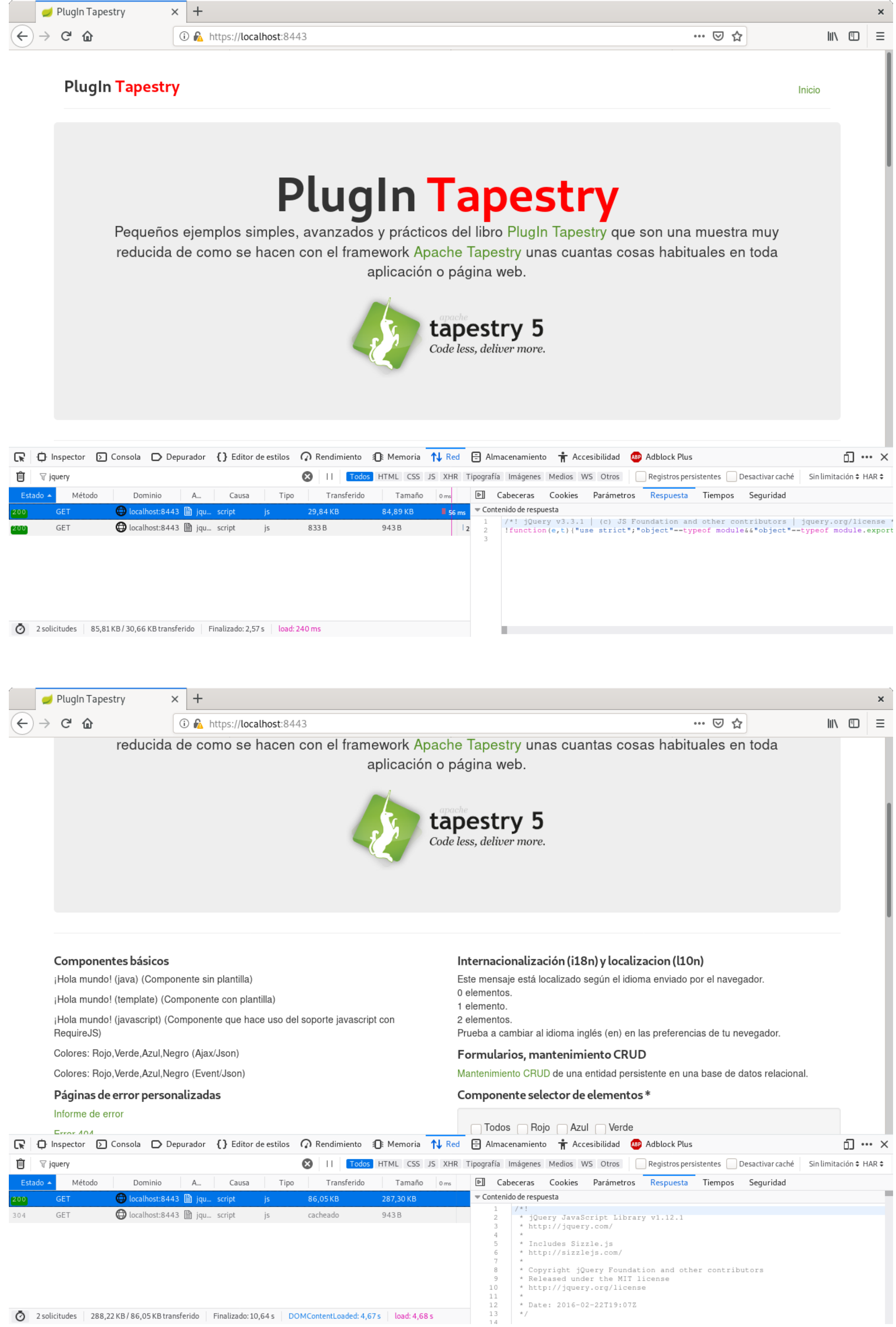
Sin embargo, las versiones de las librerías de lado del cliente que incorpora de serie son antiguas. En la versión 5.4.3 de RequireJS se incorpora la versión 2.1.17, de jQuery la versión 1.12.1 y de Underscore la versión 1.8.3 cuando en el momento de publicar este artículo sus versiones más nuevas son 2.3.5, 3.3.1 y 1.9.1 respectivamente. Dado que de Apache Tapestry no se publican versiones frecuentemente el framework no sigue el ritmo de actualizaciones más rápido de las librerías JavaScript. Pero pueden ser actualizadas sin mucho esfuerzo.

Apache Tapestry es un framework extremadamente personalizable, adaptable y extensible, prácticamente cualquier cosa interna de su funcionamiento puede ser modificada gracias a su propio gestor de dependencias o inversión de control. Las versiones de las librerías anteriores se definen en el archivo JavaScriptModule.java del código fuente de Tapestry y haciendo una contribución en el contenedor de dependencias a la configuración del servicio JavaScriptStack se pueden modificar.

Estas pocas líneas de código bastan para redefinir las versiones de las librerías.

Listado 5.6: AppModule.java

```java
package io.github.picodotdev.plugintapestry.services;

...

public class AppModule {

    ...

    @Core
    @Contribute(JavaScriptStack.class)
    public static void contributeJavaScriptStack(OrderedConfiguration<StackExtension> configuration) {
        configuration.override("requirejs", StackExtension.library("classpath:/META-INF/resources/webjars/requirejs/2.3.5/require.js"));
        configuration.override("jquery-library", StackExtension.library("classpath:/META-INF/resources/webjars/jquery/3.3.1-1/jquery.min.js"));
        configuration.override("underscore-library", StackExtension.library("classpath:/META-INF/resources/webjars/underscore/1.9.0/underscore-min.js"));
    }

    ...
}
```



Además, en este caso las nuevas versiones las he proporcionado gestionando las dependencias de lado del cliente con webjars que se incluyen como cualquier otra dependencia Java del proyecto. Esto permite saber qué dependencias de lado de cliente tiene el proyecto, obtener las dependencias de forma automática y actualizarlas de forma sencilla con la herramienta de construcción del proyecto como Gradle.

Al usar una versión más reciente de las librerías es importante asegurarse y revisar que todas las funcionalidades necesarias son compatibles hacia atrás. Al hacer en el caso de jQuery una actualización a una versión mayor hay que probar y leer las notas de publicación de las versiones por si hubiera un problema de compatibilidad en las funcionalidades que requiere de ella el framework Apache Tapestry.

Capítulo 6

Formularios

La sangre de cualquier aplicación son los datos y en una aplicación web en gran parte los datos de entrada recogidos en formularios en el navegador. Ya se trate de un formulario de búsqueda, una pantalla de inicio de sesión o un asistente de registro multipágina los formularios son la forma de comunicarse con la aplicación. Tapestry destaca en la creación de formularios y en la validación de datos. La validación de los datos de entrada es declarativa lo que significa que tu simplemente indicas que validaciones aplicar a un determinado campo y Tapestry realiza todo el trabajo en el servidor (una vez implementado) y en el cliente si se quiere. Finalmente, Tapestry es capaz no solo de presentar los errores de vuelta a un usuario sino de decorar los campos y las etiquetas de los campos marcándolos como que contienen errores, principalmente usando clases CSS.

6.1 Eventos del componente Form

El componente Form emite una serie de eventos de componente. Necesitarás proporcionar manejadores de eventos para algunos de estos. Al renderizarse el componente de formulario emite dos notificaciones: primero prepareForRender y luego prepare. Esto permite al contenedor del componente Form configurar cualesquiera campos o propiedades que serán referenciadas en el formulario. Por ejemplo, este es un buen lugar para crear un objeto temporal usado en la renderización o cargar una entidad de la base de datos para ser editada.

Cuando el usuario envía el formulario del cliente ocurren una serie de pasos en el servidor. Primero, el Form emite una notificación prepareForSubmit, luego una notificación prepare. Estos permiten al contenedor asegurar que los objetos está configurados y listos para recibir la información enviada en el formulario. A continuación, todos los campos dentro del formulario son activados para asignar los valores de la petición, validarlos y si son válidos almacenar los valores en las propiedades de los objetos asociadas a los elementos del formulario, esto significa que no tendremos que recuperar los datos de la request, hacer una conversión a su tipo correcto y posteriormente almacenarlo en la propiedad que deseemos, de todo esto se encarga Tapestry. Después de que los campos han hecho su procesado, el Form emite un evento validate. Esta es una oportunidad para realizar las validaciones que no pueden ser descritas declarativamente como validaciones de varios campos dependientes. Seguidamente, el Form determina si ha habido errores de validación, si los ha ha habido, el envió del formulario es considerado un error y se emite un evento failure. Si no ha habido errores de validación entonces se emite

un evento success. Finalmente, el formulario emite un evento submit para casos en los que no importa si se produce un success o un failure.

- prepareForRender: Evento de la fase render. Antes de realizar el renderizado permite cargar una la entidad a editar.
- prepare: Evento de la fase render. Antes de renderizar el formulario pero después del evento prepareForRender.
- prepareForSubmit: Evento de la fase submit. Antes de que el formulario enviado sea procesado.
- prepare: Evento de la fase submit. Antes del que el formulario enviado sea procesado pero después de prepareForSubmit.
- validate: Evento de la fase submit. Después de que los campos hayan sido rellenados con los valores enviados y validados. Permite hacer cualquier otra validación.
- failure: Evento de la fase submit. Después de que hayan ocurrido uno o más errores de validación.
- success: Evento de la fase submit. Cuando la validación se ha completado sin errores. Permite salvar los cambios de las entidades en la base de datos.
- submit: Evento de la fase submit. Después de que la validación haya ocurrido con éxito o con errores.

6.2 Seguimiento de errores de validación

Asociado con el formulario hay un objeto ValidationTracker que rastrea todos los datos proporcionados por el usuario y los errores de validación para cada campo en el formulario. El rastreador puede proporcionarse al Form mediante el parámetro tracker pero es raramente necesario.

El Form incluye los métodos isValid() y getHasErrores() que son usados para ver si el rastreador de validación del formulario contiene algún error. En tu propia lógica es posible grabar tus propios errores para ello el formulario incluye dos versiones del método recordError(), una que especifica un Field (una interfaz implementada por todos los elementos componentes de formulario) y otra que es para errores globales que no están asociados con ningún campo en particular.

6.3 Almacenando datos entre peticiones

Como en otras acciones de peticiones el resultado de un envió de formulario (excepto en las Zones) es enviar una redirección al cliente lo que resulta en una segunda petición (para renderizar la página). El ValidationTraker debe ser persistido (generalmente en el objeto HttpSession) entre las dos peticiones para prevenir la pérdida de la información de validación. Afortunadamente, el ValidationTraker por defecto proporcionado por el formulario es persistente de modo que normalmente no necesitas preocuparte por ello. Sin embargo, por la misma razón los campos individuales actualizados por los componentes de formulario deberían ser persistidos entre peticiones y esto es algo que debes hacer tú, generalmente con la anotación @Persist.

Nota: desde la versión 5.4 este comportamiento para los errores de validación ha cambiado. En esta versión el renderizado de la página ocurren en la misma petición en vez de emitir una redirección cuando hay errores. Esto elimina la necesidad de usar un campo persistente en la sesión para almacenar el rastreador de validación cuando un error de validación ocurra, lo que permite evitar crear sesiones.

Por ejemplo, una página de inicio de sesión, que recoge un nombre de usuario y una contraseña podría ser como:

```
public class Login {

  @Persist
  @Property
  private String usuario;

  @Property
  private String password;

  @Inject
  private UserAuthenticator authenticator;

  @InjectComponent(id = "password")
  private PasswordField passwordField;

  @Component
  private Form form;

  /**
   * Hacer validaciones personalidas
   */
  void onValidateFromForm() {
    if (!authenticator.isValid(usuario, password)) {
      // almacenar un error, y prevenir que Tapestry emita un evento success
      form.recordError(passwordField, "Usuario o contraseña inválidos.");
    }
  }

  /**
   * Sin errores de validación, rederigir a la página post autenticación.
   */
  Object onSuccess() {
    return PostLogin.class;
  }
}
```

Dado que el envió del formulario realmente consiste en dos peticiones, el propio envió (que resulta en una respuesta de redirección) y luego una segunda petición para la página (que resulta en un renderizado de la página) es necesario persistir el campo del nombre del usuario entre las dos peticiones usando la anotación

@Persist. Esto sería necesario también para el campo contraseña excepto que el componente PasswordField nunca renderiza su valor.

Para evitar perdida de datos los valores de los campos almacenados en la HttpSession deben ser serializables, particularmente si quieres usar un cluster para tu aplicación o preservar las sesiones entre reinicios del servidor. El Form solo emite un evento success si no hay errores de validación, esto significa que no es necesario escribir:

```
1 if (form.getHasErrors())
    return;
```

Finalmente, nota como la lógica de negocio encaja en la validación. El servicio UserAuthenticator es responsable de asegurar que el nombre del usuario y la contraseña son válidas. Cuando retorna false se le indica al Form que registre un error. Se le proporciona una instancia de PasswordField como primer parámetro esto asegura que el campo contraseña y su etiqueta sean decorados cuando el formulario se renderice para presentar los errores al usuario.

6.4 Configurando campos y etiquetas

La plantilla para la página de inicio de sesión contiene una mínima cantidad de instrumentación:

```
1  <html t:type="layout" t:titulo="Inicio de sesión" xmlns:t="http://tapestry.apache.org/
       schema/tapestry_5_4.xsd" xmlns:p="tapestry:parameter">
   <div class="col-md-4 col-md-offset-4">
3    <img src="${context:images/tapestry.png}" alt="Apache Tapestry" title="Apache Tapestry"
        />

     <t:form t:id="form" clientValidation="none">
       <t:errors class="literal:alert alert-danger" />

8      <div class="form-group">
         <t:label for="usuario"/>
         <input t:type="TextField" t:id="usuario" t:validate="required,minlength=3" label="
       Usuario" />
       </div>
       <div class="form-group">
13       <t:label for="password" />
         <input t:type="PasswordField" t:id="password" t:validate="required,minlength=3"
       label="Contraseña" />
       </div>
       <div class="btn-toolbar">
         <input type="submit" class="btn btn-primary" value="Iniciar sesión" />
18     </div>
     </t:form>
```

```
</div>
</html>
```

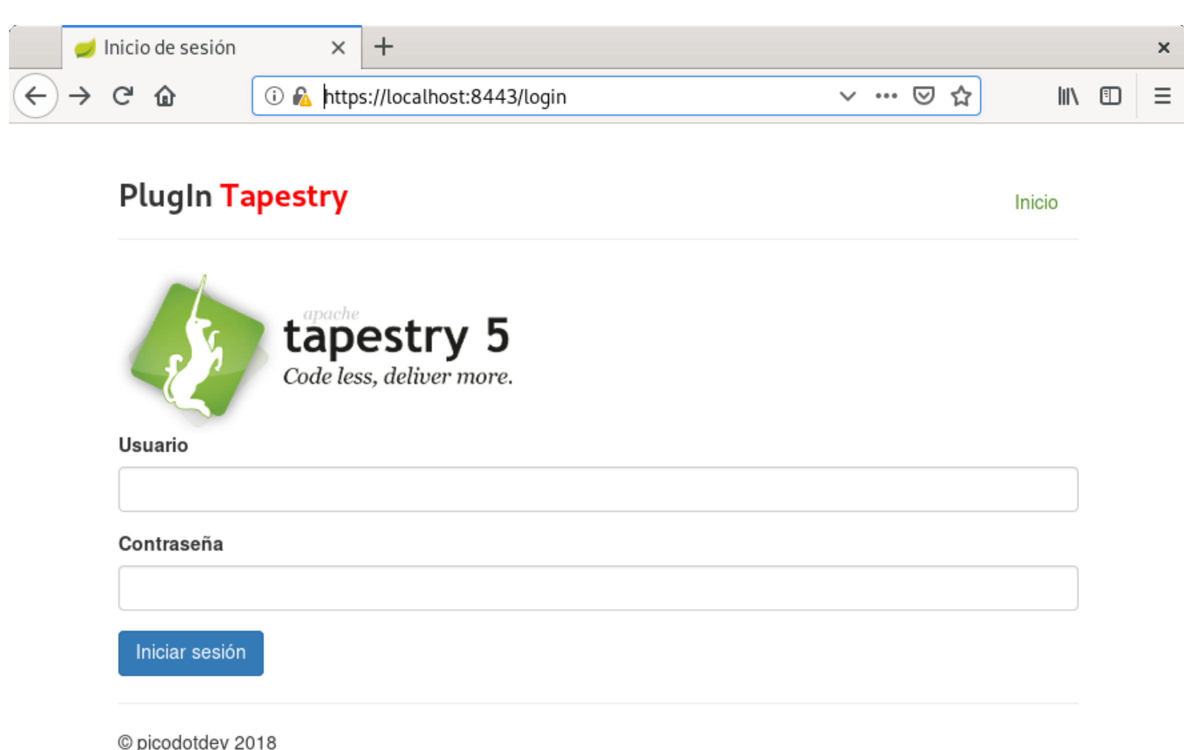
El componente Form es responsable de crear la URL necesaria para el envió del formulario (esto es responsabilidad de Tapestry no tuya). El componente Errors debe ser colocado dentro de un Form, mostrará todos los errores para los campos dentro del Form como una lista, usa unos estilos simples para hacer el resultado más presentable. Cada componente de campo como TextField es emparejado con su etiqueta Label. El Label renderizará un elemento label conectado al campo, esto es muy importante para la usabilidad, especialmente para usuarios con discapacidades visuales. También significa que puedes hacer clic en la etiqueta para mover el cursor al campo correspondiente. El parámetro for del Label es el id del componente con el que se quiere asociar.

Para el TextField, proporcionamos un id de componente, usuario. Podemos especificar el parámetro value pero por defecto se coge la propiedad del contenedor, la página Login, que coincida con el id del componente si esa propiedad existe. Omitir el parámetro value ayuda a mantener la plantilla más clara aunque menos explícita.

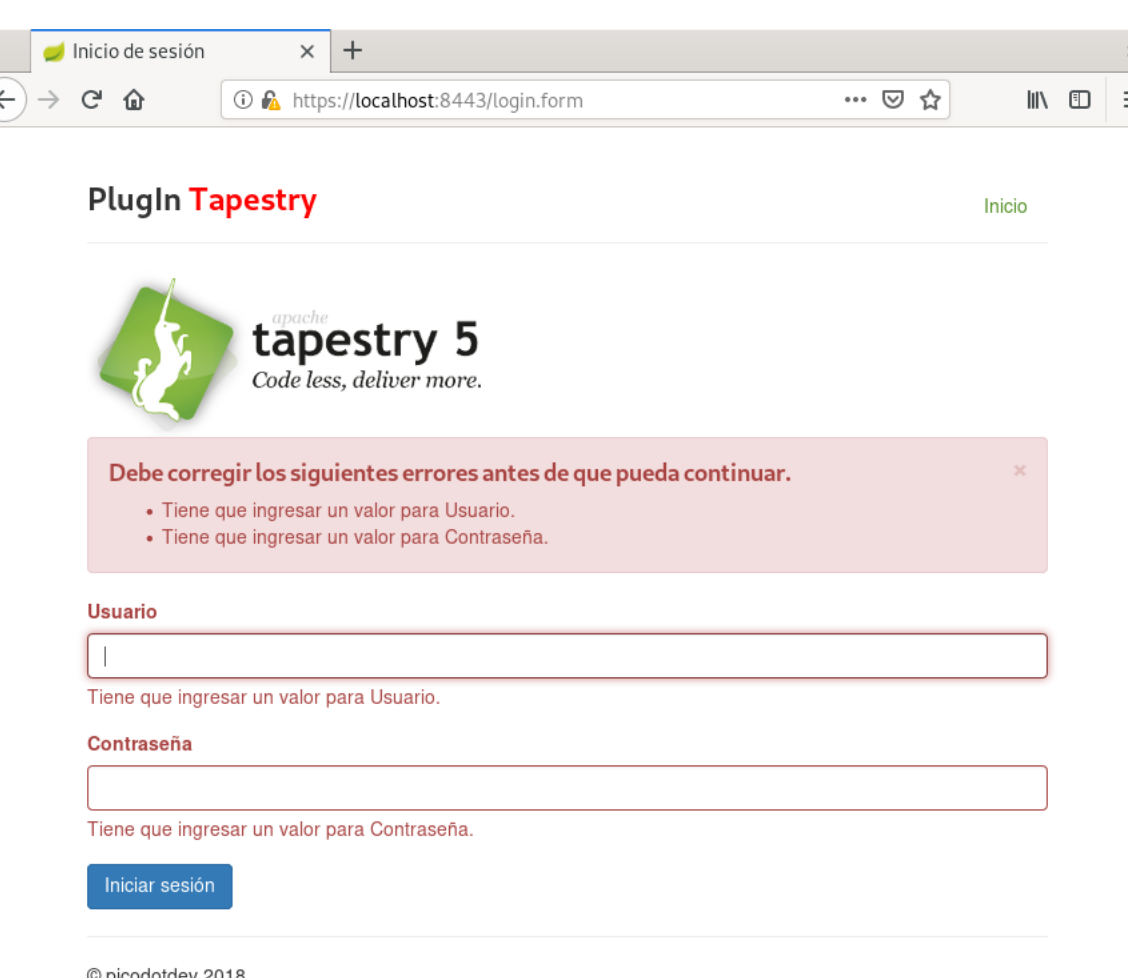
El parámetro validate identifica que validaciones deberían ocurrir para el campo. Esta es una lista de nombres de validadores. Los validadores pueden ser configurados y la lista de validadores disponibles es extensible. required es el nombre de uno de los validadores integrados que asegura que el valor enviado no es una cadena vacía y minlen asegura que el valor tenga una longitud mínima. El parámetro validate es indicado en el namespace t: esto no es estrictamente necesario sin embargo ponerlo asegura que la plantilla sea xhtml válido.

6.5 Errores y decoraciones

Cuando activas una página por primera vez los campos y formularios se renderizarán normalmente vacíos esperando datos:

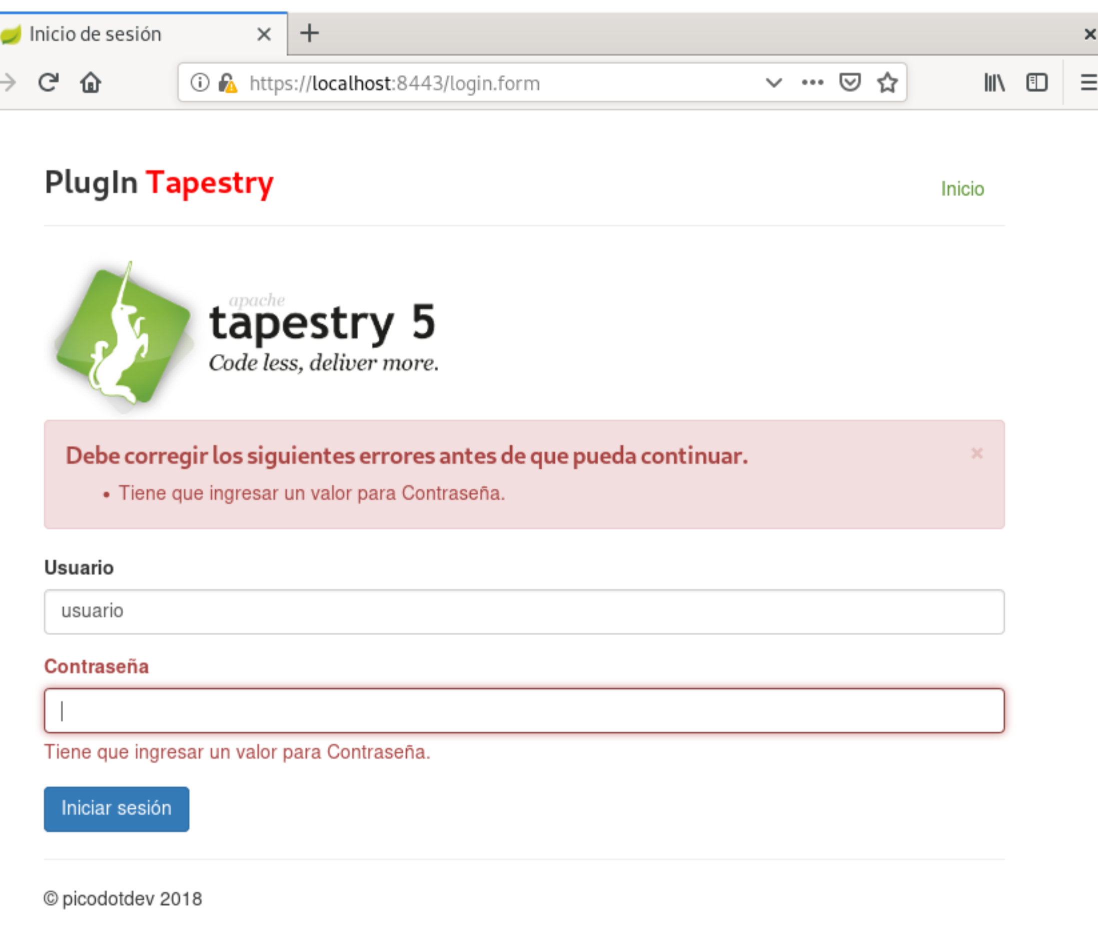


Si el formulario se envía como está los campos no cumplirán la restricción required y la página se mostrará de nuevo para presentar esos errores al usuario:



Han ocurrido una serie de cosas sutiles aquí. Primero, Tapestry registra todos los errores para todos los campos. El componente Errors los ha mostrado encima del formulario. Además el decorador de validación por defecto ha añadido decoraciones a las etiquetas y los campos, añadiendo algo de clases CSS a los campos y etiquetas para marcar los campos con error.

Si escribes un usuario y contraseña incorrectos la lógica dentro de la página de Login añadirá errores a los campos:



Esto esta muy bien y mantiene la coherencia ya que se mantiene el mismo comportamiento y estilo visual para ambos tipos de errores, para los indicados en los atributos t:validate y para los generados mediante lógica de aplicación.

6.6 Validación de formularios

6.6.1 Validadores disponibles

Estos son los validadores integrados en Tapestry:

- email: Asegura que el valor del campo sea una dirección de correo electrónico

```
1 <t:textfield value="email" validate="email" />
```

- max (long): Asegura un valor máximo entero.

```
1 <t:textfield value="age" validate="max=120,min=0" />
```

- maxLength (int): Asegura que un valor String tenga un máximo de caracteres.

```
1 <t:textfield value="zip" validate="maxlength=7" />
```

- min (long): Asegura un valor mínimo entero.

```
1 <t:textfield value="age" validate="max=120,min=0" />
```

- minLength (int): Asegura que un valor String tenga un mínimo de caracteres.

```
1 <t:textfield value="somefield" validate="minlength=1" />
```

- none: No hace nada, usado para sobrescribir la validación @Validate.

```
1 <t:textfield value="somefield" validate="none" />
```

- regexp pattern: asegura que un valor String cumpla el patrón de una expresión regular.

```
1 <t:textfield value="letterfield" validate="regexp=^[A-Za-z]+$" />
```

- required: Asegura que un valor String no sea nulo y no sea una cadena vacía.

```
1 <t:textfield value="name" validate="required" />
```

6.6.2 Centralizando la validación

La anotación @Validate puede tomar el lugar del parámetro de validación del TextField, PasswordField, TextArea y otros componentes. Cuando el parámetro validate no esta indicado el componente comprobará la anotación @Validate y lo usa como la definición de validación. La anotación puede ser colocada en los métodos getter y setter o en una propiedad.

De esta forma las validaciones no hace falta ponerse en todas las plantillas de las páginas en las que se quiera validar las propiedades de una entidad, sino que pueden estar centralizadas en la entidad a validar. Las anotaciones de validación de entidades también están soportadas.

```java
package io.github.picodotdev.plugintapestry.entities;

...

@Entity
public class Producto implements Serializable {

  private static final long serialVersionUID = 4301591927955774037L;

  @Id
  @GeneratedValue
  private Long id;

  @NotNull
  @Length(min = 3, max = 100)
  @Column(name = "nombre", length = 100)
  private String nombre;

  @NotNull
  @Length(min = 0, max = 5000)
  @Column(name = "descripcion", length = 5000)
  private String descripcion;

  @NotNull
  @Min(value = 0)
  @Max(value = 1000)
  @Column(name = "cantidad")
  private Long cantidad;

  @NotNull
  @Column(name = "fecha")
  private Date fecha;

  ...
}
```

6.6.3 Personalizando los errores de validación

Cada validador (como required o minlength) tiene un mensaje por defecto cuando la restricción es violada, esto es, cuando el valor no es valido.

El mensaje puede ser personalizado añadiendo una linea en el catálogo de mensajes de la página (o en el del componente). Como cualquier propiedad localizada, esto puede ponerse también en el catálogo de mensajes de la aplicación.

La primera clave comprobada es formId-fieldId-validatorName-message. Donde cada parte se corresponde con:

- formId: el id local del componente Form.
- fieldId: el id local del componente de campo (usuario, ...).
- validatorName: el nombre del validador (required, minlength, ...).
- message: cadena literal para indicar que se trata de un mensaje de validación.

En el ejemplo de inicio de sesión si quisiésemos mostrar un mensaje personalizado para la validación required del campo usuario la clave a incluir en el catálogo de mensajes sería form-usuario-required-message.

Si no hay mensaje para esa clave se realiza una segunda comprobación para fieldId-validatorName-message. Si eso no coincide con un mensaje, entonces se usa el mensaje incorporado por defecto para el validador.

6.6.4 Configurar las restricciones de validación en el catálogo de mensajes

Es posible omitir la restricción de validación del parámetro validate en cuyo caso se espera que este almacenado en el catálogo de mensajes. Esto es útil cuando es incómodo que la restricción de validación se incluya directamente en la plantilla, como una expresión regular para usar con el validador regexp. La clave para esto es similar a personalizar el catálogo de mensajes: formId-fieldId-validatorName o solo fieldId-validatorName. Por ejemplo, tu plantilla puede tener lo siguiente:

```
1 <t:textfield t:id="snn" validate="required,regexp"/>
```

Y tu catálogo de mensajes puede contener:

```
1 ssn-regexp=\d{3}-\d{2}-\d{4}
  ssn-regexp-message=Social security numbers are in the format 12-34-5678.
```

Esto también es util cuando la expresión regular a aplicar depende del idioma como podría ser el caso de las fechas (dd/MM/yyyy para los españoles o MM/dd/yyyy para los ingleses).

6.6.5 Macros de validación

Los validadores puede combinarse en macros. Este mecanismo es conveniente para asegurar una validación consistente en toda la aplicación. Para crear una macro de validación simplemente contribuye al servicio ValidationMacro en tu clase de módulo añadiendo una nueva entrada al objeto de configuración, tal y como se muestra a continuación. El primer parámetro es el nombre de la macro y el segundo es una lista separada por comas de validadores::

```
public static void contributeValidatorMacro(MappedConfiguration<String, String>
    configuration) {
  configuration.add("password","required,minlength=5,maxlength=15,");
}
```

De este modo puedes usar esta nueva macro en tus plantillas de componente y clases:

```
<input t:type="textField" t:id="password" t:validate="password" />
```

```
@Validate("password")
private String password;
```

6.7 Subiendo archivos

Tapestry proporciona un componente de subida de archivos basado en Apache Commons FileUpload para hacer fácil el manejo de los archivos subidos a través de formularios web usando el elemento estándar <input type="file">. El módulo que lo contiene, tapestry-upload, no está automáticamente incluido en Tapestry dadas las dependencias adicionales que requiere. Para incluirlo añade la dependencia de tapestry-upload a tu aplicación, algo como esto usando Gradle:

```
compile 'org.apache.tapestry:tapestry-upload:5.4'
```

El componente upload soporta el binding por defecto (basado en id) y validación.

```
<t:form>
  <t:errors/>

  <input t:type="upload" t:id="file" t:value="file" validate="required"/><br/>
  <input type="submit" value="Upload"/>
</t:form>
```

```
public class EjemploUpload {

  @Property
  private UploadedFile file;
```

```
  public void onSuccess() {
    File temp = File.createTempFile("temp", null);
    file.write(temp);
9   }
}
```

Excepciones de subida

En algunos casos las subidas de archivos pueden fallar. Esto puede suceder por una simple excepción de comunicación o más probablemente porque el tamaño máximo de archivo ha sido excedido. Cuando ocurre una excepción de subida de archivo, Tapestry producirá un evento UploadException en la página para notificar el error. Todo el proceso normal es saltado (no hay evento activate ni envió de formulario, ...).

El manejador de evento debería retornar un objeto no nulo, el cual será tratado como el resultado de navegación:

```
1 @Persist(PersistenceConstants.FLASH)
  @Property
  private String mensaje;

5 Object onUploadException(FileUploadException ex) {
    mensaje = "Excepción de subida de archivo: " + ex.getMessage();
    return Pagina.class;
  }
```

Un método manejador de evento void o uno que retorne null resultará que la excepción sea reportada al usuario como una excepción no capturada en tiempo de ejecución.

Configuración

Se pueden configurar cuatro símbolos de configuración relacionados con la subida de archivos:

- upload.repository-location: El directorio al que serán escritos los archivos que son demasiado grandes para mantenerlos en memoria. El valor por defecto es java.io.tmpdir.
- upload.repository-threshold: Tamaño en bytes, a partir de los que los archivos serán escritos al disco en vez de a memoria. El valor por defecto son 10 KiB.
- upload.requestsize-max: Tamaño máximo, en bytes, para la petición completa. Si es excedido se producirá una excepción FileUploadException. No hay máximo por defecto.
- upload.filesize-max: Tamaño máximo, en bytes, para un archivo individual. De nuevo, se producirá una excepción FileUploadException si se excede. No hay máximo por defecto.

La clase UploadSymbols define constantes para estos cuatro símbolos.

6.8 Conversiones

Las clases que implementan la interfaz Translator en Tapestry permiten convertir el valor de un campo de texto a un objeto (a través del método parseClient) y de un objeto a un texto que será incluido en un elemento de formulario en el cliente (a través del método toClient). Esta conversión es necesaria ya que lo que enviamos al cliente y lo que recibimos de él es una cadena. Al recibir los datos desde cliente en el servidor necesitaremos alguna forma de convertir esos datos representados en formato texto a su representación en objeto que hagamos en el servidor. Estas dos tareas que en un principio no son muy complejas son tremendamente necesarias y básicas en cualquier aplicación web, siendo algo básico el framework que usemos debería dar un buen soporte a estas tareas. Con los translators podremos evitar repetirnos en diferentes puntos de la aplicación haciendo constantemente las mismas conversiones.

Una vez que tengamos definido el translator, Tapestry buscará el adecuado según el tipo de objeto a traducir y lo usará según sea necesario sin necesidad de que tengamos que hacer nada más. Vamos a ver un ejemplo, supongamos que en un campo de un formulario necesitamos mostrar una fecha con un determinado formato. En nuestras clases trabajaremos con objetos de tipo Date. El usuario deberá introducir la fecha con formato «dd/MM/yyyy».

```
1  package io.github.picodotdev.plugintapestry.misc;
2
   ...

   public class DateTranslator extends AbstractTranslator<Date> {

7    private String patron;

     public DateTranslator(String patron) {
       super("date", Date.class, "date-format-exception");
       this.patron = patron;
12   }

     @Override
     public String toClient(Date value) {
       if (value == null) {
17       return null;
       }
       // Convertir el objeto date a su representación en String utilizando un patrón de
       fecha.
       return new SimpleDateFormat(patron).format(value);
     }
22
     @Override
     public Date parseClient(Field field, String clientValue, String message) throws
       ValidationException {
       if (clientValue == null) {
         return null;
```

```
27      }
        try {
          // Convertir la represetación del objeto fecha en String a su representación en
        objeto Date.
          return new SimpleDateFormat(patron).parse(clientValue);
        } catch (ParseException e) {
32        throw new ValidationException(MessageFormat.format(message, field.getLabel()));
        }
      }

      @Override
37    public void render(Field field, String message, MarkupWriter writer, FormSupport
        formSupport) {
      }
    }
```

Estamos extendiendo una clase del paquete org.apache.tapestry5.internal que es algo no recomendado pero la utilizamos por sencillez y para no tener que implementar nosotros lo que hace el propio AbstractTranslator. Para que Tapestry lo utilice deberemos hacer una contribución en el módulo de nuestra aplicación donde básicamente decimos que para una determinada clase se utilice un determinado Translator.

Listado 6.1: AppModule.java

```
1 public static void contributeTranslatorSource(MappedConfiguration configuration) {
    configuration.add(Date.class, new DateTranslator("dd/MM/yyyy"));
  }
```

A partir de este momento podríamos tener en archivo .tml de una página o componente lo siguiente y en la propiedad fecha del componente o página tendríamos un objeto de tipo Date olvidándonos por completo de la traducción que hará Tapestry por nosotros.

```
1 <t:label for="fecha"/>: <t:textfield t:id="fecha" value="fecha" size="12" label="Fecha"/>
```

Hay otra interfaz que hacen algo similar a los Translators, es la interfaz ValueEncoder pero la diferencia entre las dos está en que en los translators puede ser necesaria algún tipo de validación por nuestra parte ya que son datos que introduce el usuario y en los encoders no ya que no son datos que introduce el usuario. Esto se ve claramente en los parámetros de los componentes TextField, Hidden y Select, el primero utiliza un Translator y los dos últimos un ValueEncoder.

Tapestry ya proporciona un ValueEncoder para las entidades de nuestro dominio si utilizamos Hibernate, la clase es HibernateEntityValueEncoder, Tapestry se encargará de insertar en el valor del hidden el id de la entidad y cuando el formulario sea enviado al servidor de recuperar la entidad:

```
<t:hidden t:id="producto" value="producto"/>
```

Para terminar, indicar que Tapestry también proporciona un ValueEncoder por defecto para los tipos Enum.

Capítulo 7

Internacionalización (i18n) y localización (l10n)

En determinadas regiones donde existen varias lenguas oficiales o en un mundo globalizado donde cualquier persona del planeta con acceso a internet puede acceder a cualquier página de la red sin importar las distancias es necesario que las aplicaciones generen su contenido en el lenguaje que el usuario entiende o prefiere.

El proceso de hacer que el mismo código de una aplicación genere el idioma deseado por el usuario se llama internacionalización (i18n). El proceso de traducir la aplicación a un nuevo lenguaje se llama localization (l10n).

El soporte para ambas cosas está integrado en Tapestry permitiéndote separar fácilmente el texto que se presenta a los usuarios según su preferencia de idioma del código que lo genera, tanto del código Java como de las plantillas de los componentes pero también ofrece soporte para internacionalizar imágenes que incluyan texto. En el código JavaScript también deberemos buscar una solución para la internacionalización.

Las necesidades básicas de internacionalización y localización son:

- Selección del recurso que contiene los mensajes localizados según el idioma.
- Soporte para interpolación de variables en los mensajes. Los mensajes pueden contener variables que son sustituidas en el momento de su procesamiento por parámetros con los valores adecuados a mostrar.
- Soporte para múltiples formas plurales, el español tiene solo dos, singular y plural pero otros idiomas tienen más. Si en una aplicación ves las típicas eses entre paréntesis, (s), o cuando aparece un mensaje tal que «1 elementos» es porque esa aplicación aunque esté internacionalizada no soporta las múltiples formas plurales, para un usuario ver esos (s) o hablar en plural de sujetos que debería se singulares crea confusión y dificulta la lectura del texto.

7.1 Catálogos de mensajes

Los mensajes localizados se guardan en catálogos de mensajes y en Tapestry cada componente puede tener el suyo propio. Los catálogos de mensajes son un conjunto de archivos properties localizados en la misma car-

peta del archivo compilado del código Java. Los ficheros de propiedades no son mas que un ResourceBundle con un par clave=valor por cada linea. Los valores se acceden por las claves y no son sensibles a mayúsculas y minúsculas.

Por ejemplo, si tuviésemos un componente io.github.picodotdev.plugintapestry.components.Componente el archivo properties estaría ubicado en io/github/picodotdev/plugintapestry/components/Componente.properties. Si el componente estuviese localizado en alemán el archivo se llamaría Componente_de.properties siguiendo los códigos ISO para los lenguajes. Las claves definidas en los archivos más específicos sobrescriben los mensajes más globales. Si el componente tuviese una archivo Componente_de_DE.properties las claves de este archivo sobrescribirían las definidas en Componente_de.properties.

Los catálogos de mensajes son leídos con la codificación de caracteres UTF-8 con lo que no será necesario que usemos la herramienta native2ascii.

Herencia de catálogos de mensajes

Si un componente hereda de otro este heredará también el catálogo de mensajes pudiendo sobrescribir ciertos mensajes por otros más específicos.

7.1.1 Catálogo global de la aplicación

La aplicación puede tener un catálogo de mensajes global para todos los componentes de la aplicación. La convención de la ubicación para este archivo es en WEB-INF y su nombre está derivado del nombre del filtro de Tapestry. Si el filtro se llamase app la ubicación del catálogo sería WEB-INF/app.properties.

Teniendo un catálogo global para toda la aplicación o específicos para cada componente nos proporciona mucha flexibilidad. Podemos crear los catálogos de mensajes como prefiramos. Con el catálogo global tendremos un único archivo grande y con catálogos de mensajes por componente tendremos varios pero más pequeños y controlados. También podemos optar por combinar ambas posibilidades usando catálogos específicos para ciertos componentes y usar el catálogo global para el resto.

7.1.2 Accediendo a los mensajes localizados

Ahora que sabemos la teoría veamos de que forma podemos acceder a los mensajes localizados en código. Los mensajes pueden ser accedidos de dos formas: Usando el binding «message» o la expansión de expresiones en las plantillas de los componentes. En las plantillas usaríamos el siguiente código:

Listado 7.1: Pagina.tml

```
1 <a t:type="any" href="http://www.google.es" alt="prop:alt" title="message:title">
    ${message:Accede_a_Google}
  </a>
```

El catálogo de mensajes tendría en español:

Listado 7.2: Pagina.properties

```
alt=Buscar en %s
results=0#Hay {0} elementos.|1#Hay {0} elemento.|1<Hay {0} elementos.
```

El catálogo localizado en inglés sería:

Listado 7.3: Pagina_en.properties

```
alt=Search in %s
results=0#There are {0} elements.|1#There is {0} element.|1<There are {0} elements.
```

En el código Java usaremos la anotación @Inject para acceder al catálogo de mensajes:

```java
@Inject
private Messages messages;

public String getAlt() {
  return messages.format("alt", "Google");
}
```

El método format usa la clase java.util.Formatter para formatear los mensajes al estilo de printf. En caso de acceder a una clave que no existe en el catálogo de mensajes en vez de lanzar una excepción se generará un sustituto del estilo [[missing key: key-not-found]]. Los catálogos de mensajes serán recargados en caliente por lo que veremos los cambios inmediatamente simplemente con actualizar el navegador.

Los mensajes que soporten múltiples formas plurales deben ser tratados de forma diferente, aunque Tapestry no los soporta de por si con la clase Messages en Java podemos manejarlos con la clase ChoiceFormat. Lo primero sería obtener el mensaje sin indicar parámetros ni formas plurales, los segundo sería obtener la forma plural adecuada y finalmente realizar la interpolación de variables.

Listado 7.4: Index.java

```java
package io.github.picodotdev.plugintapestry.pages;

...

/**
 * @tapestrydoc
 */
public class Index {

  ...
```

```
    @Inject
    private Messages messages;
14
    ...

    public String getMensajeFormasPlurales(String key, long num) {
      // Comprobar si existe, devolver el placeholder si no se encuentra
19        if (!messages.contains(key)) {
            return messages.get(key);
        }

        // Obtener el mensaje con las diferentes formas plurales
24        String message = messages.get(key);

        // Seleccionar la forma plural adecuada del mensaje
        ChoiceFormat format = new ChoiceFormat(message);
        String pluralized = format.format(num);
29
        // Realizar la interpolación de variables
        return MessageFormat.format(pluralized, num);
    }
}
```

Listado 7.5: Index.tml

```
1 <t:outputraw value="getMensajeFormasPlurales('
      Este_mensaje_esta_localizado_formas_plurales', 0)"/>
2 <t:outputraw value="getMensajeFormasPlurales('
      Este_mensaje_esta_localizado_formas_plurales', 1)"/>
  <t:outputraw value="getMensajeFormasPlurales('
      Este_mensaje_esta_localizado_formas_plurales', 2)"/>
```

Por supuesto para evitar duplicar estas varias lineas en cada componente para cada mensaje pluralizado es recomendable añadirlas en algún método de utilidad.

7.2 Imágenes

Algunas imágenes llevan texto pero este no puede ser modificado salvo proporcionando otra imagen. Afortunadamente Tapestry busca la imagen más específica disponible al igual que se hace con los catálogos de mensajes. Así podemos tener: imagen.png para español e imagen_de.png para alemán, en función del locale preferido por el usuario se usará una u otra.

7.3 Selección del locale

El locale de cada petición es determinado por las cabeceras HTTP enviadas por el navegador del usuario. Puede ser más (en_GB) o menos específico (en). En las aplicaciones de Tapestry se especifican los locales soportados y Tapestry se encarga de convertir el locale solicitado a la mejor opción de entre las soportadas. Los locales soportados se especifican con el símbolo tapestry.supported-locales. Por ejemplo, una petición con el locale fr_FR coincidiría con fr pero no con de. Si no se encuentra una coincidencia se usa el locale por defecto que es el primero de los locales soportados y especificado en el símbolo tapestry.supported-locales.

El cambio de un locale a otro se hace a través del servicio PersistentLocale.

```
@Inject
private PersistentLocale persistentLocale;

void setLocale(Locale locale) {
  persistentLocale.set(locale);
}

public String getDisplayLanguage() {
  return persistentLocale.get().getDisplayLanguage();
}
```

Cuando se cambia el locale este es incluido en el path de las URL generadas de forma que persistirá entre petición y petición y el usuario podrá guardar el enlace en sus marcadores. Cuando se cambia el locale este no se refleja hasta el siguiente ciclo de renderizado de una página.

Los locales soportados por defecto son los siguientes, en caso de necesitar uno no incluido en esta lista podría ser añadido:

- en (English)
- es (Spanish)
- ja (Japanese)
- pt (Portuguese)
- zh (Chinese)
- bg (Bulgarian)
- fi (Finnish)
- mk (Macedonian)
- ru (Russian)
- da (Danish)

- fr (French)
- nl (Dutch)
- sr (Serbian)
- de (German)
- hr (Croatian)
- no (Norwegian)
- sv (Swedish)
- el (Greek)
- it (Italian)
- pl (Polish)
- vi (Vietnamese)

7.4 JavaScript

Teniendo el lado del cliente cierta complejidad quizá nos encontremos con la necesidad de proporcionar internacionalización (i18n) para los textos o mensajes mediante una librería JavaScript. Una de la que más me ha gustado de las que he encontrado ha sido i18next pero hay varias opciones más, incluidas dos que merecen ser nombradas que son polyglot y messageformat, estas tres opciones son parecidas pero no tienen exactamente las mismas funcionalidades, deberemos evaluarlas para elegir una según lo que necesitemos.

En el caso de i18next los archivos de literales son poco más que una relación de claves valor similar a los archivos properties de Java aunque en el caso de i18next se definen en archivos con formato json. Por cada idioma localizado necesitamos crea un archivo con los literales:

Listado 7.6: translation-dev.json

```
{
"Lista_de_tareas": "Lista de tareas",
"COMPLETADAS_tareas_completadas_de_TOTAL": "__completadas__ tarea completada de __total__
    ",
"COMPLETADAS_tareas_completadas_de_TOTAL_plural": "__completadas__ tareas completadas de
    __total__",
"Muy_bien_has_completado_todas_las_tareas": "¡Muy bien! has completado todas las tareas",
"Limpiar": "Limpiar",
"Introduce_una_nueva_tarea": "Introduce una nueva tarea" }
```

Listado 7.7: translation-en.json

```
{
"Lista_de_tareas": "Tasks List",
"COMPLETADAS_tareas_completadas_de_TOTAL": "__completadas__ task completed of __total__",
"COMPLETADAS_tareas_completadas_de_TOTAL_plural": "__completadas__ tasks completed of __total__",
"Muy_bien_has_completado_todas_las_tareas": "¡Perfect! You have done all tasks",
"Limpiar": "Clean",
"Introduce_una_nueva_tarea": "Type a new task"
}
```

Para obtener un mensaje y otro con varias formas plurales hacemos:

```
i18n.t('Lista_de_tareas');
i18n.t('COMPLETADAS_tareas_completadas_de_TOTAL', { count: 1, completadas: 1, total: 4 });
```

7.5 Convenciones para archivos properties l10n

Te recomiendo seguir algunas convenciones para que la gestión de los literales no se convierta en algo difícil de mantener. Esto es independiente de si decides tener catálogos de mensajes por componente, a nivel global o una mezcla de ambos.

Lo ideal es que las claves de los literales sean semánticas para identificar el literal. Para los literales largos que son frases (mensajes, errores, confirmaciones, ...) probablemente podamos encontrar una clave semántica pero para los literales cortos formados por una, dos o tres palabras que serán los más comunes (etiquetas, acciones, botones, opciones de selección) en algunos casos nos resultará difícil encontrar algo semántico que lo identifique ya que a veces estos literales no tienen carga semántica.

Lo que se debe evitar es tener el mismo literal varias veces repetido y cada vez que debamos utilizarlo en un nuevo sitio de la aplicación tener que añadirlo de nuevo, por ejemplo, como en el caso de que los literales los agrupemos por la pantalla donde están, este puede ser el caso de literales cortos comunes de etiquetas y acciones como Nombre, Descripción, Aceptar, Cancelar, Eliminar, Buscar, ...

Un pequeño ejemplo de como organizar los archivos de literales podría ser el siguiente:

Listado 7.8: Literales.properties

```
# Literales no semánticos, ordenados alfabéticamente, respetando capitalización
Aceptar=Aceptar
Buscar=Buscar
Cancelar=Cancelar
Descripcion=Descripción
Eliminar=Eliminar
```

```
Nombre=Nombre

# Literales semánticos genéricos
error.valor_no_valido=El valor introducido no es válido
error.valor_demasiado_largo=El valor introducido es demasiado largo

confirmar.eliminar=¿Desea eliminar el elemento?

# Literales semánticos específicos de página
pagina1.confirmar.eliminar=¿Desea eliminar el repositorio?
pagina2.confirmar.eliminar=¿Desea eliminar el módulo?
```

Siguiendo estas u otras convenciones evitaremos que según vaya pasando el tiempo estos archivos se conviertan en un problema de mantenimiento, cuando en una aplicación tenemos unos miles de literales y varias decenas de idiomas, créeme puede llegar a serlo.

Capítulo 8

Persistencia en la capa de presentación

La persistencia a la que se refiere este capítulo es a la persistencia en la capa de presentación no a la persistencia de las entidades de dominio en una base de datos relacional o NoSQL.

8.1 Persistencia de página

En ocasiones las aplicaciones necesitan almacenar un poco de información para posteriores peticiones. La persistencia en una única página puede conseguirse mediante la anotación @Persist en el campo que queramos persistir entre peticiones. Se aplica de la siguiente forma:

```
1 @Persist
  private long valor;
```

Los campos anotados de esta forma mantiene su valor entre diferentes peticiones. Por defecto esto se consigue usando el objeto Session de las aplicaciones web en Java aunque hay otras estrategias de persistencia que no utilizan la sesión. Cuando el valor de la propiedad es cambiado se guarda al final de la petición, en siguientes peticiones el valor es recuperado en la misma propiedad de forma automática.

8.1.1 Estrategias de persistencia

La estrategia de persistencia define de que forma serán guardados y restaurados los valores entre peticiones.

Estrategia de sesión

Esta estrategia almacena los datos en la sesión indefinidamente mientras no se termine, la sesión se crea cuando es necesario. Es la estrategia que se usa por defecto a menos que se indique lo contrario.

```
public class MiPagina {

  @Persist
  private Integer id;
}
```

Estrategia flash

Esta estrategia también almacena los datos en sesión pero su tiempo de vida es muy corto. Solo dura hasta que los valores se restauran una vez lo que normalmente ocurre en la siguiente petición de la página. Puede usarse para guardar mensajes que solo se deben mostrar una vez como podrían ser unos errores de validación.

```
public class MiPagina {

  @Persist(PersistenceConstants.FLASH)
  private Integer value;
}
```

Estrategia de cliente

De forma diferente a las dos estrategias anteriores no guarda los datos en la sesión sino que lo hace añadiendo parámetros a los enlaces que se generan en la página o añadiendo campos ocultos en cada formulario. Es una estrategia que de la que no hay que abusar ya que la información viaja en cada petición entre el cliente y el servidor, guardar una información considerable de esta forma supondrá un coste extra de procesado en cada petición.

No hay que asustarse tampoco y por esto tampoco hay que no usarla, usándola con cuidado y almacenando una cantidad mínima de información es una buena opción y puede ser mejor que vernos obligados a crear una sesión para cada usuario. Intenta almacenar la clave primaria de las entidades antes que el objeto en sí.

```
public class MiPagina {

  @Persist(PersistenceConstants.CLIENT)
  private Integer value;
}
```

8.2 Valores por defecto

Las propiedades con la anotación @Persist no deberían tener valores por defecto ya sean definiéndolos en linea o dentro del constructor. Hacerlo de esta forma haría que la sesión se crease siempre que un usuario visitase la página y puede suponer un problema de escalabilidad.

Limpiando valores por defecto

Si conoces un momento en que los valores persistentes pueden ser descartados es posible hacerlo de forma programática usando el servicio ComponenteResources y su método descardPersistenFieldCanges(). Esto vale para cualquiera de las estrategias anteriores.

A tener en cuenta en clusters

La API de los servlets fue desarrollada pensando en que en la sesión solo se almacenaría una pequeña cantidad de información inmutable como números y cadenas. La mayoría de los servidores de aplicaciones realizan una serialización y lo distribuyen cuando se usa HttpSession.setAttribute().

Esto puede crear un problema de consistencia si se modifica un dato en la session, y no se invoca setAttribute(). El valor no será distribuido a otros servidores en el cluster. Tapestry soluciona este problema restaurando el valor de la sesión al inicio de la petición y lo guarda al final de la misma, esto asegura que todos los datos mutables sean distribuidos adecuadamente a todos los servidores del cluster. Pero a pesar de que esto soluciona el problema de la consistencia lo hace a costa de rendimiento ya que todas esas llamadas a setAttribute resultan en replicaciones innecesarias si el estado interno del objeto inmutable no ha cambiado. Tapestry también tiene soluciones para esto.

- Anotación ImmutableSessionPersistedObject: Tapestry conoce que un String, Number o Boolean son inmutables y no requieren un realmacenamiento en la sesión. Pero desconoce si cualquier otro objeto es en realidad inmutable, con esta anotación se le informa de ello y se evita la replicación.

- Interfaz OptimizedSessionPersistedObject: los objetos que implementan esta interfaz pueden controlar este comportamiento. Un objeto con esta interfaz puede monitorizar si ha cambiado y cuando se le pregunte responder si lo hizo desde la última vez que se le preguntó. Esto permite que solo cuando realmente haya cambiado se haga la replicación. Normalmente en vez de implementar la interfaz se extiende de la clase base BaseOptimizedSessionPersistedObject.

- Servicio de interfaz SessionPersistedObjectAnalyzer: el servicio SessionPersistedObjectAnalyzer es el responsable último de determinar cuando un objeto persistente de sesión se necesita replicar o no. Extendiendo este servicio se pueden implementar nuevas estrategias para nuevas clases.

8.3 Persistencia de sesión

Muchas aplicaciones necesitan almacenar algunos datos durante la navegación a través de varias páginas. Este podría ser el caso de un usuario que ha iniciado una sesión o un carrito de la compra.

La persistencia a nivel de página no es suficiente para casos como este ya que las propiedades persistentes solo está disponibles a nivel de esa página, no es compartida a través de múltiples páginas.

Objetos de estado de sesión

Con los objetos de estado de sesión (SSO, Session State Objects) los valores son almacenados fuera de la página siendo la estrategia de almacenamiento la sesión. La sesión es la misma para para todas las páginas del mismo usuario y diferente para diferentes usuarios.

Un campo que almacene un SSO debe marcarse con la anotación @SessionState.

```
1 public class MiPagina {

    @SessionState
    private CarritoCompra carritoCompra;
5 }
```

Cualquier otra página que declare una propiedad del mismo tipo, independientemente del nombre la propiedad y esté marcado con la anotación @SessionState compartirán el mismo valor, es así de simple. Solo hay que evitar NO usar SSO para tipos simples de (Boolean, String, Long, ...), solo se debe usar con clases propias creadas con este propósito.

La primera vez que se accede a un SSO se crea una sesión automáticamente. Asignar un valor a una propiedad SSO almacena el valor, asignarle null lo elimina. Normalmente un SSO tiene un constructor sin argumentos pero se le podrían inyectar dependencias tal y como se puede hacer con un servicio.

Controlar la creación

Las aplicaciones escalables no crean sesiones innecesariamente. Si puedes evitar crear sesiones, especialmente en la primera visita a la aplicación, podrás soportar un mayor número de usuarios. De manera que si puedes evitar crear la sesión deberías hacerlo.

¿Pero como evitarlo? Simplemente por el hecho de acceder a la propiedad haciendo «carritoCompra != null» forzaría la creación del SSO y la sesión para almacenarlo. Se puede forzar a que el SSO no se cree automáticamente:

```
1 public class MiPagina {

    @SessionState(create=false)
    private CarritoCompra carritoCompra;
5 }
```

En este caso carritoCompra será null hasta que se le asigne un valor y tendrá un valor si se le ha asignado alguno o está definido el SSO en otro lugar con create=true.

Nota: otra forma es usando una convención que está explicada en Check for Creation.

Configurando los SSO

Se puede controlar como es instanciado el objeto de estado de sesión. De esta forma se le pueden inyectar algunos valores cuando sea creado para inicializarlo. Para ello se ha de proporcionar un ApplicationStateCreator que será el responsable de crear el SSO cuando sea necesario. Esta técnica puede ser usada cuando queramos que el SSO esté representado por una interfaz en vez de una clase.

Un SSO se configura haciendo una contribución al servicio ApplicationStateManager. En el módulo de la aplicación:

```
1 public void contributeApplicationStateManager(MappedConfiguration<Class,
      ApplicationStateContribution> configuration) {
    ApplicationStateCreator<MyState> creator = new ApplicationStateCreator<CarritoCompra>()
       {
        public CarritoCompra create() {
          return new CarritoCompra(new Date());
5       }
    };
    configuration.add(CarritoCompra.class, new ApplicationStateContribution("session",
      creator));
  }
```

En este ejemplo muy simple el creador usa un constructor alternativo con una fecha. No hay nada que nos impida definir un constructor que inyecte cualquier servicio del contenedor de dependencias.

Atributos de sesión

Como alternativa a los SSO, los atributos de sesión o session attributes proporcionan un mecanismo que permite almacenar datos en la sesión por nombre en vez de por tipo. Esto es particularmente útil para aplicaciones heredadas que manipulan directamente el objeto HttpSession.

```java
public class MiPagina {

  @SessionAttribute
  private Usuario usuario;
}
```

O usando el mismo nombre de atributo de sesión pero utilizando el nombre de variable que queramos.

```java
public class MiPagina {

  @SessionAttribute("usuario")
  private Usuario usuarioQueHaIniciadoSesion;
}
```

8.3.1 Datos de sesión externalizados con Spring Session

Por defecto los datos de la sesión de una aplicación web Java se guardan en el servidor de aplicaciones y en memoria, esto produce que al reiniciar el servidor por un despliegue los datos de la sesión se pierdan y provoque en los usuarios alguna molestia como tener que volver a iniciar sesión. En Tomcat existe la posibilidad de que los datos de las sesiones sean persistidas en disco con la opción saveOnRestart del elemento de configuración Manager que evita que los datos de las sesiones se pierdan en los reinicios, al menos para los servicios formados por una única instancia. Para evitar que los usuarios perciban los reinicios o caídas del servidor hay varias soluciones algunas tratando de diferentes formas externalizar las sesiones del servidor de aplicaciones. Con estas soluciones se pueden hacer despliegues sin caídas, sin que las perciban los usuarios, siendo útil para hacer actualizaciones frecuentemente, continuas, y en cualquier momento cuando tengamos una nueva versión de la aplicación.

Las soluciones más comentadas son:

- Cluster de servidores: para evitar las caídas podemos formar un cluster de máquinas de forma que si una se reinicia las peticiones sean atendidas por el resto de servidores del cluster. Añadiendo una poca configuración se puede formar un cluster de servidores Tomcat. Si el cluster está formado por unos pocos servidores esta solución es válida pero si el cluster es grande (¿media docena de máquinas?) el tráfico que se genera para sincronizar los datos de sesión en todas las máquinas puede ser significativo, momento en el cual se opta por otras soluciones.
- Sesión en base de datos relacional: los datos de la sesión se pueden guardar en una base de datos relacional, al llegar una petición al servidor se recupera de la base de datos la sesión con una consulta y al finalizar la petición se lanza otra consulta de actualización. En las aplicaciones la base de datos suele ser un cuello de botella prefiriéndose guardar la sesión en otro servidor que no sea el servidor de base de datos para no generarle más carga.

- Caché externa: en esta opción los datos se guardan en un servidor externo al servidor de aplicaciones de forma que todos los servidores del cluster las compartan pero no en la base de datos relacional, algunas opciones que se pueden utilizar son memcached o Redis que almacenan los datos en memoria y son muy rápidas. Esta opción añade una pieza más a la infraestructura de la aplicación que hay que mantener. A continuación pondré un ejemplo usando esta opción utilizando Spring Session y un servidor Redis.

- Sesión en cookie: para no añadir una pieza más a la infraestructura del servidor se puede externalizar la sesión en el cliente mediante una cookie. Como la cookie es enviada por el navegador cliente en cada petición el servidor puede recuperar los datos de la sesión. Sin embargo, como los datos son guardados en el cliente los datos de la cookie han de ser cifrados y firmados digitalmente para evitar problemas de seguridad ante modificaciones de los datos. También deberemos evitar guardar muchos datos y tendremos cierta limitación para que la cookie no sea grande, el tamaño recomendado no exceder es 4096 bytes si lo hacemos puede que ocasionemos errores con el mensaje 400 bad request, request header or cookie too large y consuma mucho ancho de banda, hay que tener en cuenta que las cookies son enviadas en cada petición al servidor origen no solo para las peticiones dinámicas sino también para los recursos estáticos como imágenes u hojas de estilos, si las cookies son grandes y el número de usuarios también el ancho de banda consumido por las cookies puede ser significativo, en estos últimos casos empleando un CDN puede aliviarse el tráfico generado. En la siguiente página podemos encontrar límites de las cookies para cada navegador y el número máximo por dominio.

Usando Spring Session podemos externalizar los datos de la sesión en un servidor Redis usándolo como caché externa. Para demostrar y enseñar el código necesario he creado una pequeña aplicación web con Spring MCV. El controlador no tiene nada especial, obtiene la sesión y guarda los datos que necesita. Usando la anotación @EnableRedisHttpSession activamos la infraestructura necesaria en el contenedor de Spring para guardar los datos de la sesión en Redis. Por supuesto deberemos añadir las dependencias que necesitemos en la herramienta de construcción que usemos.

Listado 8.1: SessionController.java

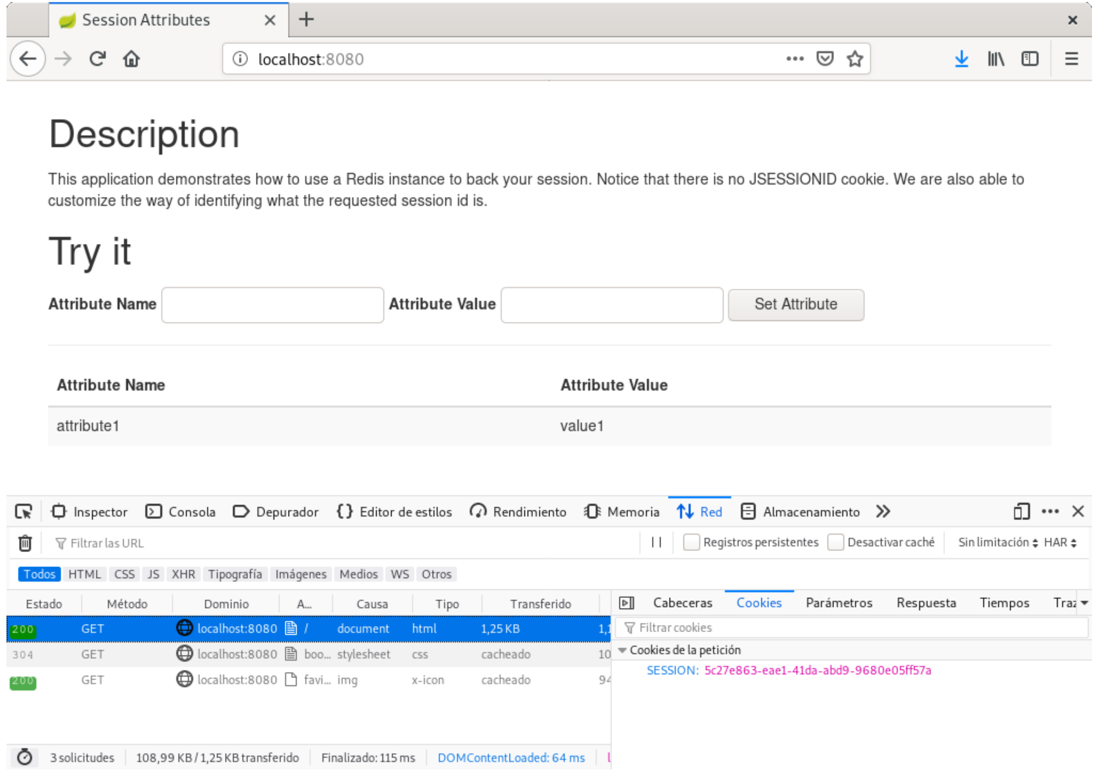
```
1  package io.github.picodotdev.springsession;

   ...

5  @Controller
   public class SessionController {

     @RequestMapping(value = "/")
     public String index() {
10     return "index";
     }

     @RequestMapping(value = "/attributes", method = RequestMethod.POST)
     public String post(@RequestParam(value = "attributeName", required = true) String name,
        @RequestParam(value = "attributeValue", required = true) String value, HttpSession
       session, Model model) {
15     session.setAttribute(name, value);
       return "redirect:/";
     }
```

```
}
```

Listado 8.2: Config.java

```
1 @Bean
2 public JedisConnectionFactory connectionFactory() {
    return new JedisConnectionFactory();
  }
```

Lanzando una petición se puede ver como el Redis se guardan los datos de la sesión. Podemos detener el servidor y volverlo a iniciar y comprobaremos que los datos de la sesión no se han perdido al estar persistidos en Redis.



Examinando los datos en redis podemos ver que se ha creado una clave con el mismo identificativo de la cookie SESSION, en la clave están guardados los valores serializados entre ellos el nombre del atributo y su valor y otros datos como la fecha de creación, el último acceso y el intervalo máximo de inactividad antes de la expiración.

Terminal

```
127.0.0.1:6379> keys *
1) "spring:session:expirations:1555700280000"
2) "spring:session:sessions:expires:5c27e863-eae1-41da-abd9-9680e05ff57a"
3) "spring:session:sessions:5c27e863-eae1-41da-abd9-9680e05ff57a"
127.0.0.1:6379>
```

En el momento de escribir este artículo Spring Session es un proyecto reciente y solo soporta la opción de Redis como caché externa pero seguramente con nuevas versiones soporte otras opciones como memcached, guardar la sesión en una cookie o en una base de datos relacional. La solución propuesta por Spring Session es válida para cualquier servidor de aplicaciones ya que se basa en crear un filtro en la aplicación que proporciona una versión modificada de HttpSession mediante el cual se guardan los datos de forma externa.

Otras posibilidades ofrecidas por Spring Session son múltiples sesiones en la misma instancia del navegador y soporte para aplicaciones RESTful y WebSocket. Los identificativos de sesión por defecto es un número de 128 bits con una entropía de 64 bits, para mayor seguridad es posible Aumentar el tamaño del identificativo de la cookie de sesión de Tomcat o Spring Session.

A tener en cuenta

Del mismo modo que los SSO, los atributos de sesión usan un almacén compartido por toda la aplicación en el que hay serias posibilidades de producirse colisiones no solo en tu aplicación sino con otros módulos o librerías. Para evitar estos problemas se debería calificar los atributos de sesión con una convención similar a los paquetes de las clases. Por ejemplo, usar algo similar a «io.github.picodotdev.plugintapestry.usuario» en vez de solo «usuario». Es mejor definir ese nombre como una constante para evitar errores de escritura. Por ejemplo:

```
1 public class MiPagina {

    public static final String USUARIO_SESSION_ATTRIBUTE = "io.github.picodotdev.
      plugintapestry.usuario";
```

```
  @SessionAttribute(USUARIO_SESSION_ATTRIBUTE)
6   private User usuario;
}
```

Capítulo 9

Persistencia en base de datos

Las aplicaciones básicamente tratan y manejan información que puede provenir de diferentes fuentes, del usuario o de un sistema externo. Muy habitualmente parte de esa información es necesario conservarla de forma permanente y consistente para ser recuperada en otro momento futuro. Para ello se desarrollaron los sistemas de base de datos que conservan grandes volúmenes de datos de forma estructurada y permiten acceder a ella de forma suficientemente rápida cuando se solicita. Hay varios tipos de sistemas de bases de datos con diferentes características, dos de ellos son:

- Relacionales
- NoSQL

9.1 Bases de datos relacionales

Cada entidad es almacenada en una tabla que tiene cierta estructura común para todas las entidades almacenadas en ella. Cada instancia de una entidad almacenada representa una fila y cada fila se divide en campos, uno por cada pieza de información que se quiera guardar de esa entidad. Los campos pueden ser de diferentes tipos: numéricos, cadenas de texto, fechas, ... Algunos campos de una tabla pueden añadirse con el único objetivo de hacer referencia a filas de otras tablas y es la forma en que las tablas se relacionan unas con otras. Todas las tablas tienen una clave primaria e identifica inequívocamente a cada una de las filas, la clave primaria no es más que uno o un grupo de campos de la fila. Las tablas que se relacionan con otras tendrán claves foráneas que también no son más que campos especiales que contienen la clave primaria de una fila de otra tabla. Se pueden distinguir diferentes tipos de relaciones según la cardinalidad:

- 1 a 1 (uno a uno): una fila de una tabla está relacionada con una fila de otra tabla.
- 1 a N (uno a varios): una fila de la parte 1 de una tabla está relacionada con varias de otra tabla de la parte N, pero las filas de la parte N solo están relacionadas con una de la parte 1.

- N a M (varios a varios): una fila de la parte N de una tabla está relacionada con varias de otra tabla de la parte M, y a su vez las filas de la parte M pueden relacionarse con varias de la parte N. Esta relación puede modelarse también como dos relaciones, una 1 a N y otra N a 1, con una tabla intermedia en la que cada fila contiene la clave primaria de ambas tablas.

9.1.1 Propiedades ACID

Los datos son una pieza muy importante dentro de una aplicación y por tanto las bases de datos relacionales tienen que garantizar que la información que almacenan es válida. Para que la información que las base de datos almacenan sea válida deben garantizar en su funcionamiento las propiedades ACID.

Atomicidad (A)

Muchas modificaciones de la base de datos implican varias acciones individuales pero que están relacionadas. Mediante esta propiedad para que un conjunto de operaciones relacionadas se consideren válidas tiene que garantizarse que se ejecutan todas o no se ejecuta ninguna, es decir, las diferentes operaciones individuales se tienen que ejecutar como una unidad de forma indivisible o atómica. Esto se consigue mediante las transacciones que garantizan la atomicidad de las operaciones desde que son iniciadas hasta que se terminan.

Consistencia (C)

Esta propiedad garantiza que cada transacción llevará a la base de datos de un estado válido a otro válido. Las bases de datos pueden aplicar reglas o restricciones a los valores de los campos garantizándose que al final de una transacción los campos cumplen todas las restricciones. Por ejemplo, puede ser requerido que un campo esté siempre entre 0 y 100. Una operación puede sumar 150 al campo, en este momento si la transacción terminase no se cumpliría la restricción y los datos no serían válidos, pero más tarde otra operación puede restar 150, el campo tendrá el mismo valor y al final de la transacción mediante esta propiedad la base de datos comprobará que la restricción se sigue cumpliendo.

Aislamiento (I)

Mediante esta propiedad se garantiza que dos transacciones llevadas al mismo tiempo no se interfieren entre ellas cuando modifican y leen los mismos datos.

Durabilidad (D)

Esta propiedad garantiza que las transacciones dadas por terminadas perduran en la base de datos aunque se produzcan otros fallos como un fallo de corriente poco después de terminar una transacción.

9.1.2 Lenguaje SQL

El lenguaje SQL (Structured Query Language, lenguaje de consulta estructurado) es el potente lenguaje utilizado para operar contra una base de datos tanto para hacer consultas como para hacer modificaciones de datos o de las tablas de la base de datos. Según sean las sentencias SQL pueden distinguirse:

- Sentencias DML (Data Manipulation Language): son sentencias que manipulan datos como altas (INSERT), modificaciones (UPDATE), eliminación (DELETE) o selección (SELECT).
- Sentencias DDL (Data Definition Language): son sentencias que se utilizan para administrar las bases de datos y las tablas. Permiten crear nuevas bases de datos, crear modificar o eliminar campos, tablas o restricciones.

Algunos ejemplos de sentencias de manipulación de datos son:

- Inserción: insert into producto (id, nombre, precio) values (1, 'Tapestry 5', 25);
- Actualización: update producto set nombre = 'Tapestry 5 - Rapid web application development in Java', precio = 20 where id = 1;
- Selección: select nombre, precio from producto where id = 1;
- Eliminación: delete from producto where id = 1;

Hay muchas bases de datos disponibles entre las que podemos elegir ya sean comerciales como Oracle y Microsoft SQL Server o libres y sin ningún costo al menos en licencias como PostgreSQL, MariaDB, MySQL y H2.

9.2 Bases de datos NoSQL

Las bases de datos NoSQL surgen por la necesidad de algunas aplicaciones de tratar cantidades enormes de datos y usuarios evitando la rigidez de los sistemas estructurados relacionales. Son bases de datos optimizadas para agregar, modificar y eliminar datos, no es necesario que los datos tengan estructuras predefinidas y suelen ser más escalables. La desventaja es que no garantizan completamente las propiedades ACID de las bases de datos relacionales pero en determinados casos se considera más prioritario la velocidad que la exactitud. Hay diferentes tipos según como guardan la información:

- Documento: la información es guardada en formatos como JSON, XML o documentos como Word o Excel.
- Grafo: los elementos están interrelacionados y las relaciones se representan como un grafo.
- Clave-valor: los valores pueden ser un tipo de un lenguaje de programación. Cada valor es identificado por una clave por la que se puede recuperar.

Algunas bases de datos NoSQL pueden entrar dentro de varias categorías de las anteriores y pueden usarse desde Java. Algunos ejemplos son: Redis (jedis), MongoDB (MongoDB Java Driver), Apache Cassandra, Amazon DynamoDB.

9.3 Persistencia en base de datos relacional

En Java disponemos de varias opciones para persistir la información a una base de datos relacional, algunas de ellas son:

- JDBC: es la API que viene integrada en la propia plataforma Java sin necesidad de ninguna librería adicional exceptuando el driver JDBC para acceder a la base de datos. Mediante esta opción se tiene total flexibilidad y evita la abstracción y sobrecarga de los sistemas como Hibernate y JPA. Se trabaja con el lenguaje SQL de forma directa y este lenguaje puede variar en algunos aspectos de una base de datos a otra con lo que para migrar a otra base de datos puede implicar reescribir las SQL de la aplicación. Normalmente se utiliza alguna de las siguientes opciones pero conocer este bajo nivel de acceso a la base de datos es importante para comprender el funcionamiento de otras opciones de más alto nivel ya que internamente lo utilizan.
- Hibernate: el modelo relacional de las bases de datos es distinto del modelo de objetos del los lenguajes orientados a objetos. Los sistemas ORM como Hibernate tratan de hacer converger el sistema relacional hacia un modelo más similar al modelo de objetos de lenguajes como Java, de forma que trabajar con ellos sea similar a trabajar con objetos. En un ORM como Hibernate normalmente no se trabaja a nivel de SQL como con JDBC para consultas simples sino que se trabaja con objetos (POJO), las consultas devuelven objetos, las relaciones se acceden a través de propiedades y las eliminaciones, actualizaciones e inserciones se realizan usando objetos y métodos. Los objetos POJO incluyen anotaciones que le indican a Hibernate cual es la información a persistir y las relaciones con otros POJO. Como Hibernate dispone de esta información en base a ella puede recrear o actualizar las tablas y los campos necesarios según la definición de esas anotaciones. El ORM es encarga de traducir las acciones a las SQL entendidas por el sistema relacional, esto proporciona la ventaja adicional de que el ORM puede generar las sentencias SQL adaptadas al dialecto de la base de datos utilizada. De esta forma se podría cambiar de una base de datos a otra sin realizar ningún cambio en la aplicación o con pocos cambios comparado con los necesarios usando JDBC. Con Hibernate se puede emplear un lenguaje de consulta para casos de consultas más avanzadas similar a SQL pero adaptado al modelo orientado a objetos, el lenguaje es HQL.
- JPA: es una especificación de Java que define una API común para los sistemas ORM. Con JPA podríamos cambiar de proveedor ORM sin realizar ningún cambio en la aplicación. JPA se ha basado en gran parte en Hibernate y su forma de trabajar es similar, el lenguaje HQL también es similar pero denominado JPQL.
- jOOQ: con JDBC no tenemos tipado seguro ni el compilador valida la sintaxis de las SQL, con los ORM añadimos una abstracción sobre el modelo relacional para adaptarlo al modelo de objetos en el que perdemos funcionalidades proporcionadas en el potente lenguaje SQL y de las avanzadas bases de datos, los lenguajes HQL y JPQL son similares a SQL pero en consultas avanzadas no proporcionan la mismas funcionalidad y los procedimientos almacenados que en los ORM suelen ser ciudadanos de segunda clase tienen su utilidad. jOOQ devuelve a las bases de datos el protagonismo que se merecen ya que contienen lo más importante de una aplicación, los datos, y que se pierde en los ORM.

Persistencia con Hibernate

Lo primero que deberemos hacer es añadir las dependencias al proyecto incluido el driver JDBC para la base de datos específica que usemos, en este caso H2. He usado H2 como base de datos ya que puede embeberse

en una aplicación sin necesidad de tener un sistema externo como ocurren en el caso de MySQL y PostgreSQL. De esta forma este ejemplo puede probarse sin necesidad de instalar previamente ningún servidor de base de datos relacional.

Listado 9.1: build.gradle

```
dependencies {
  ...
  compile "org.apache.tapestry:tapestry-hibernate:$versions.tapestry"
  compile "org.apache.tapestry:tapestry-beanvalidator:$versions.tapestry"
  ...
  // Dependencias para persistencia con Hibernate
  compile "org.hibernate:hibernate-core:$versions.hibernate"
  compile "org.hibernate:hibernate-validator:$versions.hibernate_validator"
  compile "com.h2database:h2:$versions.h2"
}
```

Una vez incluidas las dependencias debemos configurar Tapestry para que nos proporcione el soporte de acceso a una base de datos, definimos en el contenedor de dependencias los servicios DAO usando Spring y al mismo tiempo configuraremos la transaccionalidad. Tapestry ofrece cierto soporte para transacciones con la anotación CommitAfter pero es muy básico de modo que es recomendable 14.1.1 y usar el soporte de transacciones de este último.

```
package io.github.picodotdev.plugintapestry.spring;

...

@Configuration
@ComponentScan({ "io.github.picodotdev.plugintapestry" })
@EnableTransactionManagement
public class AppConfiguration {

    @Bean(destroyMethod = "close")
    public DataSource dataSource() {
        BasicDataSource ds = new BasicDataSource();
        ds.setDriverClassName(Driver.class.getCanonicalName());
        ds.setUrl("jdbc:h2:./misc/database/app");
        ds.setUsername("sa");
        ds.setPassword("sa");
        return ds;
    }

    // Hibernate
    @Bean(name = "sessionFactory")
    public LocalSessionFactoryBean sessionFactoryBean(DataSource dataSource) {
        Map<String, Object> m = new HashMap<>();
        m.put("hibernate.dialect", H2Dialect.class.getCanonicalName());
        m.put("hibernate.hbm2ddl.auto", "create");
```

```
        // Debug
        m.put("hibernate.generate_statistics", true);
        m.put("hibernate.show_sql", true);

        Properties properties = new Properties();
        properties.putAll(m);

        //
        LocalSessionFactoryBean sf = new LocalSessionFactoryBean();
        sf.setDataSource(dataSource);
        sf.setPackagesToScan("io.github.picodotdev.plugintapestry.entities");
        sf.setHibernateProperties(properties);
        return sf;
    }

  ...

    // Servicios
    @Bean
    public HibernateProductoDAO hibenateProductoDAO(SessionFactory sessionFactory) {
        return new DefaultHibernateProductoDAO(sessionFactory);
    }
}
```

Las clases con capacidad de persistencia han de ubicarse en un subpaquete que proporcionaremos al definir en el contenedor de Spring el bean SessionFactory tal que ${tapestry.app-package}.entities, por ejemplo, si el paquete de las clases de Tapestry indicado en el parámetro de inicialización tapestry.app-package fuese io.github.picodotdev.plugintapestry el paquete de las entidades de Hibernate debe ser io.github.picodotdev.plugintapestry.e Esta es la convención y la forma preferida de hacerlo, si se quiere cambiar es posible hacerlo modificando el valor de la propiedad packagesToScan de la clase LocalSessionFactoryBean al definir el bean en el contenedor de Spring.

El código de acceso a base de datos suele ponerse en una clase denominada servicio o DAO. Ya que las operaciones de acceso a base de datos son candidatas a ser reutilizadas desde varias páginas o componentes es recomendable hacerlo así, además de hacer que las páginas de Tapestry sean más pequeñas (ya tienen suficiente responsabilidad con hacer de controlador en el modelo MVC) permite que si un día cambiamos de framework web solo tendríamos que modificar la capa de presentación. Todo el código de los servicios nos serviría perfectamente sin ninguna o pocas modificaciones.

El contenedor de dependencias se encargará de en el momento que necesite construir una instancia del servicio DAO y pasarle en el constructor los parámetros necesarios, también se puede inyectar los servicios que necesite usando la anotación @Inject. En este caso una de las clases principales de la API de Hibernate es Session. Una vez con la referencia al objeto Session usamos sus métodos para realizar las consultas y operaciones que necesite proporcionar el DAO.

Listado 9.2: GenericDAO.java

```
package io.github.picodotdev.plugintapestry.services.dao;
```

```
...

public interface GenericDAO<T> {
  T findById(Serializable id);
  List<T> findAll();
  List<T> findAll(Pagination paginacion);
  long countAll();

  void persist(T entity);
  void remove(T entity);
  void removeAll();
}
```

Listado 9.3: HibernateProductoDAO.java

```
package io.github.picodotdev.plugintapestry.services.dao;

import io.github.picodotdev.plugintapestry.entities.hibernate.Producto;

public interface HibernateProductoDAO extends GenericDAO<Producto> {
}
```

Listado 9.4: DefaultHibernateProductoDAO.java

```
package io.github.picodotdev.plugintapestry.services.dao;

...

@SuppressWarnings({ "unchecked" })
public class DefaultHibernateProductoDAO implements HibernateProductoDAO {

    protected SessionFactory sessionFactory;

    public DefaultHibernateProductoDAO(SessionFactory sessionFactory) {
        this.sessionFactory = sessionFactory;
    }

    @Override
    @Transactional(readOnly = true)
    public Producto findById(Long id) {
        return (Producto) sessionFactory.getCurrentSession().get(Producto.class, id);
    }

    @Override
    @Transactional(readOnly = true)
    public List<Producto> findAll() {
        return findAll(null);
    }
```

```
    @Override
    @Transactional(readOnly = true)
    public List<Producto> findAll(Pagination pagination) {
29      Criteria criteria = sessionFactory.getCurrentSession().createCriteria(Producto.
    class);

        if (pagination != null) {
            List<Order> orders = getOrders(pagination);
            for (Order order : orders) {
34              criteria.addOrder(order);
            }
        }

        if (pagination != null) {
39          criteria.setFirstResult(pagination.getOffset());
            criteria.setFetchSize(pagination.getNum());
        }

        return criteria.list();
44  }

    @Override
    @Transactional(readOnly = true)
    public long countAll() {
49      Criteria criteria = sessionFactory.getCurrentSession().createCriteria(Producto.
    class);

        criteria.setProjection(Projections.rowCount());

        return (long) criteria.uniqueResult();
54  }

    @Override
    @Transactional(propagation = Propagation.REQUIRED)
    public void persist(Producto object) {
59      sessionFactory.getCurrentSession().saveOrUpdate(object);
    }

    @Override
    @Transactional(propagation = Propagation.REQUIRED)
64  public void remove(Producto object) {
        sessionFactory.getCurrentSession().delete(object);
    }

    @Override
69  @Transactional(propagation = Propagation.REQUIRED)
    public void removeAll() {
        String hql = String.format("delete from %s", Producto.class.getName());
        Query query = sessionFactory.getCurrentSession().createQuery(hql);
```

```
        query.executeUpdate();
74  }

    private List<Order> getOrders(Pagination pagination) {
        List<Order> orders = new ArrayList<Order>();
        for (Sort s : pagination.getSort()) {
79          Order o = s.getOrder();
            if (o != null) {
                orders.add(o);
            }
        }
84      return orders;
    }
}
```

La interfaz HibernateProductoDAO extiende GenericDAO que puede servir como como implementación base proporcionando métodos básicos de búsqueda, persistencia y eliminación.

Persistencia con jOOQ

Con el auge de los lenguajes de programación orientados a objetos han surgido varias herramientas que intentan hacer que el trabajo de unir el mundo orientado a objetos del lenguaje que empleemos y el modelo relacional de las bases de datos sea más transparente, estas herramientas son conocidas como Object Relational Mapping (ORM). Una de las más conocidas y usada en la plataforma Java es Hibernate. Sin embargo, aunque facilitan el acceso a los datos no están exentas de problemas y están surgiendo nuevas alternativas para tratar de solventar algunos de ellos, una de ellas es jOOQ.

Si hemos usado Hibernate sabremos que aunque este ampliamente usado facilitando la conversión entre el modelo relacional en base de datos y el modelo orientado a objetos del lenguaje Java también presenta problemas. Uno de los problemas es que al abstraer el acceso a base de datos no somos tan conscientes de las sentencias SQL que se envían a la base de datos provocando los problemas 1+N y que la aplicación sea lenta, poco eficiente y sobrecargar la base de datos. Otro problema es que cuando necesitamos realizar una consulta compleja o avanzada el lenguaje HQL no nos ofrezca todo lo que necesitamos haciendo que tengamos que escribir directamente la consulta en lenguaje SQL con lo que perdemos la validación del compilador y si usamos una funcionalidad específica de un motor de base de datos la independencia del mismo. También puede ocurrirnos que diseñamos los modelos para complacer al framework de persistencia ORM.

jOOQ es una herramienta que facilita el acceso a la base de datos usando un enfoque diferente de los ORM, no trata de crear una abstracción sobre la base de datos relacional sino que pone el modelo relacional como elemento central de la aplicación en cuanto a la persistencia. Algunas de las características destacables de jOOQ son:

- La base de datos primero: los datos son probablemente lo más importante de una aplicación. En los ORM los modelos de objetos dirigen el modelo de base de datos, no siempre es sencillo (más) en bases de datos heredadas que no tienen la estructura necesaria usable por los ORMs. En jOOQ el modelo relacional dirige el modelo de objetos, para jOOQ el modelo relacional es más importante que el modelo de objetos.

- jOOQ usa SQL como elemento central: en jOOQ se pueden construir las SQLs usando una API fluida con la que el compilador puede validar la sintaxis, metadatos y tipos de datos. Se evitan y se detectan rápidamente los errores de sintaxis con la ayuda del compilador y con la ayuda de un IDE se ofrece asistencia de código que facilita el uso de la API. Está a un nivel bastante cercano al lenguaje SQL.
- SQL con tipado seguro: las sentencias se construyen usando código Java con la que el compilador validará el código y que los tipos de los datos usados sean los correctos, los errores los encontraremos en tiempo de compilación en vez de en tiempo de ejecución. jOOQ proporciona un DSL y una API fluida de fácil uso y lectura.
- Generación de código: jOOQ genera clases a partir de los metadatos (el modelo relacional) de la base de datos. Cuando se renombre una tabla o campo en base de datos generados los modelos el compilador lo indicará. Si en algún momento hay que renombrar una columna de la base de datos deberemos modificar los modelos, jOOQ permite regenerar las clases Java de acceso a la base de datos y el compilador nos avisará de aquello que no esté sincronizado entre la base de datos y el código Java.
- Multi-Tenancy: permite configurar la base de datos o bases de datos a la que se accederán en desarrollo, pruebas y producción.
- Active Records: jOOQ puede generar el código de acceso a la base de datos a partir del esquema, estas clases emplean el patrón Active Record. La implementación de este patrón ya proporciona las operaciones CRUD (uno de los avances de Hibernate) con lo que no tendremos que escribirlas para cada uno de los modelos de la aplicación, nos ahorraremos mucho código. Este código que se genera es opcional, jOOQ puede usarse simplemente para generar las sentencias SQL y usar JDBC sin la abstracción de los Active Records.
- Estandarización: las bases de datos tienen diferencias en los dialectos SQL. jOOQ realiza transformaciones de expresiones SQL comunes a la correspondencia de la base de datos de forma que las SQLs escritas funcionen en todas las bases de datos de forma transparente, esto permite migrar de un sistema de datos sin cambiar el código de la aplicación. Este también era un avance proporcionado por los ORM, incluido Hibernate.
- Ciclo de vida de las consultas: proporciona llamadas (hooks) de forma que se puedan añadir comportamientos, por ejemplo para logging, manejo de transacciones, generación de identificadores, transformación de SQLs y más cosas.
- Procedimientos almacenados: los procedimientos almacenados son ciudadanos de primera clase y pueden usarse de forma simple al contrario de lo que sucede en los ORM. Para algunas tareas los procedimientos almacenados son muy útiles.

Los ORMs ofrecen como ventajas sobre el uso directo de JDBC la implementación de las operaciones CRUD, construir las SQLs con una API en vez de concatenando Strings propensos a errores al modificarlos y la independencia del motor de base de datos usado pudiendo cambiar a otro sin afectar al código de la aplicación. La navegación de las relaciones es más explícita que en Hibernate y obtener datos de múltiples tablas con jOOQ diferente.

Si nos convencen estas características y propiedades de jOOQ podemos empezar leyendo la guía de inicio donde se comenta los primeros pasos para usarlo. La documentación de jOOQ está bastante bien explicada pero no se comentan algunas cosas que al usarlo en un proyecto tendremos que buscar.

En el siguiente ejemplo mostraré como usar jOOQ y la configuración necesaria para emplearlo junto con Spring. En la siguiente configuración de Spring usando únicamente código Java se construye un DataSource, un Datasource con soporte de transacciones para el acceso a la base de datos, un ConnectionProvider que usará el DataSource para obtener las conexiones a la base de datos, con la clase Configuration realizamos la configuración, finalmente DSLContext es el objeto que usaremos para construir las sentencias SQL normalmente en los servicios.

Listado 9.5: AppConfiguracion.java

```java
package io.github.picodotdev.plugintapestry.spring;

...

@Configuration
@ComponentScan({ "io.github.picodotdev.plugintapestry" })
@EnableTransactionManagement
public class AppConfiguration {

    @Bean(destroyMethod = "close")
    public DataSource dataSource() {
        BasicDataSource ds = new BasicDataSource();
        ds.setDriverClassName(Driver.class.getCanonicalName());
        ds.setUrl("jdbc:h2:./misc/database/app");
        ds.setUsername("sa");
        ds.setPassword("sa");
        return ds;
    }

    ...

    @Bean
    public DataSource transactionAwareDataSource(DataSource dataSource) {
        return new TransactionAwareDataSourceProxy(dataSource);
    }

    @Bean
    public ResourceTransactionManager transactionManager(DataSource dataSource) {
        return new DataSourceTransactionManager(dataSource);
    }

    // jOOQ
    @Bean
    public ConnectionProvider connectionProvider(DataSource dataSource) {
        return new DataSourceConnectionProvider(dataSource);
    }

    @Bean
    public org.jooq.Configuration config(ConnectionProvider connectionProvider) {
        DefaultConfiguration config = new DefaultConfiguration();
        config.set(connectionProvider);
```

```
        config.set(SQLDialect.H2);
        return config;
44    }

    @Bean
    public DSLContext dsl(org.jooq.Configuration config) {
        return DSL.using(config);
49    }

    ...

    // Servicios
54    @Bean
    public JooqProductoDAO jooqProductoDAO(DSLContext context) {
        return new DefaultJooqProductoDAO(context);
    }
}
```

Podemos usar jOOQ como generador de sentencias SQL y posteriormente ejecutarlas con JDBC, también podemos usar el patrón Active Record o una combinación de ambas opciones. Usando el patrón Active Record si necesitamos incluir campos adicionales a los presentes en la base de datos que manejen cierta lógica en la aplicación, también puede que necesitemos incluir métodos de lógica de negocio adicionales. Para incluir estos datos y métodos tendremos que extender la clase Active Record que genera jOOQ. En aquellos sitios de la aplicación que necesitemos usar esas propiedades y métodos adicionales deberemos transformar la instancia de la clase que usa jOOQ (ProductoRecord) por la clase que tenga esos datos adicionales (AppProductoRecord). Para ello la API de la clase Record ofrece el método into o from como muestro en el código de AppProductoRecord a continuación.

jOOQ nos genera automáticamente las clases que implementa el patrón Active Record y dispondremos de los métodos CRUD heredados de la clase Record.

Listado 9.6: AppProductoRecord.java

```
1 package io.github.picodotdev.plugintapestry.records;
2
  ...

  public class AppProductoRecord extends ProductoRecord {

7   // Propiedades y métodos adicionales propios a los incluídos
    // en la clase ProductoRecord generada por jOOQ
  }
```

Los desarrolladores de jOOQ abogan por la eliminación de capas en la arquitectura de la aplicación pero puede que aún preferimos desarrollar una capa que contenga las consultas a la base de datos que sea usada y compartida por el resto la aplicación para el acceso los datos, quizá más que una capa en este caso es una forma de organizar el código. Los Active Records proporcionan algunos métodos de consulta pero probablemente necesitaremos más. En el siguiente ejemplo podemos ver como son las consultas con jOOQ. Si necesitamos

métodos de búsqueda adicionales a los que por defecto jOOQ proporciona en Blog Stack he creado una clase DAO por cada entidad de la base de datos. En el siguiente ejemplo se puede ver como se construyen las sentencias SQL con jOOQ usando su API fluida.

Listado 9.7: JooqProductoDAO.java

```
1  package io.github.picodotdev.plugintapestry.services.dao;

   import io.github.picodotdev.plugintapestry.entities.jooq.tables.pojos.Producto;

   public interface JooqProductoDAO extends GenericDAO<Producto> {
6
   }
```

Listado 9.8: DefaultJooqProductoDAO.java

```
1  package io.github.picodotdev.plugintapestry.services.dao;

3  ...

   public class DefaultJooqProductoDAO implements GenericDAO<Producto>, JooqProductoDAO {

       private DSLContext context;
8
       public DefaultJooqProductoDAO(DSLContext context) {
           this.context = context;
       }

13     @Override
       @Transactional(readOnly = true)
       public Producto findById(Long id) {
           return context.selectFrom(PRODUCTO).where(PRODUCTO.ID.eq(id)).fetchOneInto(
       Producto.class);
       }
18
       @Override
       @Transactional(readOnly = true)
       public List<Producto> findAll() {
           return context.selectFrom(PRODUCTO).fetchInto(Producto.class);
23     }

       @Override
       @Transactional(readOnly = true)
       public List<Producto> findAll(Pagination pagination) {
28         return context.selectFrom(Tables.PRODUCTO).orderBy(getSortFields(pagination)).
       limit(pagination.getOffset(), pagination.getNum()).fetchInto(Producto.class);
       }

       @Override
       @Transactional(readOnly = true)
```

```
    public long countAll() {
        return context.selectCount().from(Tables.PRODUCTO).fetchOne(0, Long.class);
    }

    @Override
    @Transactional(propagation = Propagation.REQUIRED)
    public void persist(Producto object) {
        getRecord(object).store();
    }

    @Override
    @Transactional(propagation = Propagation.REQUIRED)
    public void remove(Producto object) {
        getRecord(object).delete();
    }

    @Override
    @Transactional(propagation = Propagation.REQUIRED)
    public void removeAll() {
        context.deleteFrom(PRODUCTO).execute();
    }

    private ProductoRecord getRecord(Producto object) {
        ProductoRecord record = context.newRecord(PRODUCTO);
        record.from(object);
        return record;
    }

    private List<SortField<?>> getSortFields(Pagination pagination) {
        return pagination.getSort().stream().map((Sort s) -> {
            Field<?> field = PRODUCTO.field(s.getProperty());
            SortField<?> sortField = ((s.getDirection() == Direction.ASCENDING)) ? field.
    asc() : field.desc();
            return sortField;
        }).collect(Collectors.toList());
    }
}
```

Para usar el generador de código de jOOQ con Gradle debemos añadir la siguiente configuración al archivo de construcción del proyecto incluyendo las dependencias de jOOQ, este generador se conectará a la base de datos, obtendrá los datos de esquema y generará todas las clases del paquete io.github.picodotdev.plugintapestry.entities.jooq Puede que queramos usar JodaTime en vez de las clase Date y Timesptamp de la API de Java al menos si no usamos aún Java 8.

Listado 9.9: build.gradle

```
...

plugins {
```

```
    ...
    id 'nu.studer.jooq' version '3.0.3'
    id 'org.liquibase.gradle' version '2.0.1'
7 }

dependencies {
    compile("org.jooq:jooq:$versions.jooq")
    compile("commons-dbcp:commons-dbcp:$versions.commons_dbcp")
12  compile("com.h2database:h2:$versions.h2")
}

jooq {
    version = '3.11.2'
17  edition = 'OSS'
    h2(sourceSets.main) {
        jdbc {
            driver = 'org.h2.Driver'
            url = 'jdbc:h2:./misc/database/app'
22          user = 'sa'
            password = 'sa'
        }
        generator {
            name = 'org.jooq.codegen.DefaultGenerator'
27          database {
                name = 'org.jooq.meta.h2.H2Database'
                inputSchema = 'PLUGINTAPESTRY'
                includes = '.*'
                excludes = ''
32              forcedTypes {
                    forcedType {
                        types = 'TIMESTAMP'
                        userType = 'java.time.LocalDateTime'
                        converter = 'io.github.picodotdev.plugintapestry.misc.
    TimestampConverter'
37                  }
                }
            }
            generate {
                interfaces = true
42              pojos = true
                relations = true
                validationAnnotations = true
            }
            target {
47              packageName = 'io.github.picodotdev.plugintapestry.entities.jooq'
                directory = 'src/main/java'
            }
        }
    }
```

```
}

liquibase {
    activities {
        main {
            changeLogFile 'misc/database/changelog.xml'
            url 'jdbc:h2:./misc/database/app'
            username 'sa'
            password 'sa'
        }
    }
    runList = 'main'
}
```

9.4 Transacciones

En servicios complejos con mucha lógica de negocio se pueden lanzar muchas sentencias de búsqueda, inserción, modificación y eliminación. Para mantener la integridad de los datos de la base de datos estos métodos de negocio han de cumplir con las propiedades ACID. Para garantizar las propiedades ACID de atomicidad, consistencia, aislamiento y durabilidad se emplean las transacciones.

9.4.1 Anotación CommitAfter

Tapestry ofrece definir las transacciones de forma declarativa con la anotación CommitAfter. Con la anotación CommitAfter si se produce una excepción no controlada («unchecked») se hará un rollback de la transacción y, esto es importante, aún produciéndose una excepción controlada («checked») se hará el commit de la transacción y es responsabilidad del programador tratar la excepción adecuadamente. La anotación es usable en los métodos de los servicios y en los métodos manejadores de eventos de los componentes.

Sin embargo, esta anotación es muy básica y probablemente no nos sirva en casos de uso complejos. Esto ha sido objeto de discusión varias veces en la lista de distribución de los usuarios [1] [2] y el JIRA de Tapestry [3].

Sabiendo como funciona la anotación se nos plantean preguntas:

- ¿Cuál es el comportamiento cuando un método del servicio anotado llame a otro también anotado del mismo servicio?
- ¿Que pasa si cada método está en un servicio diferente?

Para el primer caso (métodos del mismo servicio) se hará una sola transacción ya que las anotaciones y los advices en Tapestry se aplican en el proxy del servicio no en la implementación. En el segundo caso (métodos en diferentes servicios) se iniciará una transacción pero haciendo un commit en la salida de cada método.

Si tenemos una aplicación compleja probablemente se nos planteará el caso de tener varios servicios que se llaman entre si y que ambos necesiten compartir la transacción, en esta situación la anotación CommitAfter probablemente no nos sirva por hacer un commit en la salida de cada método.

Tapestry no pretende proporcionar una solución propia que cubra todas las necesidades transaccionales que puedan tener todas las aplicaciones sino que con la anotación CommitAfter pretende soportar los casos simples, para casos más complejos ya existen otras opciones que están ampliamente probadas. Si necesitamos un mejor soporte para las transacciones que el que ofrece Tapestry debemos optar por Spring o por los EJB. Sin embargo, la solución de Spring nos obliga a definir los servicios transaccionales como servicios de Spring y los EJBs nos obligan a desplegar la aplicación en un servidor de aplicaciones que soporte un contenedor de EJB como JBoss/Wildfly, Geronimo, TomEE, ...

9.4.2 Transacciones con Spring

Si necesitamos algo más de lo que ofrece la anotación CommitAfter y no queremos mantener una solución propia como la anterior podemos optar por gestionar las transacciones mediante Spring o EJB. Unos buenos motivos para optar tanto por Spring como por los EJB es que son soluciones ampliamente probadas con lo que solo tendremos que integrarlo en nuestros proyectos, además ambas son ampliamente usadas incluso en proyectos grandes y complejos, será muy raro que no ofrezcan todo lo que necesitemos. Entre optar por Spring o los EJB depende de varios factores como puede ser si la aplicación va a ser desplegada en un servidor de aplicaciones con soporte para EJB (como JBoss/Wildfly, Geronimo, ...) o no (Tomcat, Jetty) o de nuestras preferencias entre ambas opciones.

Para conseguir que sea Spring el que gestione las transacciones deberemos hacer una Integración con Spring. Habiéndonos integrado con Spring para definir la transaccionalidad en los servicios con la lógica de negocio debemos usar la anotación Transactional usando los valores por defecto o indicando la propagación, el aislamiento, si es de solo lectura, timeout, etc, ... según consideremos. Debido a lo simple de la lógica de negocio de la aplicación de este ejemplo la anotación se aplica al DAO, sin embargo, en una aplicación más compleja y con mas clases sería mejor definirlo a nivel de servicio de lógica de negocio o punto de entrada a la lógica de negocio y no al nivel de los DAO que están en una capa de la aplicación más baja. En los ejemplos anteriores de las clases DefaultHibernateProductoDAO.java y DefaultJooqProductoDAO.java está incluidas las anotaciones Transactional de Spring, en los métodos de consulta se establece el modo solo lectura para la transacción y en los métodos que modifican datos la forma de propagación.

Capítulo 10

AJAX

Tapestry posee un excelente soporte para trabajar con Ajax incluso llegando al punto de no ser necesario escribir ni una sola línea de javascript para hacer este tipo de peticiones. Esto se consigue con unos cuantos componentes que ofrece Tapestry de los disponibles en el propio framework.

10.1 Zonas

Las zonas proporcionan un mecanismo para actualizar dinámicamente determinadas zonas de la página sin tener que actualizar la página por completo, son la aproximación para hacer actualizaciones parciales de una página lo que en muchas ocasiones supone una mejor experiencia de usuario además suponer menos carga para el servidor que cargar la página completa. Una zona puede ser actualizada como resultado de un EventLink, ActionLink, Select component o Form. Aquellos componentes que poseen el parámetro zone (como por ejemplo ActionLink, EventLink, Form, ...) producirán su evento de forma normal, la diferencia es que se enviará un página parcial al cliente y el contenido de esa respuesta es usado para actualizar la zona. Además de utilizar un componte que posea un parámetro zone hay que definir las zonas mediante el componente Zone que pueden ser actualizadas.

```
1 <t:actionlink t:id="enlace" zone="zona">Actualizar</t:actionlink>

  <t:zone t:id="zona">La hora actual es ${horaActual}</t:zone>
```

10.1.1 Retorno de los manejadores de evento

En las peticiones normales el valor devuelto por el manejador del evento es usado para determinar la página que se mostrará a continuación enviando una redirección. En cambio en una petición Ajax es usado para obtener una respuesta parcial en la misma petición.

Normalmente el valor devuelto es el cuerpo de la zona aunque puede ser también un componente inyectado o bloque. El código html de esos componentes es usado para actualizar la zona.

```
@InjectComponent
private Zone zona;

Object onClickFromSomeLink() {
  return zona.getBody();
}
```

La lista completa que pueden devolver los manejadores de eventos es:

- Un bloque o componente cuyo código generado es enviado como respuesta.
- El cuerpo de la propia zona.
- Un objeto JSONObject o JSONArray.
- Un objeto StreamResponse.
- Un objeto Link que producirá un redirect.
- Un nombre de página como un String, una clase de página o una instancia de página que enviarán un redirect a la página indicada.

Un manejador de evento puede conocer si la petición que va a procesar es una petición Ajax o normal pudiendo hacer un degradado de la funcionalidad si el cliente no soporta javascript.

```
@Inject
private Request request;

@InjectComponent
private Zone zona;

Object onClickFromEnlace() {
  // Retornar o el cuerpo de la zona (ajax) o toda la página (non-ajax)
  // dependiendo de tipo de petición
  return request.isXHR() ? zona.getBody() : null;
}
```

10.1.2 Actualización del múltiples zonas

En alguna ocasión puede ser necesario actualizar varias zonas como consecuencia de un evento. Tapestry ofrece soporte para hacerlo muy fácilmente. Para ello hay que usar el objeto AjaxResponseRenderer. Teniendo dos zonas y conociendo sus ids, en una misma página podríamos hacer:

```java
@InjectComponent
private Zone inputs;

@InjectComponent
private Zone ayuda;

@Inject
private AjaxResponseRenderer ajaxResponseRenderer;

void onActionFromRegister() {
  ajaxResponseRenderer.addRender("inputs", inputs).addRender("ayuda", ayuda);
}
```

Zone Component Id contra Zone Element Id

Como todos los componentes las zonas tienen un id especificado por el parámetro t:id, sin embargo, para coordinar las cosas en el lado del cliente estos necesitan conocer el id que se les asignará en el lado del cliente. Esto se especifica mediante el parámetro id del componente Zone. Si el id es desconocido se generará uno con un valor difícil de predecir. El valor se podrá obtener mediante la propiedad clientId del componente.

Recuerda que t:id se usa para inyectar el componente en el código Java del componente que lo contiene. El parámetro id del cliente es usado para orquestar las peticiones y actualizaciones.

Efectos

Una zona puede estar visible o invisible inicialmente. Cuando una zona es actualizada se hace visible si no lo estaba. A esa aparición se le puede aplicar un efecto. Por defecto se usa el efecto highlight para resaltar el cambio pero alternativamente se puede especificar un efecto diferente mediante el parámetro update. La lista de efectos son: highlight, show, slidedown, slideup y fade y si quieres puedes definir tus propios efectos.

```xml
<t:zone t:id="zona" t:update="show">
```

Limitaciones

Usar zonas dentro de cualquier tipo de bucle puede causar problemas dado que el id del cliente de la zona será el mismo para todas las zonas dentro del bucle.

Una de las cosas que hay que destacar es lo sencillo que es pasar de una aplicación no-ajax a una Ajax si esto se ajusta a lo que necesitamos, para ello basta usar los parámetros zone de los componentes y definir las zonas

en la propia página, hay que notar que no es necesario separar ese contenido de la zonas en otro archivo para devolverlo únicamente cuando se haga la petición Ajax, todo está en un único archivo y Tapestry se encarga de devolver únicamente el contenido relevante para actualizar la zona cuando esta vaya a ser refrescada en una petición Ajax. Con lo que no tendremos que trocear la página de presentación para dar soporte a las peticiones Ajax, lo que simplificará y hará más sencillo el desarrollo.

10.2 Peticiones Ajax que devuelven JSON

El actualizar fragmentos de una página con el contenido html generado por una zona cubre la mayoría de los casos en los que es necesario trabajar con Ajax, sin embargo, podemos querer trabajar de otra forma haciendo que sea el cliente el encargado de formatear los datos y presentarlos en el navegador, nosotros mismos haremos la petición Ajax esperando obtener datos en formato json que luego son procesados en el cliente para tratarlos. Esto tiene la ventaja de que puede ser el cliente el encargado de actualizar el html de la página en vez de ser el servidor el que devuelva los datos formateados con el html. Devolver json y formatearlo en el cliente es la tendencia que aplican muchos frameworks javascript como Backbone, Angular JS, Knockout, ...

A continuación un ejemplo de esta forma de hacer las cosas. El componente provoca una llamada Ajax para obtener una lista de colores en formato json al cabo de unos segundos de cargarse la página, una vez obtenido la lista de colores se muestra en un elemento del html.

Listado 10.1: Ajax.java

```
package io.github.picodotdev.plugintapestry.components;

...

public class Ajax {

  @Parameter(defaultPrefix = BindingConstants.LITERAL)
  private String selector;

  @Environmental
  private JavaScriptSupport support;

  @Inject
  private ComponentResources componentResources;

  Object onGetColores() {
    return new JSONArray("Rojo", "Verde", "Azul", "Negro");
  }

  protected void afterRender(MarkupWriter writer) {
    String link = componentResources.createEventLink("getColores").toAbsoluteURI();

    JSONObject spec = new JSONObject();
```

```
24      spec.put("selector", selector);
        spec.put("link", link);

        support.require("app/colores").invoke("init").with(spec);
      }
29  }
```

Listado 10.2: META-INF/modules/app/colores.js

```
1   define("app/colores", ["jquery"], function($) {
      function Colores(spec) {
        var _this = this;
        this.spec = spec;
        setTimeout(function() {
6         _this.getColores();
        }, 2000);
      }

      Colores.prototype.getColores = function() {
11      var _this = this;
        $.ajax({
          url: this.spec.link,
          success: function(colores) {
            var c = colores.join();
16          $(_this.spec.selector).html(c);
          }
        });
      }

21    function init(spec) {
        new Colores(spec);
      }

      return {
26      init: init
      }
    });
```

Listado 10.3: Index.tml

```
1   <p>
2     Colores: <span id="holaMundoAjax">Cargando...</span> (Ajax)
      <t:ajax selector="#holaMundoAjax"/>
    </p>
```

El javascript del ejemplo utiliza la librería jquery para hacer la petición Ajax, como Tapestry desde la versión 5.4 usa RequireJS podemos definir los assets y sus dependencias como módulos y se cargarán de forma dinámica sin necesidad de incluir ninguna etiqueta script de forma global en las páginas, no tendremos que incluir ni siquiera el script de RequireJS ya que lo hará Tapestry por nosotros.

10.3 Lanzar eventos desde JavaScript

Algunos frameworks proporcionan cierto soporte para JavaScript y recursos CSS en otros es muy escaso o inexistente. En el caso de Apache Tapestry proporciona un gran soporte no solo en la parte del servidor sino también para la parte cliente.

Una de estas funcionalidades que proporciona Tapestry es poder lanzar eventos desde el cliente mediante una petición Ajax para que sean procesados en el servidor y obtener la respuesta que se devuelva desde el servidor normalmente en formato Json. Hay que definir un manejador de evento en el servidor siguiendo la convención on[Event] y en caso de querer lanzar un evento desde el cliente anotándolo con @PublishEvent.

Listado 10.4: Event.java

```
1   package io.github.picodotdev.plugintapestry.components;

    ...

    /**
6    * @tapestrydoc
     */
    public class Event {

        @Parameter(defaultPrefix = BindingConstants.LITERAL)
11      private String selector;

        @Component
        private Any span;

16      @Environmental
        private JavaScriptSupport support;

        @PublishEvent
        Object onGetColores() {
21          return new JSONArray("Rojo", "Verde", "Azul", "Negro");
        }

        protected void afterRender(MarkupWriter writer) {
            JSONObject spec = new JSONObject();
26          spec.put("selector", span.getClientId());
```

```
        support.require("app/event").invoke("init").with(spec);
    }
}
```

Listado 10.5: Event.html

```
<!DOCTYPE html>
<t:container xmlns="http://www.w3.org/1999/xhtml" xmlns:t="http://tapestry.apache.org/
    schema/tapestry_5_4.xsd" xmlns:p="tapestry:parameter">
    Colores: <span t:type="any" t:id="span">Cargando...</span> (Event/Json)
</t:container>
```

En el código JavaScript asociado a una página o componente hay que hacer uso del módulo que ofrece el soporte para Ajax y los eventos desde el cliente, con RequireJS se obtiene una referencia a él. Solo es necesario indicar como parámetro el nombre del evento a lanzar, los parámetros si los hubiese y los manejadores de respuesta, tanto en el caso de ser correcta que recibirá los datos devueltos en el servidor como incorrecta. En el archivo ajax.coffee están documentados todos los parámetros que posee la función ajax del módulo t5/core/ajax.

Listado 10.6: event.js

```
define("app/event", ["t5/core/ajax", "jquery"], function(ajax, $) {
    function Colores(spec) {
        var _this = this;

        this.spec = spec;

        setTimeout(function() {
            _this.getColores();
        }, 2000);
    }

    Colores.prototype.getColores = function() {
        var _this = this;

        ajax('getColores', {
            element: $(_this.spec.selector),
            success: function(response) {
                var c = response.json.join();
                $(_this.spec.selector).html(c);
            }
        });
    }

    function init(spec) {
        new Colores(spec);
    }
```

```
    return {
        init: init
    }
});
```

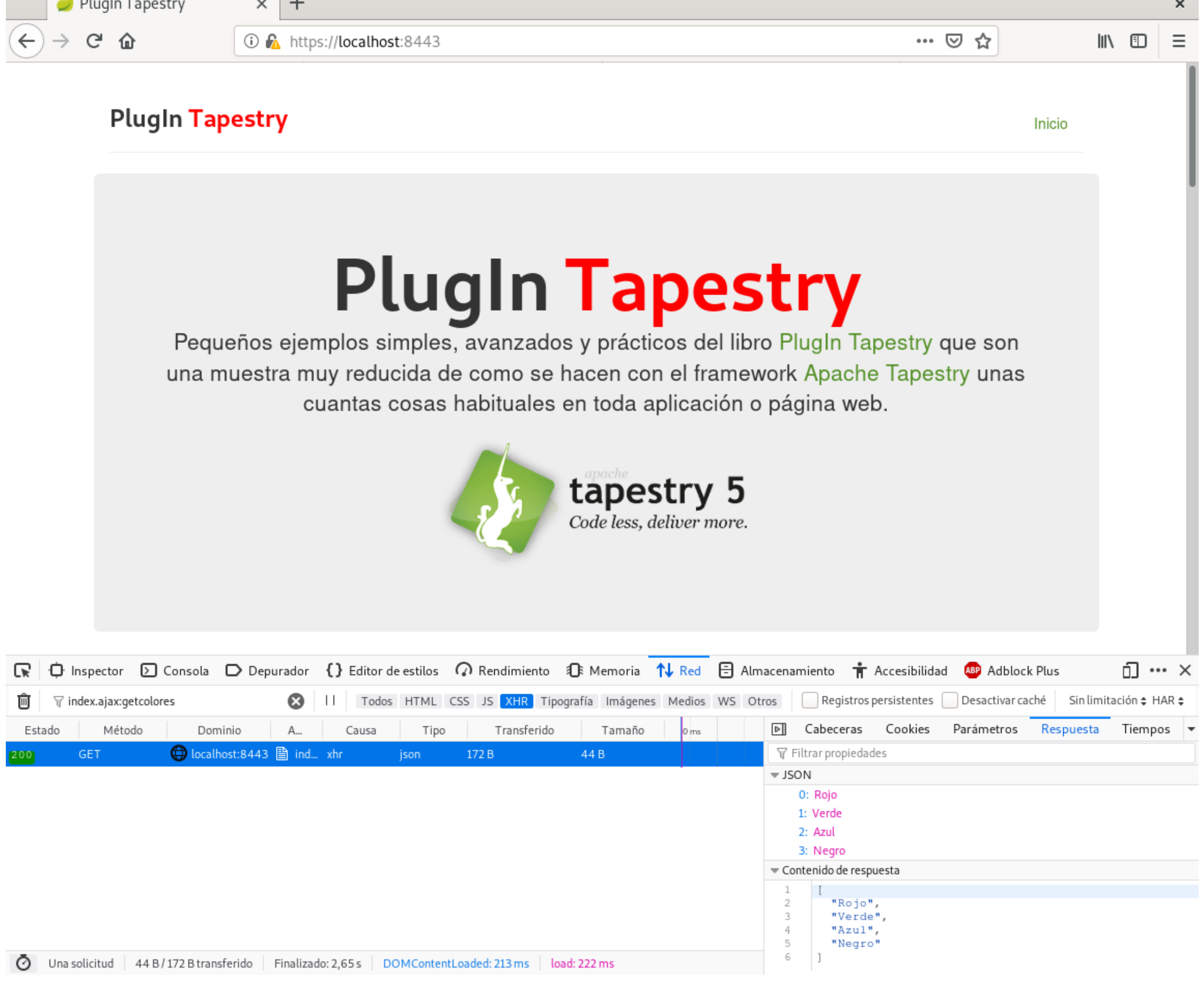
En el primer elemento del HTML se añade un atributo data-componenent-events que contiene la URL necesaria para cada evento que haya sido declarado como lanzable. A partir del elemento indicado en la opción element se busca la URL en el atributo data-componenent-events siguiendo un orden empezando por el propio elemento, en los previos al mismo nivel jerárquicamente empezando por el más cercano desde abajo hacia arriba, en los padres y finalmente en el elemento body.

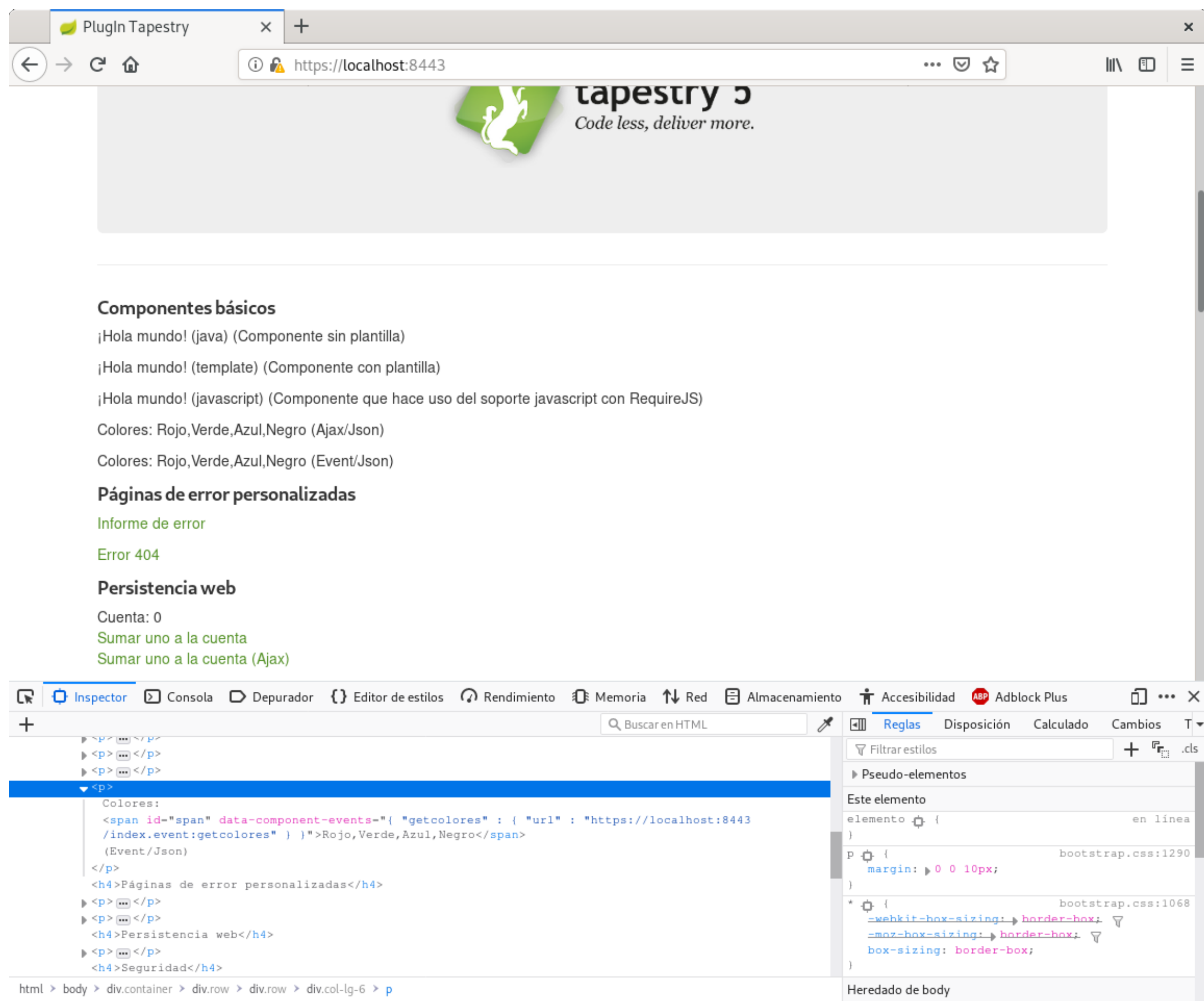
Esta funcionalidad se incorporó en Apache Tapestry 5.2 donde hasta entonces era necesario construir la URL del evento en el servidor con ComponentResources.createEventLink() y enviarlo al componente haciendo uso de JavaScriptSupport como se muestra en el componente Ajax que no hace uso de esta funcionalidad de eventos.

Listado 10.7: Ajax.java

```java
package io.github.picodotdev.plugintapestry.components;

...

/**
 * @tapestrydoc
 */
public class Ajax {

    @Parameter(defaultPrefix = BindingConstants.LITERAL)
    private String selector;

    @Environmental
    private JavaScriptSupport support;

    @Inject
    private ComponentResources componentResources;

    Object onGetColores() {
        return new JSONArray("Rojo", "Verde", "Azul", "Negro");
    }

    protected void afterRender(MarkupWriter writer) {
        String link = componentResources.createEventLink("getColores").toAbsoluteURI();

        JSONObject spec = new JSONObject();
        spec.put("selector", selector);
        spec.put("link", link);
```

```
        support.require("app/ajax").invoke("init").with(spec);
    }
}
```





Cualquiera de las tres formas que no son excluyentes sino complementarias según sea el caso, devolver json y que sea el cliente de presentar los datos en el html, dejando al servidor esa tarea o lanzar eventos desde JavaScript, Tapestry proporciona una gran ayuda ¿no te ha parecido?.

Capítulo 11

Seguridad

Además de la persistencia en una base de datos, otra de las funcionalidades comunes que suele necesitar una aplicación es la seguridad. En la seguridad hay varios aspectos a tratar que son:

- Autenticación: que consiste en identificar al usuario en el sistema y comprobar que el usuario es quien dice ser. Normalmente la autenticación se suele realizar pidiéndole al usuario su identificativo, nombre de usuario o correo electrónico y una contraseña que solo él conoce. Aunque hay otras formas de realizarlo entre ellas los certificados.
- Autorización: que consiste en determinar si el usuario autenticado tienen permisos para realizar una determinada operación. La autorización puede realizarse mediante roles, permisos o una combinación de ambos dependiendo de lo adecuado para la operación. Pero en ocasiones no solo hay que validar si un usuario tiene permisos para para realizar una acción, también puede ser necesario restringir la operación sobre ciertos datos, los que se determinen que él está autorizado a modificar, si no se hiciese esto un usuario podría alterar los datos de otro y el sistema tener una brecha de seguridad. La autenticación y la autorización son solo dos aspectos a considerar pero no son suficientes para considerar una aplicación segura.

Otros aspectos a tener en cuenta son:

- XSS (Cross Site Scripting) y CSRF (Cross-site request forgery).
- Inyección de SQL.
- Conexiones cifradas TLS/SSL.
- Tratamiento información sensible (contraseñas, información personal, tarjetas de crédito).

11.1 Autenticación y autorización

La información de autenticación y autorización puede guardarse en diferentes formas en lo que se conocen como Realms comúnmente en Java. Algunos Realms puede ser simples archivos de texto plano aunque por su

dificultad de mantenimiento al añadir nuevos usuarios, permisos o roles y que puede requerir un reinicio de la aplicación se suele optar por opciones como una base de datos relacional, un sistema LDAP o una base de datos nosql.

Para tener un sistema seguro no basta con ocultar las opciones que un usuario no puede realizar. Ocultar las opciones está bien de cara a usabilidad pero también hay que realizar las comprobaciones de autorización en el caso de una aplicación web en el servidor, al igual que no basta con realizar las comprobaciones de validación de datos en el cliente con javascript, en ambos casos las comprobaciones hay que hacerlas en el lado del servidor también, de lo contrario nada impediría a un usuario conociendo la URL y datos adecuados a enviar realizar algo que no debería (advertido estás si no quieres que te llamen un sábado de madrugada).

La seguridad puede aplicarse de dos formas o una combinación de ambas:

- De forma declarativa: ya sea mediante anotaciones o en un archivo independiente del código. Esta es la opción preferida ya que de esta manera el código de la aplicación no está mezclado con el aspecto de la seguridad.
- De forma programática: si la opción declarativa no no es suficiente para algún caso podemos optar por hacerlo de forma programática, mediante código, con la que tendremos total flexibilidad para hacer cosas más específicas si necesitamos aunque mezclaremos el código de la aplicación con el código de seguridad.

Para aplicar seguridad en una aplicación Java disponemos de varias librerías, entre las más conocidas están:

- Spring Security
- Apache Shiro

Las dos librerías son similares aunque se comenta que Apache Shiro es más fácil de aprender. Además de integraciones con estas librerías Apache Tapestry dispone de módulos para realizar autenticación con servicios de terceros como Facebook, Twitter o sistemas OpenID.

Pero veamos como aplicar seguridad a una aplicación web. En el ejemplo usaré el módulo tapestry-security que a su vez usa Apache Shiro. El ejemplo consiste en una página en la que solo un usuario autenticado podrá poner a cero una cuenta. Para autenticase se usa un formulario aunque perfectamente podría usarse una autenticación BASIC.

Por simplicidad en el ejemplo los usuarios, passwords, roles y permisos los definiré en un archivo de texto, aunque en un proyecto real probablemente usaríamos una base de datos accedida mediante hibernate para lo cual deberíamos implementar unos pocos métodos de la interfaz Realm o si necesitamos autorización la interfaz AuthorizingRealm de Shiro. El archivo shiro-users.properties sería el siguiente:

```
1 # Archivo que contiene los usuarios y contraseñas junto con los permisos y roles
  # Usuarios, passwords y roles
3
  # Usuario «root» con contraseña «password» y rol «root»
```

```
user.root = password,root

# Permisos de los roles # Rol «root» con permiso «cuenta:reset»
role.root = "cuenta:reset"
```

Por una parte se definen los usuarios con su password y roles que posee y por otro se definen que permisos tienen cada rol.

La única configuración que deberemos indicarle a Tapestry es la URL de la página que autenticará a los usuarios y la página a mostrar en caso de que el usuario no esté autorizado para realizar alguna operación y el Realm a usar, lo hacemos añadiendo el siguiente código al módulo de la aplicación:

Listado 11.1: AppModule.java

```
public static void contributeWebSecurityManager(Configuration<Realm> configuration) {
  ExtendedPropertiesRealm realm = new  ExtendedPropertiesRealm("classpath:shiro-users.
    properties");
  configuration.add(realm);
}

//public static void contributeSecurityConfiguration(Configuration<SecurityFilterChain>
    configuration, SecurityFilterChainFactory factory) {
//  configuration.add(factory.createChainWithRegEx("^*/login*$").add(factory.anon()).
    build());
//  configuration.add(factory.createChainWithRegEx("^*/index*$").add(factory.user()).
    build());
//}
```

La página que realiza la autenticación es muy simple, poco más se encarga de recoger el usuario y password introducidos en el formulario de autenticación y a través del Subject realiza el inicio de sesión.

```
package io.github.picodotdev.plugintapestry.pages;

...

public class Login {

  @Property
  private String usuario;

  @Property
  private String password;

  @Inject
  private SecurityService securityService;
```

```
    @Component
    private Form form;

    Object onActivate() {
      // Si el usuario ya está autenticado redirigir a la página Index
      if (securityService.isUser()) {
        return Index.class;
      }
      return null;
     }

     Object onValidateFromForm() {
      if (form.getHasErrors()) {
        return null;
      }

      Subject subject = securityService.getSubject();
      if (subject == null) {
        return null;
      }
      // Recolectar en el token los datos introducidos por el usuario
      UsernamePasswordToken token = new UsernamePasswordToken(usuario, password);
      token.setRememberMe(true);

      try {
        // Validar e iniciar las creadenciales del usuario
        subject.login(token);
      } catch (UnknownAccountException e) {
        form.recordError("Cuenta de usuario desconocida");
        return null;
      } catch (IncorrectCredentialsException e) {
        form.recordError("Credenciales inválidas");
        return null;
      } catch (LockedAccountException e) {
        form.recordError("Cuenta bloqueada");
        return null;
      } catch (AuthenticationException e) {
        form.recordError("Se ha producido un error");
        return null;      }
      // Usuario autenticado, redirigir a la página Index
      return Index.class;
    }
}
```

Una vez configurado el módulo y hecha la página que realiza la autenticación solo debemos usar de forma declarativa las anotaciones que proporciona Shiro, en el caso de que quisiésemos que la página Index solo accedieran los usuarios autenticados usaríamos la anotación @RequiresUser y sobre los métodos @RequiresPermissions

para requerir ciertos permisos para ejecutarlos o @RequiresRoles para requerir ciertos roles. Estas anotaciones podemos usarlas no solo en las páginas y componentes de Tapestry que forman parte de la capa de presentación sino también en los servicios que desarrollemos y que forman la capa de lógica de negocio.

Si las anotaciones no son suficientes podemos hacerlo de forma programática, este es el probable caso de que un usuario solo debería modificar los datos relativos a él sin poder modificar los de otros usuarios. El código variará en función de la forma de determinar si el usuario tiene permisos para un dato. Para comprobar si un usuario tiene ciertos permisos de forma programática debemos usar el objeto Subject que tiene muchos métodos para realizar comprobaciones, como para reinicializar la cuenta se ha de tener el permiso cuenta:reset se debe hacer lo codificado en el método onActionFromReiniciarCuenta:

```
package io.github.picodotdev.plugintapestry.pages;

...

public class Index {

  @Property
  @Persist(value = PersistenceConstants.SESSION)
  private Long cuenta;

  ...

  /**
    * Evento que reinicializa la cuenta.
    */
  @RequiresPermissions("cuenta:reset")
  void onActionFromReiniciarCuenta() throws Exception {
    cuenta = 0l;
  }

  ...
```

En la plantilla de presentación podríamos hacer algunas comprobaciones para mostrar o no el botón para reinicializar la cuenta, podemos comprobar si el usuario autenticado tiene ciertos permisos o tiene un rol.

Listado 11.2: Index.tml

```
<h4>Seguridad</h4>
<p>
  <t:security.haspermission permission="cuenta:reset">
    <a t:id="reiniciarCuenta" t:type="actionLink" class="btn btn-primary btn-mini" style=
    "color: white;">Reiniciar cuenta</a>
  </t:security.haspermission>
</p>
...
```

```
<p>
9   <t:security.hasrole role="root">
      <a t:id="reiniciarCuenta" t:type="actionLink" class="btn btn-primary btn-mini" style=
      "color: white;">Reiniciar cuenta</a>
    </t:security.hasrole>
</p>
```

Para hacer uso de tapestry-security deberemos incluir la librería como dependencia en el archivo build.gradle del proyecto:

```
1 dependencies {
      ...
3
      compile("org.tynamo:tapestry-security:$versions.tapestry_security") { exclude(group:
      'org.apache.shiro') }
      compile "org.apache.shiro:shiro-all:$versions.shiro"

      ...
8 }
```

Para finalizar, a pesar de lo simple del ejemplo pero suficientemente representativo de lo que podría requerir una aplicación real comentar lo sencillo y limpio que es aplicar la seguridad, por una parte gracias al uso de anotaciones y por otra gracias a Tapestry de por si.

11.2 XSS e inyección de SQL

Otros dos aspectos muy a tener en cuenta desde el inicio y durante el desarrollo de una aplicación web son los siguientes:

- XSS (Cross site scripting): es una vulnerabilidad que pueden sufrir las aplicaciones por básicamente no controlar los datos que un usuario envía a través de formularios o como parámetros en las URL. Por ejemplo, supongamos una aplicación recibe un formulario con un nombre que se escupe tal cual se envió en otra página de la aplicación y que otros usuarios pueden visualizar en sus navegadores posteriormente cuando accedan a las páginas que los muestran. Una posible situación puede darse cuando los datos enviados se guardan en una base de datos, un usuario los envía se guardan en la base de datos y otro usuario los ve. Ese dato puede ser una cadena inofensiva como el nombre que se pide pero un usuario malicioso puede enviar código javascript o una imagen con una URL que puede recolectar con cualquier propósito la información de los usuario que la ven en su navegador. Un usuario enviando los datos adecuados puede explotar esta vulnerabilidad y conseguir desde obtener la sesión de otro usuario y hacer cualquier tipo de acción como si fuera ese otro, hasta obtener datos y distribuir virus a los usuarios a través de nuestra propia página web.
- Inyección SQL: esta vulnerabilidad puede ser explotada también por confiar en los valores que envía el usuario pero en vez afectar al html que genera la aplicación web afecta a las base de datos que utilice

la aplicación. Si usamos los parámetros enviados por una fuente no confiable para construir las sql de forma dinámica concatenando trozos de sentencia con los parámetros, un parámetro con el valor adecuado puede modificar completamente la sentencia. Concatenando elementos se puede terminar una sql y hacer cualquier otra a continuación. Las posibilidades de esto es que se podría extraer cualquier dato o borrar completamente la base de datos con sentencias delete o drop. Por ejemplo, hacer esto tiene el problema de la inyección de sql: "select * from producto where id = " + id. Si el parámetro id tuviese el valor «1; delete from producto;» podríamos borrar todos los datos de la tabla.

Por tanto, tanto para evitar fallos de seguridad por XSS y de inyección SQL no se debe confiar en ningún dato enviado por el usuario o de un sistema externo. En realidad en ambos problemas de seguridad la situación es el misma pero que afecta a distintas partes de la aplicación, en un caso a la base de datos (inyección sql) y en otro a la capa de presentación de la aplicación (XSS).

11.3 Cross-site request forgery (CSRF)

Otro problema de seguridad es CSRF (Cross-site request forgery) en el que básicamente un sitio al que se accede devuelve un enlace malicioso que provoca una acción en otro, el atacado. El enlace devuelto puede producir cualquier acción que el sitio atacado permita, el ejemplo que se suele poner es el de un sitio bancario y el intento de hacer una transferencia de la cuenta del usuario que tiene iniciada una sesión en la página de su banco a la cuenta del atacante pero podría ser la realización de un cambio de contraseña a una que conozca el atacante y de esta forma posteriormente este pueda autenticarse con la cuenta de ese usuario en el sitio atacado. A diferencia de XSS donde el usuario confía en lo que obtiene del servidor en el caso de CSRF es al contrario, el servidor confía en las peticiones del cliente, aunque puedan provenir de un sitio malicioso. En la wikipedia este problema de seguridad está más ampliamente explicado con ejemplos, limitaciones y como prevenirlo.

11.4 ¿Que hay que hacer para evitar estos problemas?

Depende del caso. Para evitar XSS todo lo que se emita en el html de la página y se envíe al navegador del usuario ha de ser codificado como html haciendo que si hay un dato malicioso sea inofensivo ya que el navegador no lo interpretará como parte del lenguaje de marcas html sino como texto. Para evitar la inyección de sql si construimos alguna sentencia dinámicamente los parámetros no se han de añadir concatenándolos. En Java con PreparedStatement, y seguro que en cualesquiera otros lenguajes, por un lado va la sql y por otro los parámetros, la clase o API que utilicemos se encargará de ejecutar la sentencia con los parámetros adecuados sin el problema de la inyección sql (además tiene la ventaja de que el código será más legible al no estar mezclada con los parámetros concatenados).

A continuación explicaré que funcionalidades proporciona Tapestry para que las aplicaciones desarrolladas con él sean más seguras en cuanto a XSS.

Para evitar XSS todo lo que se genere como html a partir de datos recuperados de la base de datos y enviados por un usuario hay que escaparlo. Y Tapestry hace eso por defecto por lo que salvo que de forma expresa no

hagamos el escapado no tendremos problemas de XSS. La generación de html se puede hacer de dos formas: en los archivos de plantilla tml o en código Java si se trata de un componente que no tiene plantilla tml asociada.

Con una plantilla tml haremos lo siguiente y el nombre se mostrará escapado en el html:

```
${dato}
```

Para evitar escapado hay que usar la opción:

```
<t:outputraw value="dato">
```

En el componente Java usaremos la clase MarkupWriter y su método write para escapar los valores y el método writeRaw para evitar el escapado si estamos seguros de que no implica un problema de seguridad:

```
witer.write(nombre);
witer.writeRaw(nombre); // ¿Seguro de que no implica un problema de seguridad?
```

Para evitar inyección de SQL usando Hibernate, JPA o la clase PreparedStatement y separando los parámetros de la sql o hql estaremos protegidos. Las buenas prácticas y un ejemplo de mala práctica usando la API de Hibernate, hql y sql para hacer las búsquedas son las siguientes:

```
// Usando HQL
String hql = "from Producto p where p.cantidad < :cantidad";
List productos = session.createQuery(hql).setParameter("cantidad", 10).list();

// Usando JPQL
String jpql = "from Producto p where p.cantidad < :cantidad";
List productos = getEntityManager().createQuery(jpql).setParameter("cantidad", 10).
    getResultList();

// Usando PreparedStatement
PreparedStatement ps = con.prepareStatement("select * from Producto p where p.cantidad <
    ?");
ps.setInteger(1, 10);
ResourSet rs = ps.executeQuery();

// ATENCIÓN: Mala práctica
PreparedStatement ps = connection.prepareStatement("select * from Producto p where p.
    cantidad < " + cantidad);
ResourSet rs = ps.executeQuery();
```

Para el caso de CSRF una opción es generar un token que deben enviar todas las peticiones (enlaces y formularios), ese token se guarda en la sesión y se comprueba que en la siguiente petición sea el mismo, el enlace malicioso no no conoce ese token y estas peticiones consideradas no válidas son rechazadas. Hay que tener en cuenta que tener un problema de XSS puede invalidar la solución CSRF ya que el atacante puede insertar un código javascript que le permita conocer el token.

En Tapestry5CSRF y gsoc2011-csrf-protection se comenta como implementar una solución a CSRF pero no se si siguen siendo válidas. A continuación mostraré como solucionar este problema de CSRF en Tapestry con una combinación de mixin, anotación, advice y objeto de estado de aplicación (SSO), similar a lo explicado en este blog pero con la adición que no solo sirve para formularios sino también para enlaces y el componente BeanEditForm.

Primero veamos el objeto estado de aplicación que contendrá el token (sid) de seguridad, lo generará y lo validará, este objeto de estado de aplicación se guardará a nivel de sesión de modo que el token que se envía en la petición pueda ser validado contra el token guardado en este SSO.

```java
package io.github.picodotdev.plugintapestry.services.sso;

import java.io.Serializable;
import java.util.UUID;

public class Sid implements Serializable {

  private static final long serialVersionUID = -4552333438930728660L;

  private String sid;

  protected Sid(String sid) {
    this.sid = sid;
  }

  public static Sid newInstance() {
    return new Sid(UUID.randomUUID().toString());
  }

  public String getSid() {
    return sid;
  }

  public boolean isValid(String sid) {
    return this.sid.equals(sid);
  }
}
```

El mixin CSRF hará que en los formularios se incluya un campo oculto con el token se seguridad del SSO y en los enlaces se incluya como un parámetro. El nombre del parámetro en ambos casos será t:sid. Este mixin puede ser aplicado a los componentes Form, ActionLink, EventLink y BeanEditForm.

```java
package io.github.picodotdev.plugintapestry.mixins;

...

import io.github.picodotdev.plugintapestry.services.sso.Sid;

@MixinAfter
public class Csrf {

  @SessionState(create = false)
  private Sid sid;

  @Inject
  private Request request;

  @Inject
  private ComponentResources resources;

  @InjectContainer
  private Component container;

  void beginRender(MarkupWriter writer) {
    if (container instanceof EventLink || container instanceof ActionLink) {
      buildSid();

      Element element = writer.getElement();
      String href = element.getAttribute("href");
      String character = (href.indexOf('?') == -1) ? "?" : "&";
      element.forceAttributes("href", String.format("%s%st:sid=%s", href, character, sid.
    getSid()));
    }
  }

  void afterRenderTemplate(MarkupWriter writer) {
    if (container instanceof BeanEditForm) {
      Element form = null;
      for (Node node : writer.getElement().getChildren()) {
        if (node instanceof Element) {
          Element element = (Element) node;
          if (element.getName().equals("form")) {
            form = element;
            break;
          }
        }
      }
      if (form != null) {
        buildSid();
```

```
48          Element e = form.element("input", "type", "hidden", "name", "t:sid", "value", sid
    .getSid());
          e.moveToTop(form);
        }
      }
    }
53
    void beforeRenderBody(MarkupWriter writer) {
      if (container instanceof Form) {
        buildSid();

58      Element form = (Element) writer.getElement();
        form.element("input", "type", "hidden", "name", "t:sid", "value", sid.getSid());
      } else if (container instanceof BeanEditForm) {
        buildSid();

63      Element form = (Element) writer.getElement();
        form.element("input", "type", "hidden", "name", "t:sid", "value", sid.getSid());
      }
    }

68  private void buildSid() {
      if (sid == null) {
        sid = Sid.newInstance();
      }
    }
73 }
```

Creamos una anotación para marcar los manejadores de evento y métodos donde queremos que se aplique la seguridad CSRF.

```
1 package io.github.picodotdev.plugintapestry.services.annotation;
2
  ...

  @Target(ElementType.METHOD)
  @Retention(RetentionPolicy.RUNTIME)
7 @Documented
  public @interface Csrf {
  }
```

Para aplicar la seguridad en los manejadores de evento y métodos marcados con la anotación lo haremos con cierta metaprogramación, con ella comprobaremos que el token del SSO se corresponda con el token enviado por la petición. Esta forma de metaprogramación es lo que en Tapestry se conoce como Advice, la funcionalidad consistirá en que si el token de seguridad es correcto se permite la invocación al método protegido, si el token no es correcto se lanza una excepción impidiendo la llamada al método protegido.

```java
package io.github.picodotdev.plugintapestry.services.workers;

...

public class CsrfWorker implements ComponentClassTransformWorker2 {
  private Request request;
  private ApplicationStateManager manager;

  public CsrfWorker(Request request, ApplicationStateManager manager) {
    this.request = request; this.manager = manager;
  }

  public void transform(PlasticClass plasticClass, TransformationSupport support,
    MutableComponentModel model) {
    MethodAdvice advice = new MethodAdvice() {
      public void advise(MethodInvocation invocation) {
        String rsid = request.getParameter("t:sid");
        Sid sid = manager.getIfExists(Sid.class);
        if (sid != null && sid.isValid(rsid)) {
          invocation.proceed();
        } else {
          invocation.setCheckedException(new CSRFException("El parámetro sid de la
    petición no se corresponde con el sid de la sesión. Esta petición no es válida (
    Posible ataque CSRF)."));
          invocation.rethrow();
        }
      }
    };

    for (PlasticMethod method : plasticClass.getMethodsWithAnnotation(Csrf.class)) {
      method.addAdvice(advice);
    }
  }
}
```

Finalmente, debemos modificar el módulo de nuestra aplicación para dar a conocer a Tapestry el Advice que aplica y comprueba la seguridad con el parámetro t:sid enviado y en el objeto SSO.

```java
@Contribute(ComponentClassTransformWorker2.class)
public static void contributeWorkers(OrderedConfiguration<ComponentClassTransformWorker2>
     configuration) {
  configuration.addInstance("CSRF", CsrfWorker.class);
}
```

Para que en los componentes sea incluido del token de seguridad haremos uso del mixin, es tan simple como añadir el atributo t:mixins y como valor el mixin csrf:

```
<h4>Solución al CSRF</h4>
<p>
  Cuenta: <t:zone t:id="csrfZone" id="csrfZone" elementName="span">${cuenta}</t:zone>
  <div class="row">
    <div class="col-md-4">
      <h5>En formulario</h5>
      <form t:id="csrfForm" t:type="form" t:zone="csrfZone" t:mixins="csrf">
        <input t:type="submit" value="Sumar 1"/>
      </form>
    </div>
    <div class="col-md-4">
      <h5>En enlace</h5>
      <a t:type="eventlink" t:event="sumar1CuentaCsrf" t:zone="csrfZone" t:mixins="csrf">
    Sumar 1</a>
    </div>
    <div class="col-md-4">
      <h5>Fallo seguridad</h5>
      <a t:type="eventlink" t:event="sumar1CuentaCsrf" t:zone="csrfZone" t:parameters="
    prop:{'t:sid':'dummy-attack'}">Sumar 1</a>
    </div>
  </div>
</p>
```

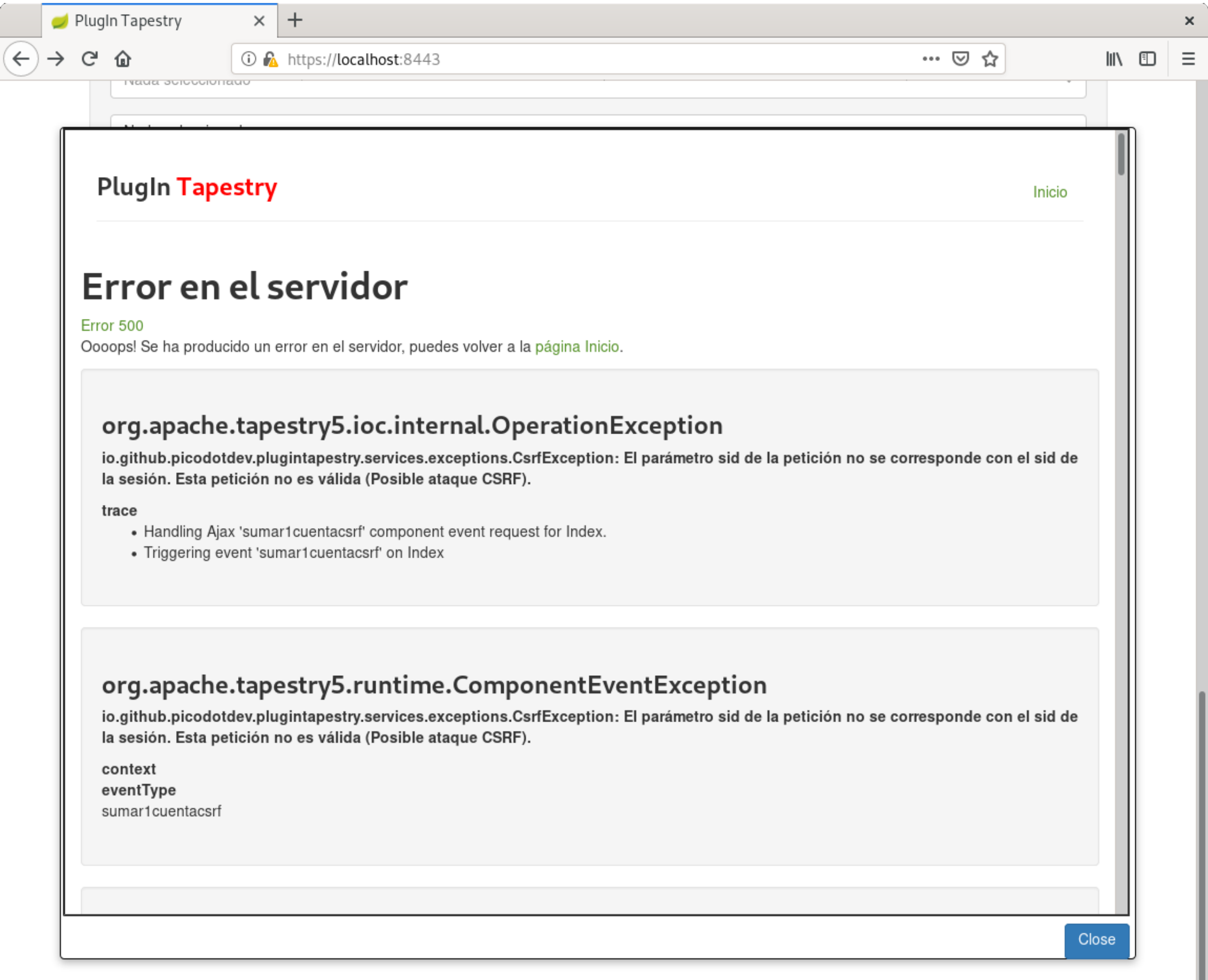
Y para proteger los manejadores de evento con la anotación Csrf de la siguiente manera:

```
@Csrf
void onSuccessFromCsrfForm() {
  cuenta += 1;
  renderer.addRender("zone", zone).addRender("submitOneZone", submitOneZone).addRender("
    csrfZone", csrfZone);
}

@Csrf
void onSumar1CuentaCsrf() {
  cuenta += 1;
  renderer.addRender("zone", zone).addRender("submitOneZone", submitOneZone).addRender("
    csrfZone", csrfZone);
}
```

Con todo esto podemos resolver el problema de seguridad CSRF de forma muy simple y de forma declarativa con una combinación de mixin y anotaciones, solo son necesarios ¡20! caracteres entre ambos.

Esta es la sección de la aplicación del ejemplo funcionando donde puede probarse el mixin y ver la diferencia del comportamiento sin el mixin aplicado.

Cuando pulsamos en el enlace que envía un t:sid inválido mediante un petición ajax provocará el siguiente informe de error con un mensaje descriptivo de lo que ha ocurrido.



En las siguientes imágenes puede verse como se añade el parámetro t:sid a un formulario o enlace al que se le ha aplicado el mixin csrf.

```
<form id="csrfForm" method="post" action="/PlugInTapestry/index.csrfform" data-update-zone="csrfZone" data-validate="submit" >
  <input type="hidden" name="t:formdata"
  value="e/DPMrnb0yKiDn08lKwiZRiZlZg=:H4sIAAAAAAAAAFvzloGlXICBzzMvJbXCqrg0KTezJN6ouIjBIb8oXS+xIDE5I1WvJLEgtbikqNJULzm
  /KDUnMwlI5xbk56XmlRTrBYPlqAQU5SenFheDecXFmfl5M4M
  /SW7d0uLMxMDkw8CRnJMJVO2ZUsIg5JOVWJaon50Yl64fXFKUmZdu7cPAnZqTmgtU4JeYmlrIUMfAWFFQwsABdY4ZgmkEACXlw2qwAAAA"
   ></input>
  <input type="hidden" name="t:sid" value="8b18ca1a-a5ec-4d33-b9b7-f0ef3b1b7288"  ></input>
  <input id="submit_6" class="btn btn-primary" type="submit" name="submit_2" data-submit-mode="normal" value="Sumar 1"  ></input>
</form>
```

```
<a href="/PlugInTapestry/index:sumar1cuentacsrf?t:sid=8b18ca1a-a5ec-4d33-b9b7-f0ef3b1b7288" data-update-zone="csrfZone" >
  Sumar 1
</a>
```

Aparte de resolver el problema de seguridad CSRF quiero destacar con se realiza la metaprogramación en Apache Tapestry con las clases Plastic de Tapestry en la clase CsrfWorker.

11.5 Usar el protocolo seguro HTTPS

El protocolo seguro https hace que los datos que viajan entre el servidor y el cliente a través de internet estén cifrados de modo que nadie más pueda saber cual es es la información intercambiada ni se pueda alterar sin el conocimiento entre las dos partes. Estas propiedades nos son de interés para ciertas partes de una aplicación o en algunos casos la aplicación entera. ¿Cuales son estos casos? Son aquellos en los que queramos garantizar una mayor seguridad, estos pueden ser para proteger usuarios y contraseñas de autenticación para iniciar sesión, ciertos datos sensibles como datos personales, datos de tarjetas de crédito, ... evitando que una tercera parte los obtenga y los utilice para su provecho propio y supongan un problema de seguridad en la aplicación.

Es casi obligatorio forzar a que ciertas páginas de una aplicación o página web funcionen mediante el protocolo seguro https como las páginas de inicio de sesión donde los usuarios se autentican normalmente introduciendo su usuario y contraseña, páginas de compra donde los usuarios introducen los datos de su tarjeta de crédito o algunas secciones de una aplicación como las secciones de las cuentas de los usuarios o un backoffice.

En Tapestry hay varias formas de forzar a que una determinada página use el protocolo seguro de modo que si se accede por el protocolo no seguro http la aplicación obligue a usar https haciendo una redirección. Una de ellas es utilizar la anotación @Secure en las páginas que queramos obligar a usar https. Basta con anotar las clases de las páginas con @Secure y Tapestry automáticamente hará la redirección al protocolo https cuando se acceda con http a la página.

Listado 11.3: Login.java

```java
package io.github.picodotdev.plugintapestry.pages;

...

@Secure
public class Login {
  ...
}
```

Probablemente nos interese configurar el puerto y el host que usará Tapestry al hacer la redirección para que coincidan con el usado en el servidor al que accede el usuario, sobre todo si en la aplicación usamos un servidor web proxy como apache, lighttpd o nginx delante del servidor de aplicaciones donde realmente se ejecuta la aplicación web. El puerto seguro del protocolo https predeterminado es 443 pero en el servidor de aplicaciones tomcat por defecto es 8443. Esto en Tapestry lo indicamos configurando con ciertos símbolos.

Listado 11.4: AppModule.java

```
package io.github.picodotdev.plugintapestry.services;

...

public class AppModule {

  public static void contributeApplicationDefaults(MappedConfiguration<String, Object>
    configuration) {
    ...
    configuration.add(SymbolConstants.SECURE_ENABLED, true);
    configuration.add(SymbolConstants.HOSTPORT, 8080);
    configuration.add(SymbolConstants.HOSTPORT_SECURE, 8443);

    ...
  }

  ...
}
```

Para probar mientras desarrollamos, al menos en nuestro equipo, que la redirección se hace correctamente empleando el plugin de gradle para tomcat podemos hacer que el servidor de desarrollo se inicie con el puerto https disponible. Para usar https se necesita un certificado digital que el plugin de gradle para tomcat se encarga de generar al iniciar la aplicación, aunque sea autofirmado y el navegador alerte que no lo reconoce como firmado un una autoridad en la que confíe, si lo aceptamos podemos acceder a la aplicación sin más problema. Usando gradle la configuración que podemos emplear es:

Listado 11.5: build.gradle

```
...

buildscript {
  repositories {
    mavenCentral()
    jcenter()
  }

  dependencies {
    classpath 'org.gradle.api.plugins:gradle-tomcat-plugin:1.2.4'
  }
}

...

tomcat {
  httpPort = 8080
  httpsPort = 8443
  enableSSL = true
}
```

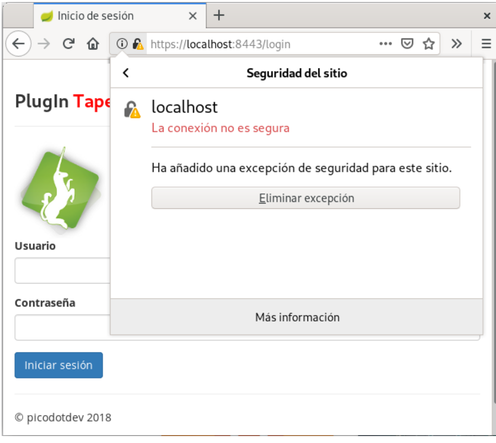
```
...
```

La anotación @Secure en Tapestry es suficiente pero podemos hacer lo mismo empleando Shiro. Integrando Shiro con Tapestry nos permite realizar autenticación y autorización, pero además empleando Shiro también podemos obligar a usar el protocolo https del mismo modo que lo hacemos con la anotación Secure. Cualquiera de las dos formas es perfectamente válida y depende más de cual prefiramos. Con la anotación @Secure deberemos anotar cada página, con Shiro podemos tener centralizado en un único punto en que páginas requerimos https. Con Shiro la configuración se hace con una contribución al servicio SecurityConfiguration y usando el método contributeSecurityConfiguration del módulo y la clase SecurityFilterChainFactory y su método ssl(). Un ejemplo es el siguiente:

Listado 11.6: AppModule.java

```java
package io.github.picodotdev.plugintapestry.services;

...

public class AppModule {

  ....

  public static void contributeSecurityConfiguration(Configuration<SecurityFilterChain>
    configuration, SecurityFilterChainFactory factory) {
    configuration.add(factory.createChain("/admin/**").add(factory.authc()).add(factory.
    ssl()).build());
  }

  ....
}
```

En cualquiera de los dos casos mostrados en este ejemplo se obliga a usar https en la página de login:



Configurar HTTPS en el servidor de aplicaciones

La configuración necesaria para usar el protocolo seguro HTTPS con TLS/SSL varía según el servidor. Aparte de la siguiente configuración deberemos generar un certificado, comprar uno válido por alguna entidad raíz o solicitarlo mediante Lets encrypt.

Spring Boot

Usando Spring Boot hay que proporcionar cierta configuración en el archivo application.yml y configurar el bean TomcatEmbeddedServletContainerFactory para hacer posible el acceso indistinto mediante HTTP o HTTPS. También necesitaremos un archivo keystore que contenga la clave privada y el certificado y en otros servidores el archivo del certificado y la clave privada.

Listado 11.7: AppConfiguration.java

```java
package io.github.picodotdev.plugintapestry.spring;

...

@Configuration
@ComponentScan({ "io.github.picodotdev.plugintapestry" })
@EnableTransactionManagement
public class AppConfiguration {

  ...

    @Bean
    public TomcatEmbeddedServletContainerFactory containerFactory() {
        Connector connector = new Connector("org.apache.coyote.http11.Http11NioProtocol")
    ;
        connector.setScheme("http");
        connector.setPort(8080);

        TomcatEmbeddedServletContainerFactory factory = new
    TomcatEmbeddedServletContainerFactory();
        factory.addAdditionalTomcatConnectors(connector);
        factory.addContextValves(new ValveBase() {
            @Override
            public void invoke(Request request, Response response) throws IOException,
    ServletException {
                getNext().invoke(request, response);
            }
        });
        return factory;
    }

  ...
}
```

Listado 11.8: application.yml

```
server:
    port: 8443
    ssl:
        key-store: classpath:keystore.jks
        key-store-password: secret
        key-password: secret

management:
    port: 8090
    context-path: '/management'

endpoints:
    metrics:
        sensitive: true
    shutdown:
        enabled: true
```

Tomcat

Listado 11.9: server.xml

```
...

<Connector port="8443" protocol="HTTP/1.1" SSLEnabled="true"
    maxThreads="150" scheme="https" secure="true"
    clientAuth="false" sslProtocol="TLS"
    SSLCertificateFile="${catalina.home}/conf/localhost.crt"
    SSLCertificateKeyFile="${catalina.home}/conf/localhost.key"/>

...
```

WildFly

Listado 11.10: standalone.xml

```
...
<security-realms>
    ...
    <security-realm name="SecureRealm">
        <server-identities>
            <ssl>
                <keystore path="localhost.keystore" relative-to="jboss.server.config.dir"
     keystore-password="wildfly"/>
            </ssl>
        </server-identities>
```

```
    </security-realm>
</security-realms>
...
<subsystem xmlns="urn:jboss:domain:undertow:2.0">
    ...
    <server name="default-server">
        <http-listener name="default" socket-binding="http" redirect-socket="https"/>
        <https-listener name="https" socket-binding="https" security-realm="SecureRealm"/
    >
        <host name="default-host" alias="localhost">
            <location name="/" handler="welcome-content"/>
            <filter-ref name="server-header"/>
            <filter-ref name="x-powered-by-header"/>
        </host>
    </server>
    ...
</subsystem>
...
```

nginx

Listado 11.11: nginx.conf

```
...
http {
     server {
          listen       443;
          server_name  localhost;

          ssl                  on;
          ssl_certificate      localhost.pem;
          ssl_certificate_key  localhost.key;

          ssl_session_timeout  5m;

          ssl_protocols  SSLv2 SSLv3 TLSv1;
          ssl_ciphers  HIGH:!aNULL:!MD5;
          ssl_prefer_server_ciphers   on;

          location / {
              root   /usr/share/nginx/html;
              index  index.html index.htm;
          }
     }
}
...
```

Apache

Listado 11.12: httpd.conf

```
LoadModule ssl_module modules/mod_ssl.so

Listen 443
<VirtualHost *:443>
    ServerName www.example.com
    SSLEngine on
    SSLCertificateFile localhost.crt
    SSLCertificateKeyFile localhost.key
</VirtualHost>
```

Lighttpd

Listado 11.13: lighttpd.conf

```
...

$SERVER["socket"] == ":443" {
    ssl.engine = "enable"
    ssl.pemfile = "localhost.pem"
}

...
```

11.6 Salted Password Hashing

Para cada servicio deberíamos emplear una contraseña de una longitud de al menos 8 caracteres que incluya letras en minúscula, mayúscula, números y símbolos, una herramienta que podemos utilizar para generar contraseñas más seguras con los criterios que indiquemos es Strong Password Generator. Sin embargo, recordar cada una de estas contraseñas es muy difícil de modo que es habitual que utilicemos la misma contraseña para varios o todos los servicios y no empleando todos los criterios anteriores. Por otro lado, los desarrolladores no deberíamos guardar en la base de datos las contraseñas que nos entregan los usuarios en texto plano, para evitar guardalas en texto plano hace un tiempo se utilizaba únicamente una función de hashing unidireccional como MD5 o SHA, de este modo si la base de datos fuese comprometida en teoría no podrían conocer la contraseña original. En este artículo comentaré que aún guardando las contraseñas con una función de hashing no es suficiente para hacerlas seguras y comentaré una implementación con Apache Shiro de una de las ideas propuestas.

Algo de teoría y algunas explicaciones

Aunque guardemos las contraseñas con MD5 o alguna variante de SHA hoy en día no es suficiente para que en caso de que alguien obtenga los hashes de las contraseñas de la base de datos pueda averiguarlas o dar con una que genere el mismo hash, usando estas funciones se pueden encontrar colisiones en un tiempo razonable y por tanto ya no se consideran seguras. Dada la computación actual de los procesadores y las tarjetas gráficas una contraseña débil puede romperse usando un ataque de fuerza bruta y quizá antes con un ataque de diccionario que pruebe las más comunes. Muchos usuarios no tienen contraseñas largas ni utilizan letras en minúscula, mayúscula, números y símbolos, muchos usuarios utilizan contraseñas sencillas para ser recordadas más fácilmente, y aún hasheando las contraseñas pueden ser averiguadas. También se pueden usar tablas arcoíris o rainbow tables con los hashes precalculados de las contraseñas de un diccionario con lo que el tiempo empleado para romper una puede requerir poco tiempo de computación.

También hay que tener en cuenta que muchos usuarios usan la misma contraseña para múltiples servicios por lo que basta que alguien obtenga la contraseña original de un servicio y podrá acceder a otros más interesantes para alguien con malas intenciones por mucha seguridad que tenga esos otros servicios, este es uno de los motivos de la autenticación en dos pasos (que emplea algo que sé, la contraseña, y algo que tengo, como el móvil) y la recomendación de usar una contraseña diferente para cada servicio. Las contraseñas por si solas tienen la seguridad más baja de los diferentes servicios donde se usen.

Con Salted Password Hashing se usa en la función de hash y un dato variable denominado salt que añade suficiente entropía y es diferente para cada contraseña, en la base de datos se guarda el resultado de la función de hash junto con el salt, esto es, el resultado de SHA-512(contraseña+ salt) y también el salt. Con Salted Password Hashing el uso de rainbow tables que aceleren el ataque no serían posibles por la entropía añadida por los salt. Aún así conociendo el salt y la función de hash empleada seguiría siendo posible un ataque de fuerza bruta y de diccionario.

Ejemplo de Salted Password Hashing usando Apache Shiro

Antes de comentar alguna opción más que dificulte los ataques de fuerza bruta o de diccionario veamos como implementar Salted Password Hashing empleando Apache Shiro como librería de autenticación y autorización para los usuarios. El ejemplo será simple, sin guardar los datos en una base de datos, pero suficiente para mostrar que se debe añadir al proyecto para que Shiro compruebe las contraseñas usando una función de hash y un salt. Básicamente deberemos crear un nuevo Realm que devuelva los datos del usuario, el hash y el salt. Una implementación suficiente para el ejemplo sería la siguiente, la parte importante está en el método doGetAuthenticationInfo y en la inicialización static de la clase:

Listado 11.14: Realm.java

```
1 package io.github.picodotdev.plugintapestry.misc;
2 
  ...

  public class Realm extends AuthorizingRealm {

7     private static Map<String, Map<String, Object>> users;
```

```
    private static Map<String, Set<String>> permissions;

    // Para hacer más costoso el cálculo del hash y dificultar un ataque de fuerza bruta
    private static final int HASH_ITERATIONS = 5_000_000;

    static {
        // Generar una contraseña de clave «password», con SHA-512 y con «salt» aleatorio
    .
        ByteSource saltSource = new SecureRandomNumberGenerator().nextBytes();
        byte[] salt = saltSource.getBytes();
        Sha512Hash hash= new Sha512Hash("password", saltSource, HASH_ITERATIONS);
        String password = hash.toHex();
        // Contraseña codificada en Base64
        //String password = hash.toBase64();

        // Permissions (role, permissions)
        permissions = new HashMap<>();
        permissions.put("root", new HashSet<>(Arrays.asList(new String[] { "cuenta:reset"
     })));

        // Roles
        Set<String> roles = new HashSet<>();
        roles.add("root");

        // Usuario (property, value)
        Map<String, Object> user = new HashMap<>();
        user.put("username", "root");
        user.put("password", password);
        user.put("salt", salt);
        user.put("locked", Boolean.FALSE);
        user.put("expired", Boolean.FALSE);
        user.put("roles", roles);

        // Usuarios
        users = new HashMap<>();
        users.put("root", user);
    }

    public Realm() {
        super(new MemoryConstrainedCacheManager());

        HashedCredentialsMatcher cm = new HashedCredentialsMatcher(Sha512Hash.
    ALGORITHM_NAME);
        cm.setHashIterations(HASH_ITERATIONS);
        //cm.setStoredCredentialsHexEncoded(false);

        setName("local");
        setAuthenticationTokenClass(UsernamePasswordToken.class);
        setCredentialsMatcher(cm);
```

```
    }

    /**
57   * Proporciona la autenticación de los usuarios.
     */
    @Override
    protected AuthenticationInfo doGetAuthenticationInfo(AuthenticationToken token)
    throws AuthenticationException {
        UsernamePasswordToken atoken = (UsernamePasswordToken) token;
62
        String username = atoken.getUsername();

        if (username == null) { throw new AccountException("Null usernames are not
    allowed by this realm."); }

67      Map<String, Object> user = findByUsername(username);
        String password = (String) user.get("password");
        byte[] salt = (byte []) user.get("salt");
        boolean locked = (boolean) user.get("locked");
        boolean expired = (boolean) user.get("expired");
72
        if (locked) { throw new LockedAccountException("Account [" + username + "] is
    locked."); }
        if (expired) { throw new ExpiredCredentialsException("The credentials for account
     [" + username + "] are expired"); }

        return new SimpleAuthenticationInfo(username, password, new SimpleByteSource(salt
    ), getName());
77  }

    /**
     * Proporciona la autorización de los usuarios.
     */
82  @Override
    protected AuthorizationInfo doGetAuthorizationInfo(PrincipalCollection principals) {
        if (principals == null) throw new AuthorizationException("PrincipalCollection was
     null, which should not happen");

        if (principals.isEmpty()) return null;
87
        if (principals.fromRealm(getName()).size() <= 0) return null;

        // Obtener el usuario
        String username = (String) principals.fromRealm(getName()).iterator().next();
92      if (username == null) return null;
            Map<String, Object> user = findByUsername(username);
            if (user == null) return null;

            // Obtener los roles
```

```
97              Set<String> roles = (Set<String>) user.get("roles");

                // Obtener los permisos de los roles
                Set<String> p = new HashSet<>();
                for (String role : roles) {
102                 p.addAll((Set<String>) permissions.get(role));
                }

                // Devolver el objeto de autorización
                SimpleAuthorizationInfo ai = new SimpleAuthorizationInfo();
107             ai.setRoles(roles);
                ai.setStringPermissions(p);
                return ai;
      }

112         private Map<String, Object> findByUsername(String username) {
            return users.get(username);
        }
    }
```

Las contraseñas hasheadas tendrán la siguiente forma, podemos guardarlas codificadas en formato hexadecimal o en formato Base64:

```
1   Hex: 53
        a8b4b7eb9f5b8a0754916bcf2e11443149e8d0eb933624abf6feec4a8f43799bc177e0817a2a9df204d7c3597a379689

    Base64: U6i0t+ufW4oHVJFrzy4RRDFJ6NDrkzYkq/b+7EqPQ3mbwXfggXoqnfIE18NZejeWifRm+bO/4
        UtTTI2CTO7uIg==
```

Se debe modificar la configuración para que se utilice el nuevo Realm el antiguo guardaba las contraseñas en texto plano (shiro-users.properties).

Listado 11.15: AppModule.java

```
1   ...
    public static void contributeWebSecurityManager(Configuration<Realm> configuration) {
3     // Realm básico
      //ExtendedPropertiesRealm realm = new ExtendedPropertiesRealm("classpath:shiro-users.
        properties");

      // Realm con «salted password hashing» y «salt»
      Realm realm = new io.github.picodotdev.plugintapestry.misc.Realm();
8
      configuration.add(realm);
    }
    ...
```

El cambio de Realm para el usuario no supone ninguna modificación y podrá seguir autenticándose con su misma contraseña. En el ejemplo con root como usuario y password como contraseña.

Este es todo el código que necesitamos para la implementación de contraseñas codificadas con una función de hashing, en este caso SHA-512, y un salt, no es mucho y además es bastante simple la implementación con Shiro y usando el framework Apache Tapestry. Estas pocas líneas de código pueden aumentar notablemente la seguridad de las contraseñas que guardamos en la base de datos. En el caso de que la base de datos se vea comprometida será más difícil para alguien con malas intenciones obtener las contraseñas originales.

El siguiente ejemplo de federatedaccounts puede verse como usar está técnica de hash con salt usando una base de datos. Básicamente es lo mismo pero accediendo a base de datos para obtener el hash de la contraseña y el salt con una entidad JPA.

Otras opciones que añaden más seguridad

Aún así como comento este ejemplo de Salted Password Hashing aunque dificulta un ataque aún es viable usar fuerza bruta o un diccionario. En el artículo Password Security Right Way comentan tres ideas más. Una es usar como función de hash Bcrypt no porque sea más segura que SHA-512 sino porque es más lenta y esto puede hacer inviable la fuerza bruta o de diccionario, hay planes de proporcionar Bcrypt en Apache Shiro en futuras versiones. En el ejemplo como alternativa a Bcrypt se usan varios millones de iteraciones de aplicación de la función para añadir tiempo de cálculo al hash, este tiempo adicional no es significativo en el cálculo de un hash pero en un ataque de fuerza bruta puede aumentarlo de forma tan significativa que sea inviable. La segunda idea interesante es además de hashear la clave es cifrarla de modo que aún habiendo sido comprometida la base de datos se necesite la clave privada de cifrado que también debería ser comprometida para producir el ataque. La tercera es partir el hash y distribuirlo entre varios sistemas de modo que sea necesario romperlos todos para obtener en hash original, lo que dificulta aún más un ataque.

Para cifrar además las contraseñas deberemos proporcionar implementaciones propias de CredentialsMatcher y de SimpleHash de Shiro.

Para terminar mucho de esto es fútil si se permiten contraseñas sencillas por lo que exigir contraseñas con cierta fortaleza de la forma comentada al principio también es necesario si la seguridad de la aplicación es un requisito importante. Por otra parte esta misma técnica puede emplearse para información igual de importante que las contraseñas como serían los datos de las tarjetas de crédito e incluso para la información personal de los usuarios que se registren en la aplicación.

Capítulo 12

Librerías de componentes

Cada aplicación contiene un conjunto de piezas que son reutilizadas varias veces a lo largo de todo el proyecto. Para aprovechar esos componentes en otros proyectos podemos desarrollar una librería de componentes. Lo fantástico de esto es lo fácil que es empaquetar esos componentes y reutilizarlos en varias aplicaciones y el hecho de que esas aplicaciones usando la librería de componentes no necesitan una especial configuración.

Por reutilización se entiende no tener que copiar y pegar código o archivos de un proyecto a otros sino simplemente añadir una dependencia en la herramienta de construcción para el proyecto. Si los componentes aumentan el código que se reutiliza en una aplicación y evita duplicidades que generan problemas en el mantenimiento, las librerías pueden ser muy útiles y producir esa misma reutilización entre diferentes proyectos. Esto sirve tanto para empresas de unas pocas personas que desarrollan muchos proyectos de unos meses a grandes empresas o administraciones públicas que realizan gran cantidad de proyectos de tamaño considerable con una fuerte inversión de capital y personas que de esta manera pueden reaprovechar.

Una librería de componentes consiste principalmente en componentes pero al igual que las aplicaciones tienen un módulo que añadirá o configurará otros servicios que esos componentes necesiten al contenedor IoC. Finalmente, los componentes pueden empaquetarse junto con sus recursos de assets como imágenes, hojas de estilo, catálogos de mensajes, servicios y librerías de javascript que se necesiten entregar al navegador.

Tapestry no impone ninguna herramienta de construcción, en este ejemplo usaré Gradle. La librería no es más que un archivo jar que puede ser generado por cualquier otra herramienta. El ejemplo consiste en un componente que muestra una imagen. Los pasos a seguir para desarrollar una librería son los explicados a continuación.

12.1 Crear una librería de componentes

Para crear una librería de componentes hay que realizar las siguientes acciones.

Elegir un nombre de paquete base

Al igual que las aplicaciones las librerías de componentes tienen un módulo y un paquete base que debe ser único, en este ejemplo será io.github.picodotdev.plugintapestry.libreria. A la librería se le aplican las mismas convenciones y el módulo deberá estar ubicado en el paquete services, los componentes en el paquete componentes y las páginas en el paquete pages del paquete base.

Crear los componentes y páginas

El componente del ejemplo es muy simple pero válido para este asunto.

```
package io.github.picodotdev.plugintapestry.libreria.components;

...

public class Logo {

  @Inject
  @Path("logo.jpg")
  private Asset logo;

  boolean beginRender(MarkupWriter writer) {
    writer.element("img", "src", logo);
    writer.end();
    return false;
  }
}
```

Este componente no tiene nada nuevo que no hayamos visto en el capítulo Páginas y componentes. El recurso relativo a la localización de la clase se inyecta en el componente y se genera una etiqueta img con la imagen.

Una librería real estará compuesta por más componentes y algo más complejos que este.

Selecciona un nombre para la carpeta virtual

Los componentes de la librería al ser usados son referenciados usando un nombre de carpeta virtual. En este ejemplo usaremos libreria como nombre de carpeta. Esto significa que la aplicación puede incluir el componente Logo en una plantilla usando cualquiera de las siguientes opciones:

```
<html xmlns:t="http://tapestry.apache.org/schema/tapestry_5_4.xsd" xmlns:p="tapestry:
    parameter" xmlns:l="tapestry-library:libreria">
  <t:libreria.logo/>
  <img t:type="libreria/logo"/>
  <l:logo/>
</html>
```

Los dos primeros usos del componente logo resultan más pesados de escribir, con la tercera opción y usando un espacio de nombres para la librería se reduce notablemente los caracteres a escribir y hará la plantilla más clara y legible.

Configurar una carpeta virtual

Tapestry necesita conocer donde buscar los componentes según el nombre de carpeta virtual que se usa en las plantillas. Esto se lleva a cabo en la clase del módulo de la librería realizando una contribución al servicio ComponentClassResolver.

```java
package io.github.picodotdev.plugintapestry.libreria.services;

import org.apache.tapestry5.ioc.Configuration;
import org.apache.tapestry5.services.LibraryMapping;

public class LibreriaModule {

  ...

  public static void contributeComponentClassResolver(Configuration<LibraryMapping>
    configuration) {
    configuration.add(new LibraryMapping("libreria", "io.github.picodotdev.plugintapestry
    .libreria"));
  }

  ...
}
```

Configurar el módulo para autocargarse

Con este archivo haremos que al añadir la librería como dependencia de un proyecto Tapestry conozca que se trata de una librería de componentes. De esta forma para usarla bastará con dejarla caer en el classpath del proyecto.

Listado 12.1: gradle.build

```
jar {
  manifest {
    attributes("Tapestry-Module-Classes": "io.github.picodotdev.plugintapestry.libreria.
    services.LibreriaModule")
  }
}
```

Extendiendo el acceso a los assets

La siguiente contribución es necesaria para permitir el acceso a los assets de la librería.

```
public class LibreriaModule {

  ...

  public static void contributeRegexAuthorizer(Configuration<String> configuration {
    configuration.add("^io/github/picodotdev/plugintapestry/libreria/.*\\.png$");
    configuration.add("^io/github/picodotdev/plugintapestry/libreria/.*\\.jpg$");
  }

  ...
}
```

Esta contribución usa una expresión regular para identificar los recursos en el classpath a los que Tapestry permitirá el acceso.

Versionado de assets

Los assets localizados en el classpath, como los empaquetados en una librería .jar de una librería, son expuestos a los navegadores en la carpeta virtual /assets debajo del contexto de la aplicación.

En el ejemplo la imagen es expuesta en la URL /app/assets/[hash]/libreria/components/logo.jpg suponiendo que el contexto de la aplicación es app.

Los assets son servidos con unas cabeceras de expiración lejanas en el futuro lo que permite a los navegadores cachearlas de forma agresiva pero esto causa el problema de que la imagen no se vuelva a pedir cuando exista una nueva versión. Para solventar este problema se usa un hash generado a partir del contenido del asset, cuando este cambie en un nuevo despliegue el hash cambiará y el cliente solicitará el nuevo recurso.

Y eso es todo. La autocarga de la librería junto con las carpetas virtuales para las imágenes que contiene se encargan de todos los problemas. Tú solo tienes que encargarte de construir el JAR, establecer el manifiesto y ponerla como dependencia en el proyecto donde se quiera usar.

12.2 Informe de componentes

Si la librería va a ser utilizada por terceras personas es interesante generar un informe en el que aparezcan cuales son los componentes de la librería, que parámetros tienen, de que tipo, que binding usan por defecto, una descripción de para que sirven y tal vez un ejemplo de uso.

Este informe se genera a partir del Javadoc de la clase Java del componente, sus anotaciones así como de documentación externa en formato xdoc que se puede proporcionar.

Para conocer como generar este informe consulta el apartado Documentación Javadoc.

Capítulo 13

Pruebas unitarias y de integración

Realizar teses unitarios, de integración y funcionales del código de la aplicación que desarrollamos es necesario para tener cierta seguridad de que lo codificado funciona como se espera al menos bajo las circunstancias de las pruebas. Una vez que tenemos un conjunto de pruebas y necesitamos hacer cambios a código existente las pruebas nos sirven para evitar introducir nuevos defectos, tendremos seguridad de que lo modificado sigue funcionando como antes y no dejaremos de hacer algo por miedo a introducir nuevos errores.

A estas alturas supongo que todos estaremos de acuerdo en que las pruebas son de gran utilidad y necesarias. Además, de lo anterior los teses nos sirven como documentación en forma de código de como se puede usar los objetos bajo prueba. Y por otra parte si usamos un lenguaje dinámico, que tan de moda están en estos momentos, en el que el compilador no suele ayudar en tiempo de desarrollo y solo nos encontramos con los errores en tiempo de ejecución porque hemos puesto mal el nombre de un variable, de método, el número o tipo de los parámetros son incorrectos las pruebas nos ayudarán a detectarlos al menos en el entorno de integración continua y no en producción aunque muy posiblemente no siempre porque casi seguro no tendremos el 100% del código cubierto con teses. Si en Java es necesario tener teses en un lenguaje dinámico como Groovy me parece vital si no queremos tener errores en producción por temas de «compilación».

Si desarrollamos código de pruebas debemos tratarlo como un ciudadano de primera clase, esto es, con la misma importancia que el resto del código de la aplicación en el que deberíamos aplicar muchas de las ideas explicadas en el libro Clean Code de forma que el código sea legible y más fácilmente mantenible. No hacerlo puede que haga que las pruebas con el tiempo dejen de tener utilidad y peor aún supongan un problema más.

Para realizar pruebas en Apache Tapestry hay algo de documentación en la propia página del proyecto y en algunas librerías relacionadas pero está esparcida por varias páginas y para alguien que está empezando no es sencillo documentarse e iniciar un proyecto haciendo pruebas desde un inicio de forma rápida. En esta entrada explicaré varias formas de hacer pruebas unitarias, de integración y funcionales y como ejecutarlas de forma cómoda haciendo uso de Gradle, Geb, Spock y JUnit junto con Mockito.

13.1 Pruebas unitarias

En Tapestry realizar pruebas unitarias consiste en probar las páginas y componentes (las páginas en realidad son también componentes y se pueden probar de la misma forma). Dado que las clases de los componentes y páginas son simples POJO (Plain Old Java Object) que no heredan de ninguna clase, no tienen necesidad de implementar ninguna interfaz y no son abstractas, una forma de probarlas es tan simple como crear una una instancia, inyectar las dependencias de las que haga uso el SUT (Subject Under Test, sujeto bajo prueba) y comprobar los resultados. Este es el caso de la prueba realizada en HolaMundoTest, en la que se prueba el método beginRender. Si el componente tuviese otros métodos podrían probarse de forma similar. En este ejemplo se realizan las siguientes cosas:

- Se crean las dependencias de las que haga el sujeto bajo prueba, en este caso un mock del servicio MensajeService que devolverá el mensaje que emitirá el componente. En un ejemplo real podría tratarse de un servicio que accediese a base de datos o se conectase con un servicio externo. El mock se crea haciendo uso de la librería Mockito.
- Se crea la instancia del componente, como la clase del componente no es abstracta es tan sencillo como hacer un new.
- Se inyectan las dependencias. El nombre al que saludará el componente y el mock que devolverá el mensaje que deseamos en la prueba. Para poder inyectar las propiedades de forma sencilla estas propiedades están definidas en el ámbito package, las propiedades de un componente pueden definirse en el ámbito private pero entonces necesitaríamos definir al menos métodos set para asignar valores a esas propiedades.
- Se crea una instancia de un objeto que necesita como parámetro el método bajo prueba beginRender y se le pasa como parámetro.
- El método bajo prueba se ejecuta.
- Finalmente, se comprueba el resultado de la ejecución con un Assert. Como conocemos los datos que ha usado el objeto (los inyectados en las dependencias) bajo prueba conocemos el resultado que debería producir y es lo que comprobamos.

Un ejemplo de este tipo de test es NumeroProductosTest que prueba el componente NumeroProductos.

Listado 13.1: NumeroProductos.java

```
1  package io.github.picodotdev.plugintapestry.components;

   ...
4
   /**
    * @tapestrydoc
    */
   public class NumeroProductos {
9
     @Inject
```

```
  JooqProductoDAO dao;

  @BeginRender
  boolean beginRender(MarkupWriter writer) {
    long numero = dao.countAll();
    writer.write(String.format("Hay %d productos", numero));
    return false;
  }
}
```

Listado 13.2: NumeroProductosTest.java

```
package io.github.picodotdev.plugintapestry.components;

...

public class NumeroProductosTest {

  @Test
  public void conNombre() {
    // Si tuviese alguna propiedad de algún servicio con la anotación @Inject tendríamos
    crear
    // un mock de la dependencia
    JooqProductoDAO dao = Mockito.mock(JooqProductoDAO.class);
    Mockito.when(dao.countAll()).thenReturn(0l);

    // Crear el componente
    NumeroProductos componente = new NumeroProductos();

    // Si tuviese parámetros (anotación @Parameter) deberíamos inyectarlos, para ello
    debemos
    // crear setters o cambiar el ámbito de visibilidad a package (sin ámbito)
    componente.dao = dao;

    // Ejecutar el sujecto bajo prueba
    MarkupWriter writer = new MarkupWriterImpl();
    componente.beginRender(writer);

    // Comprobar el resultado
    Assert.assertEquals("Hay 0 productos", writer.toString());
  }
}
```

La misma prueba usando Spock:

Listado 13.3: NumeroProductosSpec.groovy

```
package io.github.picodotdev.plugintapestry.components

import org.apache.tapestry5.MarkupWriter
```

```
import org.apache.tapestry5.internal.services.MarkupWriterImpl

import io.github.picodotdev.plugintapestry.services.dao.JooqProductoDAO

import spock.lang.Specification

class NumeroProductosSpec extends Specification {

    def conNombre() {
        setup:
        // Si tuviese alguna propiedad de algún servicio con la anotación @Inject
    tendríamos crear
        // un mock de la dependencia
        def dao = Mock(JooqProductoDAO.class)
        dao.countAll() >> 0l

        // Crear el componente
        def componente = new NumeroProductos()

        // Si tuviese parámetros (anotación @Parameter) deberíamos inyectarlos, para ello
     debemos
        // crear setters o cambiar el ámbito de visibilidad a package (sin ámbito)
        componente.dao = dao

        // Ejecutar el sujecto bajo prueba
        def writer = new MarkupWriterImpl()

        when:
        componente.beginRender(writer)

        then:
        // Comprobar el resultado
        "Hay 0 productos" == writer.toString()
    }
}
```

Las pruebas unitarias se ejecutan con:

```
$ ./gradlew test
```

13.1.1 Pruebas unitarias incluyendo código HTML

En un framework web aparte de comprobar el funcionamiento del código Java (u otro lenguaje) es solo una parte de lo que nos puede interesar probar. Un framework web nos puede interesar tener pruebas del código html que se genera, en el caso de Tapestry los componentes o páginas que generan su html con las plantillas .tml o

como en el caso anterior en el método beginRender. Las páginas pueden probarse de forma sencilla haciendo uso de la clase TapestryTester, aunque como las páginas pueden incluir muchos componentes (y tendríamos que inyectar muchas dependencias y mocks) no es lo mejor para hacer pruebas unitarias, las pruebas de las páginas enteras es mejor dejarlo para pruebas funcionales y realizar pruebas unitarias sobre los componentes individuales.

Dado que en Tapestry un componente no puede ser usado sino no es dentro de una página, para probar el html de un componente generado con plantillas .tml de forma unitaria debemos crear un página de pruebas en la que incluimos únicamente ese componente. El componente HolaMundo no tiene una plantilla .tml que genera el html pero esto es indiferente para las pruebas, independientemente de si el componente genera el html con el método beginRender o con un .tml podemos hacer la prueba de la misma forma.

Las cosas que tendríamos que hacer son:

- Crear una página de pruebas en la que insertaremos el componente que queramos probar. Para el ejemplo la página de pruebas es NumeroProductosTest.
- En la prueba unitaria, NumeroProductosTesterTest, disponemos una instancia de TapestryTester. Dado que la creación del TapestryTester va a ser igual para todos los teses que tuviésemos creamos una clase abstracta de la que heredarán todos, AbstractTest. Para crear el TapestryTester necesitaremos indicar el paquete de la aplicación, el nombre de la aplicación, el directorio del contextRoot y los módulos adicionales a cargar. El módulo adicional de pruebas TestModule añadirá las páginas que se usarán para hacer las pruebas como si se tratase de una librería adicional de componentes, esto se hace en el método contributeComponentClassResolver.
- Crear los mocks y dependencias que use el componente. En el método before de NumeroProductosTesterTest se crea el mock del servicio que usa el componente. Con la anotación @ForComponents de la librería Tapestry Testify sobre la propiedad dao del test hacemos que los componentes de la prueba que usen un servicio de interfaz JooqProductoDAO se les inyecte la referencia del mock que hemos creado. En el caso de que el componente tenga parámetros la forma de pasarle el valor que deseamos se consigue inyectando el objeto primeramente en la página de prueba que hace uso del componente y posteriormente hacemos que la página le pase el valor en el momento que lo usa. Dado que se trata de un String, y Tapestry hace las inyecciones de los servicios en función del tipo, debemos darle un nombre único para que el contenedor de dependencias de Tapestry distinga que String queremos inyectar.
- Ejecutar la prueba consistirá en renderizar la página con renderPage.
- Finalmente, la comprobación la realizaremos mediante asserts sobre valores devueltos por objeto Document que representa al DOM de la página.

Un ejemplo de este tipo de test es HolaMundoTesterTest.

Listado 13.4: NumeroProductosTest.java

```
1 package io.github.picodotdev.plugintapestry.components;

  ...
4
```

```
public class NumeroProductosTest {

  @Test
  public void conNombre() {
    // Si tuviese alguna propiedad de algún servicio con la anotación @Inject tendríamos
    crear
    // un mock de la dependencia
    JooqProductoDAO dao = Mockito.mock(JooqProductoDAO.class);
    Mockito.when(dao.countAll()).thenReturn(0l);

    // Crear el componente
    NumeroProductos componente = new NumeroProductos();

    // Si tuviese parámetros (anotación @Parameter) deberíamos inyectarlos, para ello
    debemos
    // crear setters o cambiar el ámbito de visibilidad a package (sin ámbito)
    componente.dao = dao;

    // Ejecutar el sujecto bajo prueba
    MarkupWriter writer = new MarkupWriterImpl();
    componente.beginRender(writer);

    // Comprobar el resultado
    Assert.assertEquals("Hay 0 productos", writer.toString());
  }
}
```

Listado 13.5: NumeroProductosTest.tml

```
<!DOCTYPE html>
<html xmlns:t="http://tapestry.apache.org/schema/tapestry_5_4.xsd">
<head></head>
<body>
  <div id="componente">
    <t:numeroProductos />
  </div>
</body>
</html>
```

Listado 13.6: NumeroProductosTesterTest.java

```
package io.github.picodotdev.plugintapestry.components;

...

public class NumeroProductosTesterTest extends AbstractTest {

  // La forma de probar el html de un componente es incluyendolo en una página. Los
    parámetros se
```

```
    // le pasan al componente a través de la página que se le inyectan como si fuesen
      servicios

    // Propiedades que se inyectarán en la página para las propiedades anotadas con @Inject
      .
11  // Los servicios de las propiedades @Inject se inyectan mediante su interfaz, en caso
      de tener
    // un tipo primitivo de datos o un String inyectar por nombre como en el caso de la
      propiedad
    // nombre.

    // @ForComponents("nombre")
16  // private String nombre;

    @ForComponents
    private JooqProductoDAO dao;

21  @Before
    public void before() {
      // Crear el mock del servicio
      dao = Mockito.mock(JooqProductoDAO.class);
      Mockito.when(dao.countAll()).thenReturn(0l);
26  }

    @Test
    public void ceroProductos() {
      Document doc = tester.renderPage("test/NumeroProductosTest");
31    Assert.assertEquals("Hay 0 productos", doc.getElementById("componente").
      getChildMarkup());
    }
  }
```

Listado 13.7: TestModule.java

```
1 package io.github.picodotdev.plugintapestry.test.services;
2
  ...

  public class TestModule {

7     public static void contributeComponentClassResolver(Configuration<LibraryMapping>
      configuration) {
        configuration.add(new LibraryMapping("test", "io.github.picodotdev.plugintapestry.
      test"));
      }
  }
```

Listado 13.8: AbstractTest.java

```
1 package io.github.picodotdev.plugintapestry.test;
```

```
...

5 public abstract class AbstractTest extends TapestryTest {

    private static final TapestryTester SHARED_TESTER = new TapestryTester("io.github.
      picodotdev.plugintapestry", "app", "src/main/webapp", TestModule.class);

    public AbstractTest() {
10      super(SHARED_TESTER);
    }
}
```

Para probar el componente con un parámetro y sin parámetro en la página de prueba el componente se puede usar varias veces. Según la prueba se obtiene el elemento id que lo contenía (componenteSinNombre, componenteConNombre) para comprobar el resultado.

13.1.2 Pruebas unitarias incluyendo código HTML con XPath

En el caso anterior para hacer las comprobaciones se hace uso del objeto Document el cual se va navegando con su API. Obtener la información necesaria para realizar las comprobaciones no es tarea simple si el html es complejo, el código Java necesario para ello puede complicarse y ser de varias lineas para obtener un simple dato. Con el objetivo de tratar de aliviar este problema se puede hacer uso de la librería Tapestry XPath mediante la cual podremos hacer uso de expresiones XPath sobre el objeto Document que obtenemos como resultado.

Listado 13.9: NumeroProductosXPathTesterTest.java

```
1 package io.github.picodotdev.plugintapestry.components;

3 ...

  public class NumeroProductosXPathTesterTest extends AbstractTest {

    // @ForComponents("nombre")
8   // private String nombre;

    @ForComponents
    private JooqProductoDAO dao;

13  @Before
    public void before() {
      dao = Mockito.mock(JooqProductoDAO.class);
      Mockito.when(dao.countAll()).thenReturn(0l);
    }
18
    @Test
    public void ceroProductos() throws JaxenException {
```

```
    Document doc = tester.renderPage("test/NumeroProductosTest");
    String text = TapestryXPath.xpath("id('componente')").selectSingleElement(doc).
    getChildMarkup();
23    Assert.assertEquals("Hay 0 productos", text);
  }
}
```

13.2 Pruebas de integración y funcionales

Si es posible es mejor realizar pruebas unitarias utilizando alguno de los casos anteriores principalmente porque son más sencillas, pequeñas y menos frágiles (menos propensas a empezar a fallar ante cambios) pero sobre todo porque se ejecutan mucho más rápido y de esta manera podemos lanzarlas muy a menudo en nuestro entorno local según desarrollamos. Si tardasen en ejecutarse mucho al final por no estar parados esperando a que se ejecutasen las pruebas acabaríamos por no ejecutarlas, si este es el caso es recomendable hacer que se ejecuten al menos en un entorno de integración continua (usar Jenkins es una buena opción).

Sin embargo, también hay casos en los que nos puede interesar hacer pruebas funcionales sobre la aplicación probando no pequeñas partes de forma individual sino todas en conjunto. Si vemos necesario realizar este tipo de pruebas funcionales o de aceptación conviene realizarlas sobre las partes importantes o vitales de la aplicación sin querer volver a probar lo ya probado de modo unitario con este tipo de pruebas. Como decía son lentas y frágiles ante cambios y si tenemos muchas nos veremos obligados a dedicar mucho esfuerzo a mantenerlas que puede no compensar.

Para realizar este tipo de pruebas en Tapestry en el siguiente ejemplo haremos uso de Gradle, el plugin de tomcat y el framework de pruebas Geb junto con Spock (que también podríamos haber utilizado para las pruebas unitarias). Para hacer las pruebas con Geb usaremos el lenguaje Groovy. Tradicionalmente hacer pruebas funcionales o de aceptación era una tarea no sencilla comparada con las pruebas unitarias, con la ayuda de Geb y Spock realizaremos pruebas funcionales de una forma bastante simple y manejable.

Con Geb los teses de denominan especificaciones. Haremos una prueba de la página Index de la aplicación que para comprobar si se carga correctamente. Para ello:

- Crearemos las especificación. Una especificación es una clase que hereda de GebSpec. Combinando Geb con Spock y su DSL (Domain Specific Language, Lenguaje específico de dominio) el test del ejemplo se divide en varias partes.
- La parte when se encargará de ejercitar el sujeto bajo prueba, en este caso la página Index.
- En la parte then realizaremos las comprobaciones que determinarán si el test se ejecutó de forma correcta.

Junto con la especificación del test podemos definir como es la página que va a probar el test, esto simplificará enormemente el código del test y es lo que hace que Geb simplifique mucho las pruebas funcionales. Si tuviésemos varios test estos pueden compartir todos ellos las definiciones de las páginas. La página se define creando una clase que extiende de Page. En el caso del ejemplo:

- La propiedad estática url, indica la URL de la página a probar. La aplicación debe estar arrancada previamente a pasar las pruebas de integración o funcionales.
- La propiedad estática at, es una comprobación que realizará Geb para determinar si la página que se obtiene con la URL es la que se espera.
- Y ahora viene lo mejor, en la propiedad estática content, podemos definir los elementos relevantes de la página para la prueba que luego en la especificación del test Geb podremos usar para realizar las comprobaciones. La notación para referirse a los elementos es similar a la utilizada en los selectores de jQuery.

Un ejemplo de este tipo de test es IndexSpec. Otros ejemplos un poco más complejos pueden verse en GoogleSpec y GoogleSearchSpec.

Listado 13.10: GoogleSpec.groovy

```
package io.github.picodotdev.plugintapestry.geb

import geb.spock.GebSpec

class GoogleSpec extends GebSpec {
    def 'go to google'() {
        when:
        go 'http://www.google.es'

        then:
        title == 'Google'
    }
}
```

Listado 13.11: GoogleSearchSpec.groovy

```
package io.github.picodotdev.plugintapestry.geb

import geb.Page
import geb.spock.GebSpec

class GoogleHomePage extends Page {
    static url = 'https://www.google.es/'
    static at = { title == 'Google' }
    static content = {
        searchField { $("input[name=q]") }
        searchButton(to: GoogleResultsPage) { $("input[value='Buscar con Google']", 1) }
    }
}

class GoogleResultsPage extends Page {
    static at = { waitFor { title.endsWith("Buscar con Google") } }
    static content = {
        results(wait: true) { $("div.g") }
        result { index -> return results[index] }
```

```
        resultLink { index -> result(index).find("h3.r a") }
    }
}

class GoogleSearchSpec extends GebSpec {
    def 'go to google'() {
        when:
        to GoogleHomePage
      searchField().value "Chuck Norris"
      searchButton().click()

        then:
      at GoogleResultsPage
      resultLink(0).text().contains("Chuck")
   }
}
```

Listado 13.12: IndexSpec.groovy

```
package io.github.picodotdev.plugintapestry.geb

import geb.Page
import geb.spock.GebSpec

import org.springframework.test.context.ContextConfiguration
import org.springframework.test.context.web.WebAppConfiguration
import org.springframework.boot.test.SpringApplicationConfiguration
import org.springframework.boot.test.SpringApplicationContextLoader
import org.springframework.boot.test.IntegrationTest

import io.github.picodotdev.plugintapestry.spring.AppConfiguration

// Definición de la página índice
class IndexPage extends Page {
    // Localización
    static url = 'http://localhost:8080/'
    // Determinar que se cargó una página
    static at = { title.startsWith('PlugIn') }
    // Definición de los elementos de la página
    static content = {
        meta { $('meta[pagina]') }
    }
}

@ContextConfiguration(loader = SpringApplicationContextLoader.class, classes =
    AppConfiguration.class)
@WebAppConfiguration
@IntegrationTest
class IndexSpec extends GebSpec {
    def 'go to index'() {
```

```
        when:
        to IndexPage

34      then:
        meta.@pagina == 'Index'
    }
}
```

Las pruebas de integración y funcionales se ejecutan con la siguiente tarea aunque deberemos añadirla al proyecto junto con el soporte necesario para Gradle.

```
1 $ ./gradlew integrationTest
```

13.3 Ejecución con Gradle y Spring Boot

Para poder ejecutar las pruebas funcionales mediante Gradle y Spring Boot hay que añadir en aquellos teses que necesiten la aplicación funcionando varias anotaciones para que previamente a su ejecución se inicie la aplicación. Las anotaciones son: ContextConfiguration, WebAppConfiguration y IntegrationTest tal como está mostrado en el caso de prueba IndexSpec.groovy. En el archivo build.gradle se encuentra todo lo necesario a incluir para ejecutar tanto los teses de integración como unitarios como el añadir el plugin de Tomcat para gradle y las dependencias así como la definición de la tarea integrationTest para los teses de integración. El plugin de Tomcat adicionalmente a pasar los teses de integración nos permitirá ejecutar la aplicación sin necesidad de tener que instalar previamente el propio Tomcat, Gradle descargará automáticamente la versión embebida de Tomcat y la aplicación se ejecutará sobre ella.

Listado 13.13: build.gradle

```
1  ...

   dependencies {
4      ...

       // Pruebas unitarias
       testCompile("org.apache.tapestry:tapestry-test:$versions.tapestry") { exclude(group:
       'org.testng'); exclude(group: 'org.seleniumhq.selenium') }
       testCompile("net.sourceforge.tapestrytestify:tapestry-testify:$versions.
       tapestryTestify")
9      testCompile("net.sourceforge.tapestryxpath:tapestry-xpath:$versions.tapestryXpath")
       testCompile("junit:junit:$versions.junit")
       testCompile("org.mockito:mockito-core:$versions.mockito")
       testCompile("org.spockframework:spock-core:$versions.spock")
       testCompile("org.spockframework:spock-spring:$versions.spock")
14
       // Pruebas de integración
```

```
    testCompile("org.gebish:geb-spock:$versions.geb")
    testCompile("org.gebish:geb-junit4:$versions.geb")
    testCompile("org.seleniumhq.selenium:selenium-support:${versions.selenium}")
19  testCompile("org.seleniumhq.selenium:htmlunit-driver:${versions.htmlunitDriver}")
    testCompile("org.springframework.boot:spring-boot-starter-test",
    excludeSpringBootStarterLogging)

    ...
}
24
test {
    // Excluir de los teses unitarios los teses de integración
    exclude '**/geb/*Spec.*'
}
29
task integrationTest(type: Test) {
    group = 'Verification'
    description = 'Runs the integration/functional tests.'
    systemProperty 'geb.driver', 'htmlunit'
34
    // Incluir los teses de integración
    include '**/geb/*Spec.*'
}
```

Con estos añadidos Apache Tapestry tiene poco que envidiar a cualquier framework fullstack, con la ventaja de que tenemos total libertad de elegir ahora y en un futuro las herramientas que más convenientes consideremos para cada tarea evitando estar encadenados a unas determinadas.

Capítulo 14

Otras funcionalidades habituales

Este libro se centra principalmente en el framework Apache Tapestry, que en definitiva es la capa de presentación de una aplicación web pero trata algunos aspectos que aunque son no propios del framework son muy habituales en todas las aplicaciones web como la seguridad y la persistencia de base de datos. También hay muchos otros aspectos adicionales que podemos necesitar tener en cuenta y tratar en una aplicación, este capítulo está centrado en como podemos resolver usando Tapestry funcionalidades que suelen ser habituales en muchas aplicaciones.

14.1 Funcionalidades habituales

14.1.1 Integración con Spring

Primeramente, decir que en cierta medida la funcionalidad proporcionada por el contenedor de dependencias de Tapestry y el contenedor de dependencias de Spring se solapan, ambos proporcionan Inversion of Control (IoC). Pero el contenedor de dependencias de Tapestry tiene algunas ventajas como permitir configuración distribuida, esto hace referencia a que cada librería jar puede contribuir con su configuración al contenedor de dependencias, la configuración en Spring se puede hacer mediante XML o código Java. Usar Java tiene la ventaja de que es más rápido, tenemos la ayuda del compilador para detectar errores y el lenguaje Java es más adecuado para expresar la construcción de objetos, usando XML si hubiese algún error en él no nos daríamos cuenta hasta iniciar la aplicación. Si podemos es preferible usar el contenedor de Tapestry que el de Spring, sin embargo, Spring ofrece un montón de funcionalidades muy útiles y esto nos puede obligar a usar el contenedor de Spring para ciertas funcionalidades. Una de ellas son las transacciones para cumplir con las reglas ACID de las bases de datos relacionales, para ello deberemos definir en el contenedor de Spring (y no en el de Tapestry) los servicios con la lógica de negocio con necesidades transaccionales y las dependencias referidas por esos servicios en la configuración del contexto de Spring. A pesar de todo en los demás casos podemos optar por la opción que prefiramos ya que tanto a los servicios de Spring se les pueden inyectar dependencias del contenedor de Tapestry y, el caso contrario, a los servicios de Tapestry se les pueden inyectar servicios de Spring. Veamos en código un ejemplo de como conseguir integración entre Tapestry y Spring.

Sin la integración con Spring usamos el filtro org.apache.tapestry5.TapestryFilter para que Tapestry procese las peticiones que llegan a la aplicación, integrándonos con Spring usaremos un filtro especial, org.apache.tapestry5.spring.Tape Usando Spring Boot podemos evitar incluso disponer del archivo web.xml tradicional de las aplicaciones web Java, en el contenedor de Spring definiendo el bean ServletContextInitializer añadimos el filtro de forma programática.

Listado 14.1: AppConfiguration.java

```
package io.github.picodotdev.plugintapestry.spring;

...

@Configuration
@ComponentScan({ "io.github.picodotdev.plugintapestry" })
@EnableTransactionManagement
public class AppConfiguration {

  ...

    @Bean
    public ServletContextInitializer initializer() {
        return new ServletContextInitializer() {
            @Override
            public void onStartup(ServletContext servletContext) throws ServletException
    {
                servletContext.setInitParameter("tapestry.app-package", "io.github.
    picodotdev.plugintapestry");
                servletContext.setInitParameter("tapestry.use-external-spring-context", "
    true");
                servletContext.addFilter("app", TapestrySpringFilter.class).
    addMappingForUrlPatterns(EnumSet.of(DispatcherType.REQUEST, DispatcherType.ERROR),
    false, "/*");
                servletContext.setSessionTrackingModes(EnumSet.of(SessionTrackingMode.
    COOKIE));
            }
        };
    }

  ...

}
```

También en la clase AppConfiguration.java empleamos código Java como la configuración del DataSource, la SessionFactory y la anotación EnableTransactionManagement para las transacciones que usará Hibernate para la conexión con la base de datos.

Listado 14.2: AppConfiguration.java

```
package io.github.picodotdev.plugintapestry.services.spring;

```

```
...

@Configuration
@ComponentScan({ "io.github.picodotdev.plugintapestry" })
@EnableTransactionManagement
public class AppConfiguration {

  @Bean(destroyMethod = "close")
  public DataSource dataSource() {
    BasicDataSource ds = new BasicDataSource();
    ds.setDriverClassName("org.h2.Driver");
    ds.setUrl("jdbc:h2:mem:test");
    ds.setUsername("sa");
    ds.setPassword("sa");
    return ds;
  }

  @Bean
  public LocalSessionFactoryBean sessionFactory(DataSource dataSource) {
    LocalSessionFactoryBean sf = new LocalSessionFactoryBean();
    sf.setDataSource(dataSource);
    sf.setPackagesToScan("io.github.picodotdev.plugintapestry.entities");
    sf.setHibernateProperties(getHibernateProperties());
    return sf;
  }

  @Bean
  public ProductoEventAdapter productoEventAdapter() {
    return new ProductoEventAdapter();
  }

  @Bean
  public ProductoDAO productoDAO(SessionFactory sessionFactory) {
    return new ProductoDAOImpl(sessionFactory);
  }

  @Bean
  public DummyService dummyService() {
    return new DummyService();
  }

  private Properties getHibernateProperties() {
    Map<String, Object> m = new HashMap<>();
    m.put("hibernate.dialect", "org.hibernate.dialect.HSQLDialect");
    m.put("hibernate.hbm2ddl.auto", "create");
    // Debug
    m.put("hibernate.generate_statistics", true);
    m.put("hibernate.show_sql", true);
```

```
52	    Properties properties = new Properties();
    properties.putAll(m);
    return properties;
  }
}
```

Listado 14.3: DummyService.java

```
1	package io.github.picodotdev.plugintapestry.services.spring;

3	import io.github.picodotdev.plugintapestry.entities.Producto;

public class DummyService {

  public void process(String action, Object entity) {
8	    if (entity instanceof Producto) {
      Producto p = (Producto) entity;
      System.out.println(String.format("Action: %s, Id: %d", action, p.getId()));
    }
  }
13	}
```

Como Spring se encargará de la configuración de Hibernate si incluimos la dependencia tapestry-hibernate tendremos un problema ya que este módulo de Tapestry también intenta inicializar Hibernate. Para evitarlo y disponer de toda la funcionalidad que ofrece este módulo como encoders para las entidades de dominio, la página de estadísticas de Hibernate o el objeto Session como un servicio inyectable en páginas o componentes hay que redefinir el servicio HibernateSessionSource. La nueva implementación del servicio es muy sencilla, básicamente obtiene el la configuración de Hibernate mediante el bean SessionFactory definido en Spring y además mediante el mismo bean se crea el objeto Session que podrá inyectarse en los componentes y páginas de Tapestry en los que lo necesitemos.

Listado 14.4: HibernateSessionSourceImpl.java

```
1	package io.github.picodotdev.plugintapestry.services.hibernate;
2	
...

public class HibernateSessionSourceImpl implements HibernateSessionSource {

7	  private SessionFactory sessionFactory;
  private Configuration configuration;

  public HibernateSessionSourceImpl(ApplicationContext context) {
    this.sessionFactory = (SessionFactory) context.getBean("sessionFactory");
12	
    // http://stackoverflow.com/questions/2736100/how-can-i-get-the-hibernate-
       configuration-object-from-spring
    LocalSessionFactoryBean localSessionFactoryBean = (LocalSessionFactoryBean) context.
       getBean("&sessionFactory");
```

```
    this.configuration = localSessionFactoryBean.getConfiguration();
  }

  @Override
  public Session create() {
    return sessionFactory.openSession();
  }

  @Override
  public SessionFactory getSessionFactory() {
    return sessionFactory;
  }

  @Override
  public Configuration getConfiguration() {
    return configuration;
  }
}
```

También deberemos añadir un poco de configuración en el módulo de la aplicación para redefinir este servicio.

Listado 14.5: AppModule.java

```
package io.github.picodotdev.plugintapestry.services;

...

public class AppModule {

  ...

  // Servicio que delega en Spring la inicialización de Hibernate, solo obtiene la
    configuración de Hibernate creada por Spring
  public static HibernateSessionSource buildAppHibernateSessionSource(ApplicationContext
    context) {
    return new HibernateSessionSourceImpl(context);
  }

  public static void contributeServiceOverride(MappedConfiguration<Class, Object>
    configuration, @Local HibernateSessionSource hibernateSessionSource) {
    configuration.add(HibernateSessionSource.class, hibernateSessionSource);
  }

  ...

  public static void contributeBeanValidatorSource(OrderedConfiguration<
    BeanValidatorConfigurer> configuration) {
    configuration.add("AppConfigurer", new BeanValidatorConfigurer() {
      public void configure(javax.validation.Configuration<?> configuration) {
        configuration.ignoreXmlConfiguration();
```

```
        }
      });
    }

    ...
}
```

Finalmente, debemos añadir o modificar las dependencias de nuestra aplicación. La dependencia tapestry-spring por defecto puede no usar la última versión de Spring, en el ejemplo la sustituyo por una versión más reciente. A continuación incluyo la parte relevante.

Listado 14.6: build.gradle

```
...

dependencies {
    // Tapestry
    compile "org.apache.tapestry:tapestry-core:$versions.tapestry"
    compile "org.apache.tapestry:tapestry-hibernate:$versions.tapestry"
    compile "org.apache.tapestry:tapestry-beanvalidator:$versions.tapestry"

    // Compresión automática de javascript y css en el modo producción
    compile "org.apache.tapestry:tapestry-webresources:$versions.tapestry"
    appJavadoc "org.apache.tapestry:tapestry-javadoc:$versions.tapestry"

    // Spring
    compile ("org.apache.tapestry:tapestry-spring:$versions.tapestry") { exclude(group: '
    org.springframework') }
    compile "org.springframework:spring-jdbc:$versions.spring"
    compile "org.springframework:spring-orm:$versions.spring"
    compile "org.springframework:spring-tx:$versions.spring"

    // Spring Boot
    compile("org.springframework.boot:spring-boot-starter:$versions.spring_boot") {
    exclude(group: 'ch.qos.logback') }
    compile("org.springframework.boot:spring-boot-starter-web:$versions.spring_boot") {
    exclude(group: 'ch.qos.logback') }
    compile("org.springframework.boot:spring-boot-autoconfigure:$versions.spring_boot") {
     exclude(group: 'ch.qos.logback') }
    compile("org.springframework.boot:spring-boot-starter-actuator:$versions.spring_boot"
    ) { exclude(group: 'ch.qos.logback') }

    ...
}

...
```

Una vez disponemos de la integración de Spring con Tapestry podemos hacer que sea Spring el que gestione las transacciones (section 9.4.2).

14.1.2 Plantillas

Una página web está formada por un conjunto de páginas enlazadas entre ellas. Cada página está formado por un html diferente pero normalmente todas las páginas de una misma web comparten el mismo aspecto variando solo una sección donde está el contenido propio de la página. La cabecera de la página, el pie de la página o los menús de navegación suelen estar presentes en todas las páginas de la web y suelen ser los mismos.

En este artículo voy a explicar como crear un componente que nos de a todas las páginas un aspecto común de una aplicación usando apache Tapestry como framework web de tal forma que esa parte común no esté duplicada en la aplicación y pueda ser reutilizada fácilmente. En el caso de Blog Stack las páginas se componen de las siguientes partes.

El esquema de la plantilla será una cabecera, una barra de navegación con enlaces a diferentes secciones de la web, un menú lateral con contenido variable según la página, el contenido que variará según la página y un pie de página. Como todo componente de Apache Tapestry está formado de una clase Java y una plantilla. El componente puede tener diferentes parámetros, y en el caso del de la plantilla muchos para poder variar el contenido por defecto de las diferentes secciones de la página, estos son aside1, aside2, aside3, aside4.

Listado 14.7: Layout.java

```
1 package info.blogstack.components;
2
...

@Import(stack = "blogstack", module = "app/analytics")
public class Layout {
7
  @Parameter(defaultPrefix = BindingConstants.LITERAL)
  @Property(read = false)
  private String title;
```

```
@Parameter(defaultPrefix = BindingConstants.LITERAL)
@Property(read = false)
private String subtitle;

@Parameter(defaultPrefix = BindingConstants.BLOCK)
@Property
private Block aside1;

@Parameter(defaultPrefix = BindingConstants.BLOCK)
@Property
private Block aside2;

@Parameter(defaultPrefix = BindingConstants.BLOCK)
@Property
private Block aside3;

@Parameter(defaultPrefix = BindingConstants.BLOCK)
@Property
private Block aside4;

@Parameter
@Property
private Adsense adsense;

@Property
private String page;

@Inject
ComponentResources resources;

void setupRender() {
  page = resources.getPageName();
}

public int getYear() {
  return DateTime.now().getYear();
}

public String getTitle() {
  if (title == null) {
    return String.format("%s", getSubtitle());
  } else {
    return String.format("%s | %s", title, getSubtitle());
  }
}

public String getSubtitle() {
  return (subtitle == null) ? "Blog Stack" : subtitle;
}
```

```
62    public String getContentClass() {
        return (isAside()) ? "col-xs-12 col-sm-12 col-md-8 content" : "col-xs-12 col-sm-12
        col-md-12 content";
      }

      public boolean isAside() {
67      return (aside1 != null || aside2 != null || aside3 != null || aside4 != null);
      }
    }
```

El archivo tml asociado al componente plantilla será el que genere el contenido html que se enviará al navegador del usuario. En esta plantilla se incluye una cabecera con el logo de la aplicación y una frase que lo describe, posteriormente está una barra de navegación con varios enlaces, con <t:body/> se incluye el contenido propio de la página que usa el componente plantilla y usando el componente <t:delegate/> se incluye el contenido de los diferentes bloques aside si se han personalizado en el uso de la plantilla, con el componente <t:if test="aside"> se comprueba si hay algún aside usándose el método isAside de la clase Layout asociada al componente plantilla y del tml. Finalmente, está el pie que será común a todas las páginas que usen este componente.

Listado 14.8: Layout.tml

```
1   <!DOCTYPE html>
    <html xmlns="http://www.w3.org/1999/xhtml" xmlns:t="http://tapestry.apache.org/schema/
        tapestry_5_4.xsd" xmlns:p="tapestry:parameter">
    <head>
      <meta http-equiv="Content-Type" content="text/html; charset=utf-8"/>
      <meta t:type="any" t:pagina="${page}" />
6     <title>${title}</title>
      <!-- Resources -->
      <link href="//fonts.googleapis.com/css?family=Open+Sans:400,700" rel="stylesheet" type=
        "text/css"/>
      <link href="//netdna.bootstrapcdn.com/font-awesome/4.0.3/css/font-awesome.css" rel="
        stylesheet" type="text/css"/>
      <link href="/feed.atom.xml" rel="alternate" type="application/atom+xml" title="Portada"
        />
11    <link href="${context:images/favicon.png}" rel="icon" type="image/png"/>
    </head>
    <body>
      <header>
        <div class="container-fluid">
16        <div class="row">
            <div class="col-xs-12 col-sm-12 col-md-4">
              <h1><a t:type="pagelink" page="index" class="blogstack"><span class="glyphicon
        glyphicon-th"></span> Blog <span class="stack">Stack</span></a></h1>
            </div>
            <div id="horizontalSkycraper" class="col-xs-12 col-sm-12 col-md-8"></div>
21        </div>
          <div class="row">
            <div  class="col-xs-12 col-sm-12 col-md-12">
```

```
        <h4>Un poco más que un agregador/planeta de bitácoras sobre programación,
  desarrollo, software libre, gnu/linux, tecnología, ...</h4>
       </div>
26    </div>
   </div>
 </header>

 <div class="container-fluid">
31  <div class="row">
     <div class="col-xs-12 col-sm-12 col-md-12">
       <nav role="navigation">
         <ul class="nav nav-pills menu">
           <li><a t:type="pagelink" page="index">Inicio</a></li>
36         <li><a t:type="pagelink" page="archive" context="[]">Archivo</a></li>
           <li><a t:type="pagelink" page="faq">Preguntas frecuentes</a></li>
         </ul>
       </nav>
     </div>
41  </div>
 </div>

 <div class="container-fluid">
   <div class="row">
46   <div t:type="any" class="prop:contentClass"><t:body /></div>
     <t:if test="aside">
       <aside class="col-xs-12 col-sm-12 col-md-4">
         <t:socialnetworks/>
         <t:if test="aside1">
51             <t:delegate to="aside1"/>
         </t:if>
         <div id="bigRectangle"></div>
         <t:if test="aside2">
           <t:delegate to="aside2"/>
56       </t:if>
         <div class="row">
           <div class="col-xs-3 col-md-2">
               <div id="wideSkycraper"></div>
             </div>
61         <t:if test="aside3">
             <div class="col-xs-3 col-md-2">
               <t:delegate to="aside3"/>
             </div>
           </t:if>
66       </div>
           <t:if test="aside4">
           <t:delegate to="aside4"/>
         </t:if>
       </aside>
71   </t:if>
```

```
    </div>
  </div>

  <footer>
76    <div class="container-fluid">
      <div class="row">
        <div class="col-xs-12 col-sm-12 col-md-12">
          <div class="footer">
            <a t:type="pagelink" page="index">Blog Stack</a> por <a href="https://twitter
    .com/picodotdev/">pico.dev</a> está publicado bajo la licencia de software libre <a
    href="http://www.gnu.org/licenses/agpl-3.0.html">GNU Affero General Public</a>.<br/>
81            El contenido agregado conserva la licencia de su bitácora.<br/>
            «Powered by» <a href="https://github.com/picodotdev/blogstack">Blog Stack</a>
    , <a href="http://tapestry.apache.org/">Apache Tapestry</a>, <a href="https://www.
    openshift.com/">OpenShift</a>, <a href="https://pages.github.com/">GitHub Pages</a>,
    <a href="http://www.oracle.com/es/technologies/java/overview/index.html">Java</a> y
    más software libre o de código abierto, inspirado en <a href="http://octopress.org/">
    Octopress</a>.<br/>
            <span class="copyleft">&copy;</span> pico.dev ${year}
          </div>
        </div>
86      </div>
    </div>
  </footer>

  <div id="fb-root"></div>
91  <t:ads adsense="adsense"/>
</body>
</html>
```

Para terminar nos queda ver como sería usar este componente en una página donde queremos usarlo. En la etiqueta html se usa la plantilla con t:type para indicar que esa etiqueta es un componente de Tapestry y se le pasan los aside1 y aside2 que en esta página tienen contenido propio. El contenido de la etiqueta html se sustituirá por la etiqueta <t:body/> de la plantilla, el contenido incluido en los componentes <t:block/> aunque esté dentro de la etiqueta html solo se mostrará cuando se haga uso de un <t:delegate/>, como se hace el componente plantilla. Este es el caso de la página índice de Blog Stack. A pesar de todo el contenido que genera y solo consta de 34 líneas de código, esto muestra lo fácil que es en Tapestry dividir las diferentes partes de una página en componentes que puede ser reutilizados.

Listado 14.9: Index.tml

```
1 <html t:type="layout" t:aside1="aside1" t:aside2="aside2" xmlns:t="http://tapestry.apache
    .org/schema/tapestry_5_4.xsd" xmlns:p="tapestry:parameter">
2
<t:data/>

<t:loop source="posts" value="post">
  <t:postcomponent post="post" excerpt="true"/>
7 </t:loop>
```

```
<section class="index-pagination">
  <div class="container-fluid">
    <div class="row">
      <div class="col-xs-4 col-sm-4 col-md-4">
        <t:if test="!lastPage">
          <a t:type="pagelink" page="index" context="nextContext"><span class="glyphicon
    glyphicon-arrow-left"></span> Más antiguo</a>
        </t:if>
      </div>
      <div class="col-xs-4 col-sm-4 col-md-4 col-xs-offset-4 col-sm-offset-4 col-md-
    offset-4 text-right">
        <t:if test="!firstPage">
          <a t:type="pagelink" page="index" context="previusContext">Más nuevo <span
    class="glyphicon glyphicon-arrow-right"></span></a>
        </t:if>
      </div>
    </div>
  </div>
</section>

<t:block id="aside1">
  <t:feeds/>
</t:block>

<t:block id="aside2">
  <t:lastposts/>
  <t:lastsourceswithposts/>
</t:block>
</html>
```

Usando el mismo componente podemos darle un aspecto común pero variando el contenido de las diferentes secciones. En este caso usamos la misma plantilla donde se muestra la misma cabecera, enlaces de navegación y pie de página pero sin el contenido lateral como en el caso de la página de preguntas frecuentes de Blog Stack, en este caso no usamos los componentes aside.

Listado 14.10: Faq.tml

```
<html t:type="layout" t:title="Preguntas frecuentes" xmlns:t="http://tapestry.apache.org/
    schema/tapestry_5_4.xsd" xmlns:p="tapestry:parameter">

<article class="text-justify">
  <header>
    <h1>Preguntas frecuentes</h1>
  </header>

  <h2>¿Qué es Blog Stack?</h2>

  <p>Blog Stack (BS) es una agregador, planeta, o fuente de información de bitácoras
    sobre programación, desarrollo, desarrollo ágil, software, software libre, hardware,
```

```
        gnu/linux o en general temas relacionados con la tecnología.</p>

    <h2>¿Por qué otro agregador?</h2>

    <p>
        Hay varios motivos, la semilla es que quería hacer un proyecto personal con cierta
        utilidad para otras personas empleando de alguna forma el framework para el
        desarrollo de
        aplicaciones web <a href="http://tapestry.apache.org/">Apache Tapestry</a>.
    </p>

    ...
</article>

</html>
```

Por supuesto, podemos crear tantos componentes plantilla como necesitemos en una aplicación y usar uno o otro en función del tipo de página.

14.1.3 Documentación Javadoc

La documentación es una de las cosas más importantes en una aplicación, sin esta es muy complicado desarrollar. El problema de la documentación es que puede no estar sincronizada con lo que hace el código realmente sobre todo si esta documentación está externalizada del código. Una de las mejores cosas de Java es su documentación proporcionada por la herramienta Javadoc que es generada a partir de los comentarios incluidos del propio código de modo que es más sencillo mantener la documentación sincronizada con lo que hace el código. El resultado es una colección de archivos html con enlaces en los que podemos ver los paquetes, clases, métodos, parámetros y comentarios descriptivos con los que desarrollar nos será más fácil y lo haremos más rápido. Gracias a los IDE como eclipse que ofrece asistencia contextual y nos muestra el Javadoc según escribimos código Java no nos será tan necesario acceder al Javadoc pero en otros lenguajes como Groovy con naturaleza dinámica y no fuertemente tipada nos seguirá siendo muy útil.

En Tapestry la mayoría de nuestro tiempo lo emplearemos en usar componentes, nuestros, de tapestry o de otras librerías y necesitaremos conocer sus parámetros, el tipo de los mismos y que hace cada uno de ellos así como un ejemplo de como se usa. De esta manera no necesitaremos revisar el código fuente de los componentes que usemos y evitaremos malgastar tiempo en averiguar como se usan inspeccionando su código fuente. Tapestry lo hace para sus propias clases y componentes y nosotros del mismo modo lo podemos hacer para los nuestros.

La documentación Javadoc la podemos generar con Gradle y el resultado lo tendremos en la carpeta build/docs/javadoc.

```
$ ./gradlew javadoc
```

Para generar la documentación incluyendo la de los componentes deberemos añadir la anotación @tapestry-doc a las clases de nuestros componentes y mixins. Los ejemplos se añaden a partir de un archivo externo del mismo en formato XDoc Maven en la misma localización que la clase Java, con el mismo nombre y con extensión xdoc, XDoc es un formato muy parecido a html. Deberemos incluir el soporte en Gradle para generar la documentación de los componentes.

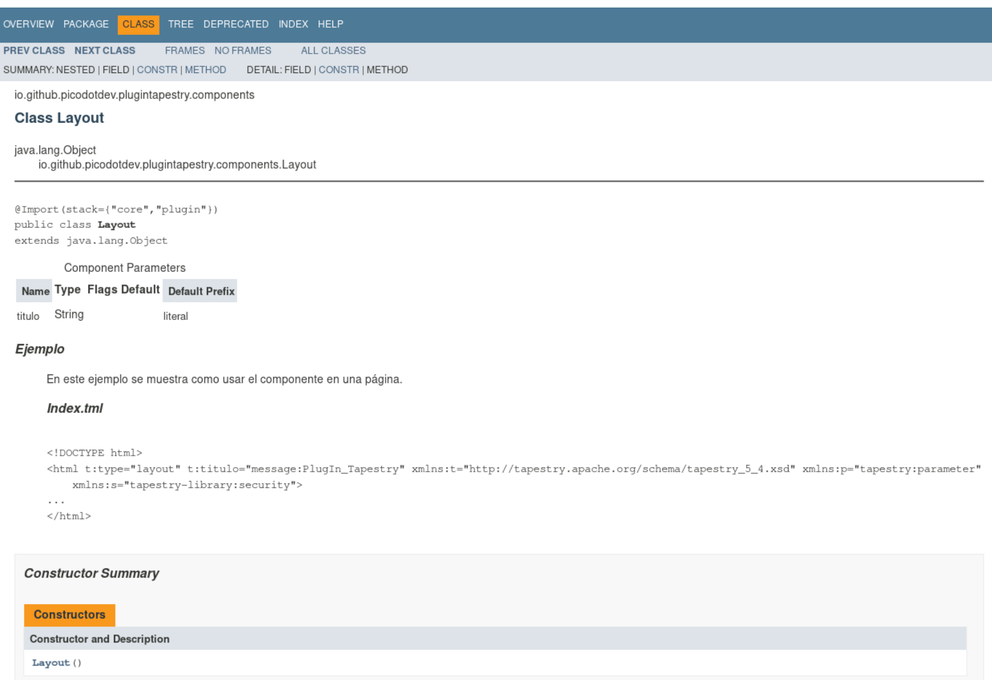
Listado 14.11: build.gradle

```
configurations {
  ...
  appJavadoc
}

dependencies {
  ...
  appJavadoc "org.apache.tapestry:tapestry-javadoc:$versions.tapestry"
  ...
}

task appJavadoc(type: Javadoc) {
  classpath = sourceSets.main.compileClasspath
  source = sourceSets.main.allJava
  destinationDir = reporting.file('appJavadoc')
  options.tagletPath = configurations.appJavadoc.files.asType(List)
  options.taglets = ['org.apache.tapestry5.javadoc.TapestryDocTaglet']

    doLast {
    copy {
      from sourceSets.main.java.srcDirs
      into appJavadoc.destinationDir
      exclude '**/*.java'
      exclude '**/*.xdoc'
      exclude '**/package.html'
    }
    copy {
      from file('src/javadoc/images')
      into appJavadoc.destinationDir
    }
  }
}
```

Ejecutamos esta tarea con:

```
$ ./gradlew appJavadoc
```

El resultado es este:



14.1.4 Páginas de códigos de error y configuración del servidor con Spring Boot

Al desarrollar una aplicación y sobre todo si se trata de una página web pública a la que puede acceder cualquier usuario de internet es casi obligatorio hacer que las páginas de los diferentes códigos de error HTTP tengan el mismo estilo que el resto de las páginas de la aplicación y con el contenido que queramos indicar al usuario para que sepa porque se ha producido ese error y que puede hacer.

En Tapestry se puede redefinir la página de cualquier error HTTP, los más habituales son la del código de error 404 que es la de una página no encontrada o la del código de error 500 que es la página que se mostrará cuando se haya producido un error en el servidor.

Usando Spring Boot y el servidor Tomcat debemos añadir un poco de configuración para el contenedor de servlets. Spring Boot ofrece el bean EmbeddedServletContainerCustomizer con el que podemos personalizar la configuración del servidor que tradicionalmente se hacía en un fichero XML de Tomcat o en el archivo web.xml de la aplicación web. Al definir el filtro de Tapestry indicamos que se encargará de procesar tanto las peticiones REQUEST como las de ERROR.

Listado 14.12: AppConfiguration.java

```java
package io.github.picodotdev.plugintapestry.spring;

...

@Configuration
@ComponentScan({ "io.github.picodotdev.plugintapestry" })
@EnableTransactionManagement
public class AppConfiguration {

```

```
...

    @Bean
    public ServletContextInitializer initializer() {
        return new ServletContextInitializer() {
            @Override
            public void onStartup(ServletContext servletContext) throws ServletException
    {
                servletContext.setInitParameter("tapestry.app-package", "io.github.
    picodotdev.plugintapestry");
                servletContext.setInitParameter("tapestry.use-external-spring-context", "
    true");
                servletContext.addFilter("app", TapestrySpringFilter.class).
    addMappingForUrlPatterns(EnumSet.of(DispatcherType.REQUEST, DispatcherType.ERROR),
    false, "/*");
                servletContext.setSessionTrackingModes(EnumSet.of(SessionTrackingMode.
    COOKIE));
            }
        };
    }

    // Tomcat
    @Bean
    public EmbeddedServletContainerCustomizer containerCustomizer() {
        return new EmbeddedServletContainerCustomizer() {
            @Override
            public void customize(ConfigurableEmbeddedServletContainer container) {
                ErrorPage error404Page = new ErrorPage(HttpStatus.NOT_FOUND, "/error404")
    ;
                ErrorPage error500Page = new ErrorPage(HttpStatus.INTERNAL_SERVER_ERROR,
    "/error500");
                container.addErrorPages(error404Page, error500Page);
            }
        };
    }

    @Bean
    public TomcatConnectorCustomizer connectorCustomizer() {
        return new TomcatConnectorCustomizer() {
            @Override
            public void customize(Connector connector) {
            }
        };
    }

    @Bean
    public TomcatContextCustomizer contextCustomizer() {
        return new TomcatContextCustomizer() {
            @Override
```

```
            public void customize(Context context) {
            }
        };
54  }

    @Bean
    public TomcatEmbeddedServletContainerFactory containerFactory() {
        Connector connector = new Connector("org.apache.coyote.http11.Http11NioProtocol")
    ;
59      connector.setScheme("http");
        connector.setPort(8080);

        TomcatEmbeddedServletContainerFactory factory = new
    TomcatEmbeddedServletContainerFactory();
        factory.addAdditionalTomcatConnectors(connector);
64      factory.addContextValves(new ValveBase() {
            @Override
            public void invoke(Request request, Response response) throws IOException,
    ServletException {
                getNext().invoke(request, response);
            }
69      });
        return factory;
    }

  ...
```

Una vez personalizados los bean de Spring anteriores deberemos crear las páginas de código de error que no son diferentes de cualquier otra página. Algo que nos puede interesar es distinguir si estamos en el modo producción o en el modo desarrollo para sacar más o menos información. En desarrollo es útil disponer de esa información adicional. El símbolo SymbolConstants.PRODUCTION_MODE nos indica si estamos en modo producción y lo podemos configurar de diferentes formas, una de ellas es como una contribución en el módulo de la aplicación.

Listado 14.13: AppModule.java

```
1 public static void contributeApplicationDefaults(MappedConfiguration<String, Object>
      configuration) {
2   configuration.add(SymbolConstants.PRODUCTION_MODE, false);
    ...
  }
```

Listado 14.14: Error404.java

```
1 package io.github.picodotdev.plugintapestry.pages;

  ...
```

```
/**
 * @tapestrydoc
 */
@SuppressWarnings("unused")
public class Error404 {

  @Property
  @Inject
  private Request request;

  @Property
  @Inject
  @Symbol(SymbolConstants.PRODUCTION_MODE)
  private boolean productionMode;
}
```

Listado 14.15: Error404.tml

```
<!DOCTYPE html>
<html t:type="layout" titulo="Página o recursos no encontrado" xmlns:t="http://tapestry.
    apache.org/schema/tapestry_5_4.xsd" xmlns:p="tapestry:parameter">

<div>
  <h1 class="error">Página o recurso no encontrado</h1>

  <h7> <a href="http://en.wikipedia.org/wiki/List_of_HTTP_status_codes" target="_blank">
    Error 404</a> </h7>

  <p>
    Oooops! Parece que lo que estás buscando ya no se encuentra en su sitio prueba a
    encontrarlo desde la <a t:type="pageLink" page="index">página Inicio</a>.
  </p>

  <t:if test="!productionMode">
    <div>
      <h2>Información</h2>
      <t:renderobject object="request" />
    </div>
  </t:if>
</div>
</html>
```

La página de cualquier otro código de error HTTP sería similar a esta. Si quisiéramos mostrar una página de estas en algún momento en la lógica de un componente deberemos retornar una respuesta con un código de error HTTP (Ver Peticiones de eventos de componente y respuestas):

Figura 14.1: Página de error HTTP 404

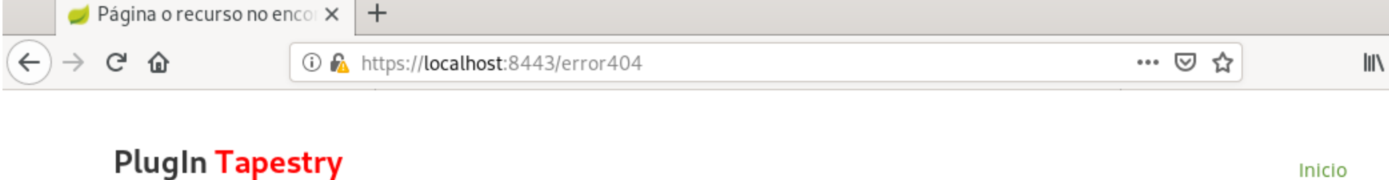
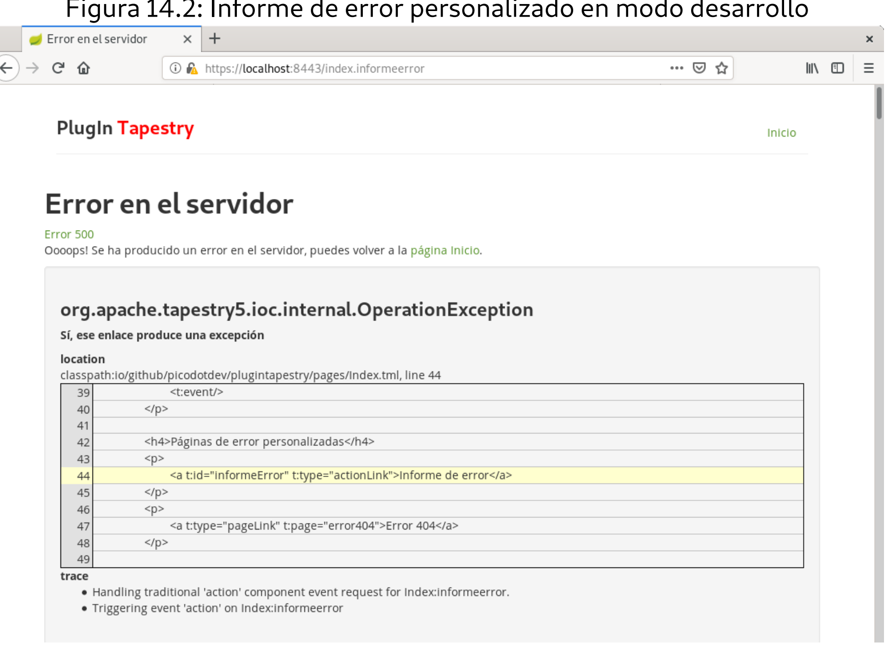


Figura 14.2: Informe de error personalizado en modo desarrollo




```java
Object onAction() {
  ...
    return new HttpError(HttpServletResponse.SC_NOT_FOUND, "Página no encontrada");
  ...
}
```

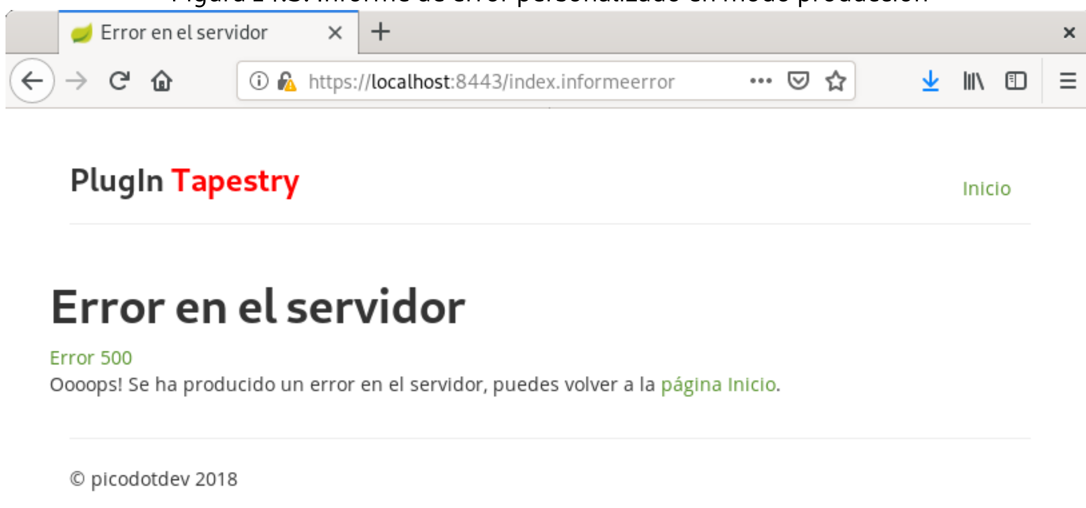
14.1.5 Página de informe de error

Si en en el apartado anterior explicaba como redefinir las páginas de los códigos de error HTTP en Apache Tapestry para personalizarlas y que tuviesen el mismo estilo que las páginas del resto de la aplicación. En este punto explicaré como redefinir la página de informe de error que se muestra cuando se produce una excepción y no es controlada.

En Tapestry se distingue entre la página de código de error HTTP 500 y la página del informe de excepción, aunque esta último también devuelve un código de error HTTP 500 se trata de otra página diferente. Esta página de informe de error también se muestra cuando se produce una excepción en una petición Ajax pero en una ventana emergente dentro de la misma página.

Como el aspecto por defecto no estará acorde con el resto de las páginas de la aplicación deberemos personalizarla. La página de informe de error tanto para las peticiones normales como Ajax tiene el siguiente aspecto ya personalizada en modo desarrollo y en modo producción.

Figura 14.3: Informe de error personalizado en modo producción



Tapestry proporciona mucha de la información que dispone acerca de la petición que nos servirá para corregir el error de forma más rápida. Esta es una muy buena característica que posee Tapestry desde el año 2001, en otros frameworks habitualmente cuando se produce una excepción solo se te informa con la pila de llamadas de la excepción o exception stack trace que a veces no es suficiente para poder reproducir el problema. Por ejemplo, dispondremos de los parámetros que se enviaron en la petición y si la excepción se produce por ciertos valores o una combinación de ellos podremos reproducir el problema mucho más rápidamente que con solo la pila de llamadas de la excepción. Adicionalmente dispondremos de cierta información acerca del entorno en el que se está ejecutando la aplicación que a veces también nos ayudará a detectar el problema y en modo desarrollo un extracto del código fuente de la plantilla donde se produjo el error.

Creando en nuestra aplicación una página de nombre ExceptionReport bastará para que se muestre la nuestra personalizada en vez de la de Tapestry por defecto. En ella distinguiremos entre el modo de desarrollo y producción, en el modo producción no mostraremos todos los detalles de la petición para no dar información interna de la aplicación al usuario.

Listado 14.16: ExceptionReport.java

```
1  package io.github.picodotdev.plugintapestry.pages;

   ...

5  /**
    * @tapestrydoc
    */
   public class ExceptionReport implements ExceptionReporter {

10   private static final String PATH_SEPARATOR_PROPERTY = "path.separator";
```

```
    // Match anything ending in .(something?)path.
    private static final Pattern PATH_RECOGNIZER = Pattern.compile("\\..*path$");

15  private final String pathSeparator = System.getProperty(PATH_SEPARATOR_PROPERTY);

    @Property
    private String attributeName;

20  @Inject
    @Property
    private Request request;

    @Inject
25  @Symbol(SymbolConstants.PRODUCTION_MODE)
    @Property(write = false)
    private boolean productionMode;

    @Inject
30  @Symbol(SymbolConstants.TAPESTRY_VERSION)
    @Property(write = false)
    private String tapestryVersion;

    @Inject
35  @Symbol(SymbolConstants.APPLICATION_VERSION)
    @Property(write = false)
    private String applicationVersion;

    @Property(write = false)
40  private Throwable rootException;

    @Property
    private String propertyName;

45  @Override
    public void reportException(Throwable exception) {
      rootException = exception;
    }

50  public boolean getHasSession() {
      return request.getSession(false) != null;
    }

    public Session getSession() {
55    return request.getSession(false);
    }

    public Object getAttributeValue() {
      return getSession().getAttribute(attributeName);
60  }
```

```
   /**
    * Returns a <em>sorted</em> list of system property names.
    */
65   public List<String> getSystemProperties() {
     return InternalUtils.sortedKeys(System.getProperties());
   }

   public String getPropertyValue() {
70     return System.getProperty(propertyName);
   }

   public boolean isComplexProperty() {
     return PATH_RECOGNIZER.matcher(propertyName).find() && getPropertyValue().contains(
       pathSeparator);
75   }

   public String[] getComplexPropertyValue() {
     // Neither : nor ; is a regexp character.

80     return getPropertyValue().split(pathSeparator);
   }
 }
```

Listado 14.17: ExceptionReport.tml

```
1  <!DOCTYPE html>
   <html t:type="layout" titulo="Error en el servidor" xmlns:t="http://tapestry.apache.org/
       schema/tapestry_5_4.xsd" xmlns:p="tapestry:parameter">
3

   <h1 class="error">Error en el servidor</h1>

   <h7> <a href="http://en.wikipedia.org/wiki/List_of_HTTP_status_codes" target="_blank">
       Error 500</a> </h7>
8
   <p>
     Oooops! Se ha producido un error en el servidor, puedes volver a la <a t:type="pageLink
       " page="index">página Inicio</a>.
   </p>

13 <t:if test="!productionMode">
     <t:exceptiondisplay exception="rootException" />

     <div class="t-env-data">

18     <h2>Tapestry Framework</h2>
       <dl>
         <dt>Tapestry Version</dt>
         <dd>${tapestryVersion}</dd>
```

```
      <dt>Application Version</dt>
23    <dd>${applicationVersion}</dd>
    </dl>

    <h2>Request</h2>
    <t:renderobject object="request" />
28
    <t:if test="hasSession">
      <h2>Session</h2>
      <dl>
        <t:loop source="session.attributeNames" value="attributeName">
33        <dt>${attributeName}</dt>
          <dd>
            <t:renderobject object="attributeValue" />
          </dd>
        </t:loop>
38    </dl>
    </t:if>

    <h2>System Properties</h2>
    <dl>
43    <t:loop source="systemProperties" value="propertyName">
        <dt>${propertyName}</dt>
        <dd>
          <t:if test="! complexProperty">
                        ${propertyValue}
48                      <p:else>
              <ul>
                <li t:type="loop" source="complexPropertyValue" value="var:path">${var:
    path}</li>
              </ul>
            </p:else>
53        </t:if>
        </dd>
      </t:loop>
    </dl>
  </div>
58 </t:if>
</html>
```

La página ExceptionReport es usada cuando no configuramos otra más específica pero podemos asociar una página de excepción distinta para algunas de las excepciones que se produzcan en la aplicación. Deberemos hacer una contribución al servicio RequestExceptionHandler asociando la excepción y la página que queremos mostrar, por supuesto, una misma página de excepción puede asociarse con varias excepciones. Para que la página de excepción sepa cual se ha producido le pasará un parámetro en el contexto de activación con el nombre de la excepción sin la palabra exception final (para una excepción del tipo IllegalArgumentException el parámetro de contexto sería illegalargument).

Listado 14.18: AppModule.java

```
package io.github.picodotdev.plugintapestry.services;

...

public class AppModule {

  ...

  public void contributeRequestExceptionHandler(MappedConfiguration<Class, Class>
    configuration) {
      configuration.add(SQLException.class, Error500.class);
      configuration.add(ValidationException.class, ExceptionReport.class);
  }

  ...
}
```

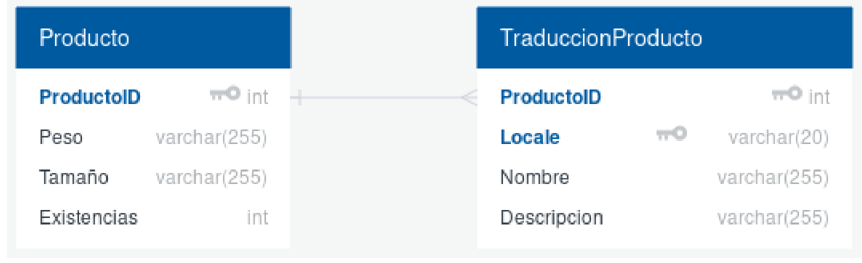
14.1.6 Logging

Disponer de un sistema de logging suele ser imprescindible para tener un registro de las cosas que han sucedido o está sucediendo en la aplicación. Este registro lo podremos consultar y obtendremos cualquier tipo de información que hayamos emitido desde la aplicación, pueden ser desde trazas de excepciones que se hayan producido o información con diferente nivel de detalle que creamos que nos puede ayudar o nos interesa registrar. Una de las mejores opciones disponible en Java es Log4j 2.

14.1.7 Internacionalización (i18n) en entidades de dominio

Si tenemos que internacionalizar los literales de la aplicación probablemente también tengamos que tener en cuenta los nombres, descripciones y textos que guardamos en la base de datos de las entidades de dominio y que puedan aparecer en en el html generado.

Para evitar problemas las buenas prácticas de diseño de las bases de datos dicen que las tablas han de estar normalizadas con lo que debemos evitar crear un campo en la entidad por cada idioma que tengamos, si tenemos muchos idiomas o estos aumentan puede suponer problemas cuando lleguemos a cierta cantidad de campos, en ese caso probablemente tendremos que refactorizar el código y quizá aplicar una solución como la del siguiente diagrama entidad/relación.



Teniendo una entidad Producto que contiene cierta información intenacionalizable asciada como el nombre y descripción se puede crear una tabla dedicada a contener esa información, Traduccion_Producto. Junto con los campos internacionalizados estará el campo idioma o locale que contendrá a que idioma están asociados esos datos. En este modelo cada fila de un producto tendrá varias en la tabla Traduccion_Producto, una por cada idioma. Si se necesita una jerarquía entre locales, es decir, que el locale es_ES herede de es y este a su vez del locale raíz que podría ser la cadena vacía al hacer una búsqueda tener esta información en cuenta en el momento de buscar la información internacionalizada según el locale deseado, la consulta necesita contener clausulas con el identificativo del producto y una clausula in con el conjunto de locales a buscar, una vez encontradas las coinciencias usar el locale más específico que tenga valor.

En este modelo cada entidad a internacionalizar tiene una tabla asociada.

14.1.8 Relaciones jerárquicas en bases de datos relacionales

Otro de los problemas que se suele presentar al trabajar con bases de datos relacionales además de como internacionalizar las entidades del dominio o como hacer búsquedas de texto completo es como modelar las relaciones jerárquicas. Para resolver el problema de las búsquedas en las bases de datos relacionales con datos jerárquicos hay varias soluciones posibles cada una con sus ventajas y desventajas y una más ideal si la base de datos lo soporta, son:

- Listas adyacentes
- Conjuntos anidados
- Variaciones de las anteriores
- Consultas recursivas (necesita soporte de la base de datos)

Listas adyacentes (adjacency lists)

En este modelo se crea una campo adicional que indicará el nodo padre de la relación jerárquica, los nodos raíz tendrán este campo a null al no tener padre.

Buscar los descendientes de un nodo, sin el soporte de queries recursivas y suponiendo una profundidad máxima en la jerarquía de diez se puede conseguir con la siguiente sql:

Listado 14.19: Obtener descendientes de una categoría

```
1 select c1.id as id1, c2.id as id2, c3.id as id3, c4.id as id4, c5.id as id5, c6.id as id6
    , c7.id as id7, c8.id as id8, c9.id as id9, c10.id as id10
  from categoria c1
  left join categoria c2 on c2.categoria_id = c1.id
  left join categoria c3 on c3.categoria_id = c2.id
5 left join categoria c4 on c4.categoria_id = c3.id
```

```
  left join categoria c5 on c5.categoria_id = c4.id
  left join categoria c6 on c6.categoria_id = c5.id
  left join categoria c7 on c7.categoria_id = c6.id
  left join categoria c8 on c8.categoria_id = c7.id
  left join categoria c9 on c9.categoria_id = c8.id
  left join categoria c10 on c10.categoria_id = c9.id
  where c1.id = ?
```

En este caso obtendremos una fila con su jerarquía por cada hoja del árbol. Todo el conjunto de identificativos obtenidos forman los descendientes. Hay que tener en cuenta que en los resultados un identificativo puede aparecer varias veces y con esta consulta el nodo del que se buscan descendientes está incluido.

Buscar los ascendientes se puede hacer de forma similar:

Listado 14.20: Obtener ascendientes de una categoría

```
select c10.id as id10, c9.id as id9, c8.id as id8, c7.id as id7, c6.id as id6, c5.id as
    id5, c4.id as id4, c3.id as id3, c2.id as id2, c1.id as id1
  from categoria c1
  left join categoria c2 on c2.id = c1.categoria_id
  left join categoria c3 on c3.id = c2.categoria_id
  left join categoria c4 on c4.id = c3.categoria_id
  left join categoria c5 on c5.id = c4.categoria_id
  left join categoria c6 on c6.id = c5.categoria_id
  left join categoria c7 on c7.id = c6.categoria_id
  left join categoria c8 on c8.id = c7.categoria_id
  left join categoria c9 on c9.id = c8.categoria_id
  left join categoria c10 on c10.id = c9.categoria_id
  where c1.id = ?
```

Con esta sql obtendremos una fila con los identificativos, c1 será el identificativo del nodo superior y c10 el nodo inferior de la jerarquía.

Con esta solución para mover un nodo de un padre a otro en el árbol basta con actualizar el identificativo del nodo padre, es simple y rápido. Sin embargo, buscar descendientes y ascendientes es más complejo e ineficiente si la base de datos no soporta queries recursivas (que las bases de datos más importantes, Oracle, SQL Server, PosgreSQL salvo MySQL soportan y a partir de la versión 5.6 ya lo hace), también puede requerir una segunda query para buscar los datos de los descendientes y ascendientes, con estas solo recuperamos los identificativos.

Conjuntos anidados (nested sets)

Esta solución se basa en que cada nodo de la jerarquía esté numerado, el padre tendrá dos campos el número de menor hijo y el número del mayor hijo, todos los nodos cuyos números estén entre esos dos números son descendientes del nodo padre. La consulta de buscar los nodos descendientes es simple y eficiente.

Listado 14.21: Obtener descendientes de una categoría

```
select c.id as id
  from categoria as c, categoria as p
  where c.left > p.left and c.rigth < p.rigth
    and p.id = ?
```

Buscar los nodos ascendientes también se puede conseguir una sql eficientemente:

Listado 14.22: Obtener ascendientes de una categoría

```
select c.id as id
  from categoria as c, categoria as p
  where c.left between p.left and p.right
    and p.id = ?
```

La desventaja de esta solución está en el momento que queremos insertar un nuevo nodo en el árbol de la jerarquía o mover un nodo dentro del árbol ya que implica reorganizar los valores de las columnas left y right, puede que de muchas filas y por tanto resultar lento.

Consultas recursivas

Con el soporte de queries recursivas se puede conseguir la simplicidad de las adjacency list y la eficiencia de los conjuntos anidados. El modelo de datos es similar al caso de las listas adyacentes con una columna del identificativo padre del nodo.

Para buscar los descendientes de un nodo sería:

Listado 14.23: Obtener descendientes recursivamente

```
with recursive descendientes as (
  select id as id from categoria c where id = ?
  union all
  select c.id as id from descendientes join categoria c on c.parent = descendientes.id
)
select id from descencientes
```

Para buscar los nodos ascendientes:

Listado 14.24: Obtener ascendientes recursivamente

```
with recursive ascendientes as (
  select id as id from categoria c where id = ?
```

```
  union all
4   select c.id as id from ascendientes join categoria c on ascendientes.id = c.parent
)
select id from ascendientes
```

Como comentaba de las bases de datos más importantes de entre Oracle, SQL Server, PostgreSQL y MySQL solo MySQL no lo soporta aunque a partir de la versión 5.6 también lo hace. Dependiendo de si hacemos más consultas que modificaciones y de si queremos complicarnos más con los conjuntos anidados deberemos optar por una solución u otra, en cualquier caso optaremos por las consultas recursivas si la base de datos lo soporta aunque si usamos un ORM como Hibernate necesitaremos lanzar una consulta nativa en vez de usar HQL.

14.1.9 Multiproyecto con Gradle

Cuando una aplicación o proyecto crece podemos tener necesidad de partir el monolito en varios módulos más pequeños y más manejables, tener varios proyectos o módulos con dependencias entre ellos exige de la herramienta de construcción que esto sea posible y sencillo. Podemos tener un módulo para que contenga la lógica de negocio y persistencia común a otros módulos o aplicaciones e independiente de la capa de presentación de cualquiera de ellas, otro para una librería de componentes de Tapestry comunes y uno o más proyectos web basados en el framework Tapestry que usen tanto el módulo de lógica de negocio y persistencia y la librería de componentes comunes.

En el artículo Ejemplo de multiproyecto con Gradle muestro de forma práctica como usando Gradle podemos dividir un proyecto en varios.

14.1.10 Máquina de estados finita (FSM)

El uso de una máquina de estados permite modelar un proceso con cierta complejidad. Puede ser el ciclo de vida de una compra, un envío, un proceso documental, financiero, ... cualquiera en el que intervengan estados, transiciones entre esos estados y realización de acciones necesarias para proporcionar la funcionalidad deseada. El proyecto Spring Statemachine permite defininir un proceso en el que intenvengan estados. En el artículo Ejemplo de máquina de estados con Spring Statemachine muestro como usarlo.

14.1.11 Configuración de una aplicación en diferentes entornos con Spring Cloud Config

La configuración de una aplicación suele variar según el entorno en el que se ejecuta, la opción recomendada es que este externalizada y que el artefacto que se despliega en cada entorno sea el mismo. Con Spring Cloud Config en vez de guardar la configuración en un archivo de la propia máquina donde se instala podemos guardar de forma centralizada en un repositorio y que la aplicación obtenga la versión más actualizada cuando se inicia.

Desarrollar una aplicación no consiste solo en programar el código que proporciona su funcionalidad, con igual de importancia está como poner en producción esa aplicación para que preste su servicio, algo de lo que el desarrollador no debería ser ajeno. Casi siempre hay algo de configuración que varia entre entornos siendo

estos al menos el de desarrollo y producción. En el ciclo de vida de una aplicación esta pasa por varios entornos de ejecución hasta llegar a producción, desde desarrollo, pruebas, QA y finalmente en producción. Casi seguro que la aplicación en cada uno de estos entornos la configuración varía, por ejemplo, las direcciones ya sean IP o nombres de dominio de las bases de datos relacional u otros servicios externos. Para que en el entorno de pruebas y QA se use exactamente el mismo artefacto (en Java un archivo war o jar) que el que se enviaría al entorno de producción la configuración de la aplicación no debería ser incluida en el propio artefacto, si la configuración fuese incluida en el propio artefacto sería distinto que el que se enviaría a producción y las pruebas no válidas, podría haber alguna diferencia en la construcción del artefacto para cada entorno.

El proyecto Spring Cloud con Spring Cloud Config proporciona un mecanismo para externalizar y actualizar de forma sencilla las varias configuraciones de una aplicación en los diferentes entornos en los que se vaya a ejecutar. La opción recomendada es crear un repositorio de Git donde se almacenarán las diferentes configuraciones de la aplicación para cada entorno y bajo un sistema de control de versiones. El que las configuraciones se obtengan de un repositorio y con Git evita que el archivo de configuración esté como un fichero regular en cada máquina del entorno de ejecución, duplicados si hay varias máquinas o con algunas diferencias en cada una. En caso de tener solo una máquina si deja de funcionar o ser accesible perderíamos el archivo de configuración y los cambios que hubiésemos hecho en él directamente, al mismo tiempo estando en un sistema de control de versiones como Git tendremos un histórico de los cambios realizados.

En el artículo de ejemplo muestro como crear un servidor de configuración para múltiples entornos y una aplicación cliente usando Spring Boot.

14.1.12 Información y métricas con Spring Boot Actuator

Es conveniente tener monitorizado el estado de una aplicación para conocer si el servicio que ofrece está funcionando o en caso de que no conocerlo cuanto antes para restaurarlo además de conocer otra serie de métricas básicas como la cantidad de CPU que se está usando, la cantidad de memoria usada y libre, número de threads instanciados, espacio ocupado y libre en disco, actividad de entrada y salida ya sea de red o de disco, tiempo de inicio del sistema y de la aplicación. Otras métricas a nivel de aplicación que puede interesarnos conocer es número de usuarios conectados, número de sesiones, páginas vistas, sentencias SQL o transacciones ejecutadas, ... que podemos obtener directamente desde la aplicación o combinándolo con otras herramientas como Google Analytics.

Monitorizar el estado de la aplicación nos permitirá conocer en poco tiempo si hay algo que va mal con la intención de restaurar el servicio con el menor tiempo de caída, también nos permitirá conociendo las métricas normales del servicio si hay algún parámetro fuera de los valores típicos como un consumo excesivo de CPU, memoria o disco, conociendo la normalidad podremos descubrir la anormalidad y después corregirla, cuanto antes sea descubierta más sencillo será determinar el cambio que la ha provocado.

En un aplicación que use Spring Boot simplemente incluyendo la dependencia org.springframework.boot:spring-boot-starter-actuator se añadirán a la aplicación varios endpoints para consultar información. Hay varios, estos son solo algunos de la lista completa:

- beans: permite conocer los beans de la aplicación.

- configprops: muestra las propiedades de configuración.
- env: muestra información de la clase ConfigurableEnvironment que incluye propiedades del sistema.
- health: permite conocer si la aplicación está en funcionamiento.
- info: muestra información arbitraria sobre la aplicación.
- metrics: permite obtener los valores de las métricas.
- trace: información de las últimas peticiones a la aplicación.

Spring Boot Actuator ofrece varias opciones de configuración y con herramientas como Prometheus las métricas se pueden monitorizar a lo largo del tiempo. En el artículo Información y métricas de la aplicación con Spring Boot Actuator comento algunos detalles más e incluyo el código necesario para usarlo.

14.1.13 Aplicaciones que tratan con importes

Si vas a desarrollar una aplicación que trata con precios o cantidades de dinero no uses los tipos de datos float ni double ya que no son capaces de representar con exactitud cantidades decimales y muy posiblemente te encuentres con pequeños errores de cálculo de unos céntimos que no cuadrarán exactamente en un desglose de precios. En vez de float y double usa el tipo de datos BigDecimal que si puede representar y hacer cálculos exactos para cantidades decimales.

Además si necesitas formatear los precios y mostrar el símbolo de la moneda puedes hacerlo, consulta el anterior enlace.

14.1.14 Cómo trabajar con importes, ratios y divisas en Java

Aún en Java 8 no tenemos una API incluida en el JDK dedicada al manejo de importes, divisas y conversiones. Si la especificación JSR-354 se incluye en alguna versión podremos hacer uso de ella sin necesidad de ninguna dependencia adicional, pero si tenemos necesidad ahora podemos usar la librería que ha producido la especificación. Usando las clases y métodos de la API evitaremos hacer y mantener una implementación nosotros mismos que además seguro no llega al nivel de esta.

Las aplicaciones de comercio electrónico o que realizan operaciones financieras con importes seguro que necesitan una forma de representar un importe junto con una divisa. Hay que tener en cuenta algunas consideraciones importantes para trabajar con importes. También si necesitan convertir importes en diferentes divisas necesitarán obtener los ratios de conversión de alguna fuente, en el artículo Servicio para obtener ratios de conversión entre divisas comentaba uno que podemos usar, Open Exchange Rates. Java incluye clases para datos numéricos y con ellos se pueden representar importes como por ejemplo BigDecimal. Para importes no debemos usar en ningún caso un tipo de dato float o double ya que estos son incapaces de representar ciertos valores de forma exacta, usando float y double tendremos errores de precisión, redondeo y representación. En vez de crear un nuevo tipo de datos (una clase) que tenga como propiedades un BigDecimal para el importe y un

String o similar para representar la divisa además de implementar las varias operaciones aritméticas y de comparación entre otras muchas cosas que necesitaremos podemos usar la librería que la especificación JSR-354 proporciona una API dedicada a importes y divisas en Java. En Java 8 no se incluyó pero en una futura versión quizá si se incluya en el propio JDK. En esta sección comentaré como usando Java 8 podemos hacer uso de esta API desde ya y que ofrece.

Incluyendo como dependencia la librería generada por la especificación de un proyecto podemos usarla, usando Gradle con:

Listado 14.25: build.gradle

```
...
dependencies {
  compile 'org.javamoney:moneta:1.0'
  ...
}
...
```

La librería hace uso de lambdas, una de las novedades que introdujo Java 8 en el lenguaje, y nos facilita varias funcionalidades. También permite usar streams. Veamos algunas de las posibilidades.

Representación de divisas e importes

Las divisas se representan con CurrencyUnit y los importes se representan usando la clase MoneyAmount, tenemos varias formas de crear instancias de estas clases.

Listado 14.26: Main1.java

```java
// getting CurrencyUnit by currency code and locale
CurrencyUnit euro = Monetary.getCurrency("EUR");
CurrencyUnit dollar = Monetary.getCurrency(Locale.US);

// getting MonetaryAmount by currency code and CurrencyUnit, without using Money (
    implementation class)
MonetaryAmount fiveEuro = Money.of(5, euro);
MonetaryAmount twelveEuro = Money.of(new BigDecimal("12"), euro);
MonetaryAmount tenDollar = Money.of(10, "USD");
MonetaryAmount tenPound = Monetary.getDefaultAmountFactory().setNumber(10).setCurrency("
    GBP").create();

System.out.println("getting MonetaryAmount by currency code and CurrencyUnit, without
    using Money (implementation class)");
System.out.printf("5 EUR: %s\n", fiveEuro);
System.out.printf("12 EUR: %s\n", twelveEuro);
System.out.printf("10 USD: %s\n", tenDollar);
System.out.printf("10 GBP: %s\n", tenPound);

// 5 EUR: EUR 5
```

```
// 12 EUR: EUR 12
19 // 10 USD: USD 10
// 10 GBP: GBP 10
```

La API ofrece varios métodos para extraer los valores numéricos, la parte entera y decimal, que una instancia de MoneyAmount contiene así como obtener los valores en un tipo de datos más básico como BigDecimal.

Listado 14.27: Main2.java

```
1  // getting currency, the numeric amount and precision
   MonetaryAmount amount = Money.of(123.45, euro);

   System.out.printf("123.45 EUR (currency): %s\n", amount.getCurrency());
5  System.out.printf("123.45 EUR (long): %s\n", amount.getNumber().longValue());
   System.out.printf("123.45 EUR (number): %s\n", amount.getNumber());
   System.out.printf("123.45 EUR (fractionNumerator): %s\n", amount.getNumber().
       getAmountFractionNumerator());
   System.out.printf("123.45 EUR (fractionDenominator): %s\n", amount.getNumber().
       getAmountFractionDenominator());
   System.out.printf("123.45 EUR (amount, BigDecimal): %s\n", amount.getNumber().numberValue
       (BigDecimal.class));
10
   // 123.45 EUR (currency): EUR
   // 123.45 EUR (long): 123
   // 123.45 EUR (number): 123.45
   // 123.45 EUR (fractionNumerator): 45
15 // 123.45 EUR (fractionDenominator): 100
   // 123.45 EUR (amount, BigDecimal): 123.45
```

Operaciones aritméticas, de comparación y operaciones personalizadas

Podemos hacer operaciones aritméticas (suma, resta, multiplicación y división) entre dos importes.

Listado 14.28: Main3.java

```
1  // aritmetic
   MonetaryAmount seventeenEuros = fiveEuro.add(twelveEuro);
   MonetaryAmount sevenEuros = twelveEuro.subtract(fiveEuro);
4  MonetaryAmount tenEuro = fiveEuro.multiply(2);
   MonetaryAmount twoPointFiveEuro = fiveEuro.divide(2);

   System.out.printf("5 EUR + 12 EUR: %s\n", seventeenEuros);
   System.out.printf("12 EUR - 5 EUR: %s\n", sevenEuros);
9  System.out.printf("5 EUR * 2: %s\n", tenEuro);
   System.out.printf("5 EUR / 2: %s\n", twoPointFiveEuro);

   // 5 EUR + 12 EUR: EUR 17
   // 12 EUR - 5 EUR: EUR 7
```

```
// 5 EUR * 2: EUR 10
// 5 EUR / 2: EUR 2.5

// negative
MonetaryAmount minusSevenEuro = fiveEuro.subtract(twelveEuro);

System.out.println("negative");
System.out.printf("5 EUR - 12 EUR: %s\n", minusSevenEuro);

// 5 EUR - 12 EUR: EUR -7
```

También podremos hacer comparaciones:

Listado 14.29: Main4.java

```
// comparing
System.out.printf("€7 < €10: %s\n", sevenEuros.isLessThan(tenEuro));
System.out.printf("€7 > €10: %s\n", sevenEuros.isGreaterThan(tenEuro));
System.out.printf("10 > €7: %s\n", tenEuro.isGreaterThan(sevenEuros));

// €7 < €10: true
// €7 > €10: false
// 10 > €7: true
```

Redondear importes

Listado 14.30: Main9.java

```
// rounding
MonetaryAmount euros = Money.of(12.34567, "EUR");
MonetaryAmount roundedEuros = euros.with(Monetary.getDefaultRounding());

System.out.println();
System.out.println("rounding");
System.out.printf("12.34567 EUR redondeados: %s\n", roundedEuros);

// 12.34567 EUR redondeados: EUR 12.35
```

E incluso implementar operaciones más complejas y habituales personalizadas con la clase MonetaryOperator que se puede aplicar usando el método with de MonerayAmount.

Formateado y analizado

Dependiendo de país o la moneda los importes se representan de forma diferente, por ejemplo, en Estados Unidos se usa «,» como separador de millares y «.» como separador de los decimales, en España es diferente, se usa «.» para los millares y «,» para los decimales. También hay monedas que no tienen decimales como el Yen japonés. Disponemos de métodos y clases para formatear correctamente el importe.

Listado 14.31: Main5.java

```
// formating
MonetaryAmountFormat spainFormat = MonetaryFormats.getAmountFormat(new Locale("es", "ES")
    );
MonetaryAmountFormat usFormat = MonetaryFormats.getAmountFormat(new Locale("en", "US"));
MonetaryAmount fiveThousandEuro = Money.of(5000, euro);

System.out.println("formating");
System.out.printf("Formato de 5000 EUR localizado en España: %s\n", spainFormat.format(
    fiveThousandEuro));
System.out.printf("Formato de 5000 EUR localizado en Estados Unidos: %s\n", usFormat.
    format(fiveThousandEuro));

// Formato de 5000 EUR localizado en España: 5.000,00 EUR
// Formato de 5000 EUR localizado en Estados Unidos: EUR5,000.00
```

Podemos hacer la operación contraria parseando o analizando la cadena, obtener un objeto MoneyAmount desde su representación en String.

Listado 14.32: Main6.java

```
// parsing
MonetaryAmount twelvePointFiveEuro = spanishFormat.parse("12,50 EUR");

System.out.printf("Analizando «12,50 EUR» es %s\n", spainFormat.format(
    twelvePointFiveEuro));

// Analizando «12,50 EUR» es 12,50 EUR
```

Ratios de conversión, conversiones entre divisas

Si necesitamos convertir el importe de una moneda a otra necesitaremos el ratio de conversión entre las monedas, es decir, por cada dólar estadounidense cuántos euros son si queremos hacer una conversión de USD a euro. Se puede obtener el ratio de conversión o hacer la conversión directamente entre las dos monedas. En el siguiente código se muestra cuántos euros son 10 USD con la cotización entre las divisas en el momento de escribir el artículo.

Listado 14.33: Main7.java

```
// exchange rates
ExchangeRateProvider exchangeRateProvider = MonetaryConversions.getExchangeRateProvider("
    ECB");
ExchangeRate exchangeRate = exchangeRateProvider.getExchangeRate("USD", "EUR");

System.out.printf("Ratio de conversión de USD a EUR: %f\n", exchangeRate.getFactor().
    doubleValue());

// Ratio de conversión de USD a EUR: 0,921489
```

```
// conversion
CurrencyConversion toEuro = MonetaryConversions.getConversion("EUR");
MonetaryAmount tenDollarToEuro = tenDollar.with(toEuro);

System.out.printf("10 USD son %s EUR\n", tenDollarToEuro);

// 10 USD son EUR 9.214891264283081 EUR
```

La librería incluye varias fuentes para las cotizaciones de cada moneda, una de ellas es el Banco Central Europeo pero también podemos crear la implementación de una nueva fuente que por ejemplo use Open Exchange Rates.

Streams y filtros

Por si todo esto fuera poco podemos usar las características de programación funcional de Java 8 ya que la librería ofrece soporte para streams para ejemplo filtrar o para agrupar.

Listado 14.34: Main8.java

```
// filter
List<MonetaryAmount> onlyDollars = amounts.stream()
  .filter(MonetaryFunctions.isCurrency(dollar))
  .collect(Collectors.toList());

System.out.printf("Solo USD: %s\n", onlyDollars);

List<MonetaryAmount> euroAndDollar = amounts.stream()
  .filter(MonetaryFunctions.isCurrency(euro, dollar))
  .collect(Collectors.toList());

// grouping
Map<CurrencyUnit, List<MonetaryAmount>> groupedByCurrency = amounts.stream()
        .collect(MonetaryFunctions.groupByCurrencyUnit());

System.out.printf("Agrupación por divisa: %s\n", groupedByCurrency);

// Agrupación por divisa: {EUR=[EUR 2], GBP=[GBP 13.37], USD=[USD 7, USD 18, USD 42]}
```

14.1.15 Validar objetos con Spring Validation

Si queremos ser puristas las validaciones deberíamos hacerlas en la base de datos usando restricciones impidiendo de esta manera que se guarden datos inválidos independientemente de la aplicación o microservicio que intente guardar algo en la base de datos. Sin embargo, realizando solo las validaciones en la base de datos puede que perdamos qué campo o campos son erróneos y los motivos por los que son erróneos, información

que seguramente nos interese para indicar detalladamente los datos no válidos al usuario permitiéndole corregirlos.

Con Spring Validation tenemos diferentes formas de realizar las validaciones, dos de ellas son con las anotaciones de la especificación de validación JSR-303 o implementando una clase de la interfaz Validator. Perfectamente podemos usar únicamente los Validator de Spring sin tener en cuenta las anotaciones de javax.validation, nótese también que podemos implementar múltiples validadores de Spring con diferentes criterios de validación aún para las mismas entidades.

En el ejemplo muestro cómo validar registros de jOOQ con Spring Validation.

14.1.16 Java para tareas de «scripting»

Para los scripts normalmente se han utilizado intérpretes como bash por su disponibilidad en cualquier sistema GNU/Linux o si se necesita un lenguaje más avanzado Python, Ruby o Groovy. Cualquiera de estas opciones son empleadas para realizar tareas de scripting que involucran desde generación de informes o archivos, envío de correos electrónicos hasta actualización de una base de datos relacional o nosql, cualquier cosa que queramos automatizar. Al no necesitar compilarse ni generar un artefacto extraño como los archivos .class o .jar de Java basta con escribir el script con cualquier editor de texto y ejecutarlo con el intérprete correspondiente directamente desde el código fuente. El despliegue en un entorno de pruebas o producción es sencillo, basta con subir el código fuente de los scripts a la máquina correspondiente donde se vaya a ejecutar y ejecutarlos, únicamente necesitaremos instalar la versión del intérprete adecuado y las dependencias adicionales del script si necesita alguna como en Python posiblemente en un virtualenv.

Pero al contrario de lo que piensa mucha gente Java puede ser usado perfectamente como lenguaje de scripting con igual simpleza o más que Python, Ruby o Groovy. Java es más verboso sí pero en mi opinión sigue habiendo buenas razones para seguir usándolo entre ellas el compilador, en el artículo Java for everything dan una buena extensa descripción de porque usar Java también para los scripts y donde se expone una forma de usarlo. El compilador de Java validará al menos que el código no tienen errores léxicos o de sintaxis que en un lenguaje interpretado son fáciles de introducir cuando hace meses que no modificas el script olvidando gran parte del código, cuando no has escrito tú el script, cuando hay prisa y presión para hacer las modificaciones por un error en producción apremiante, cuando el tamaño del código empieza a ser considerable, es modificado por varias personas o cuando no se domina el lenguaje profundamente. ¿Qué prefieres, algo más verbosidad (ahora menos con varias de las novedades introducidas en Java 8) o evitar que un script importante en producción se quede a medio ejecutar por un error de compilación y provoque alguna catástrofe? Yo lo tengo claro, que el compilador me salve el culo.

En el artículo Java para tareas de «scripting» comento como con la ayuda de la herramienta de construcción Gradle puede conseguirse la misma facilidad.

14.1.17 DAO genérico

Si usamos un ORM (Object-Relational Mapping) para la capa de persistencia en una base de datos relacional de nuestra aplicación ya sea Hibernate o JPA probablemente después de desarrollar unos cuantos servicios DAO nos daremos cuenta que tenemos unas cuantas operaciones básicas muy similares en todos.

Si queremos evitar tener duplicado ese código y ahorrarnos la codificación de esas operaciones básicas podemos desarrollar un DAO genérico que nos sirva para todas las entidades persistentes de nuestra aplicación usando los generics del lenguaje Java. Las operaciones candidatas a incluir en este DAO son: búsqueda por id, persistencia de un objeto, borrado, actualización, búsqueda paginada de todos los objetos y número de entidades persistidas de un tipo, ...

En la sección section 9.4.2 puede consultarse el código fuente de un DAO genérico usando Hibernate con las transacciones gestionadas por Spring con la anotación Transactional.

Por supuesto, el DAO del ejemplo es muy simple y no nos sirve para todos los casos pero podría ser ampliado para permitir hacer búsquedas además de sobre todos los elementos de una tabla y con paginación con unos determinados criterios de búsqueda que nos cubran las mayoría de los casos que necesitemos, es decir, podríamos implementar métodos de búsqueda que pueden servir para cualquier DAO como:

- findByCriteria(DetachedCriteria criteria, Pagination pagination)
- findByNamedQuery(String namedQuery)
- findByQuery(String query)

14.1.18 Mantenimiento de tablas con un CRUD

Puede que necesitemos hacer en la aplicación el mantenimiento de una serie de tablas con las operaciones básicas como altas, bajas, modificaciones y borrados además de tener un listado paginado para visualizar los datos. Si tenemos varias tablas como estas el desarrollar la funcionalidad aunque sea sencillo será repetitivo. Tapestry no proporciona scaffolding pero si que proporciona una serie de componentes como los componentes Grid para un listado tabluar de elementos yBeanEditor para proporcionar un formualrio de edición de un objeto con los cuales es muy fácil hacer este tipo de funcionalidades. Escribí un artículo sobre como crear un CRUD y otro artículo sobre el componente Grid puedes consultar una entrada de mi blog que escribí acerca de este tema. Una mantenimiento de este tipo solo te supondrá alrededor de 200 lineas de código java, 80 de plantilla tml y aunque use Ajax para la paginación ¡ninguna de código javascript!.

14.1.19 Doble envío (o N-envío) de formularios

Ciertos usuarios por que están acostumbrados a hacer doble clic para hacer algunas acciones, por error o simplemente porque la aplicación tarda en procesar una petición y se cansan de esperar pueden tener oportunidad de realizar un doble clic sobre un botón o enlace, lo que puede desencadenar en un doble envío de una petición al servidor. Esta doble petición puede producir que una de ellas o ambas produzcan un error en el servidor o peor aún una acción indeseada en la aplicación. Para evitar esto podemos hacer que una vez pulsado un botón o enlace o se envíe un formulario el elemento se deshabilite.

Una de las formas en las que podemos implementar esta funcionalidad es mediante un mixin. A continuación pondré el código de uno que sirve tanto para formularios, botones y enlaces tanto si producen una acción Ajax

como no Ajax dando solución al problema desde el lado del cliente mediante javascript. Primero veamos el código Java del mixin, básicamente provoca la inclusión de un módulo javascript al usarse.

```java
package io.github.picodotdev.plugintapestry.mixins;

import org.apache.tapestry5.ClientElement;
import org.apache.tapestry5.ComponentResources;
import org.apache.tapestry5.annotations.InjectContainer;
import org.apache.tapestry5.ioc.annotations.Inject;
import org.apache.tapestry5.json.JSONObject;
import org.apache.tapestry5.services.javascript.JavaScriptSupport;

public class SubmitOne {
  @Inject
  private JavaScriptSupport support;

  @InjectContainer
  private ClientElement element;

  @Inject
  private ComponentResources resources;

  public void afterRender() {
    JSONObject spec = new JSONObject();
    spec.put("elementId", element.getClientId());

    support.require("app/submitOne").invoke("init").with(spec);
  }
}
```

El trabajo importante del mixin está en el módulo javascript con el uso de las funciones de jQuery ajaxStart y ajaxStop y los eventos asociados al elemento submit si se trata de un formulario o click si se trata de cualquier otro tipo de elemento html.

```javascript
define("app/submitOne", ["jquery"], function($) {
  var SubmitOne = function(spec) {
    this.spec = spec;
    this.timeout = null;

    var element = $('#' + this.spec.elementId);

    this.blocked = false;

    var _this = this;
    $(document).ajaxStart(function() {
      _this.onAjaxStart();
```

```
    });
14  $(document).ajaxStop(function() {
      _this.onAjaxStop();
    });
    if (element.is('form')) {
      element.on('submit', function(event) {
19      return _this.onSubmit(event);
      });
    } else {
      element.on('click', function(event) {
        return _this.onSubmit(event);
24    });
    }
  }

  SubmitOne.prototype.onSubmit = function(event) {
29  if (this.isBlocked()) {
      event.preventDefault();
      return false;
    } else {
      this.block();
34    return true;
    }
  }

  SubmitOne.prototype.onAjaxStart = function() {
39  this.block();
  }

  SubmitOne.prototype.onAjaxStop = function() {
    this.unblock();
44 }

  SubmitOne.prototype.isBlocked = function() {
    return this.blocked;
  }
49
  SubmitOne.prototype.block = function() {
    this.blocked = true;
  }

54 SubmitOne.prototype.unblock = function() {
    this.blocked = false;
  }

  function init(spec) {
59  new SubmitOne(spec);
  }
```

```
  return {
    init: init
64  };
});
```

A partir de este momento su uso es tan simple como incluir los siguientes veinte caracteres, «t:mixins="submitOne"», en los componentes que queremos que solo produzcan una nueva petición hasta que la anterior haya terminado. En el siguiente listado para el caso de un formulario, botón y enlace.

```
1  Cuenta: <t:zone t:id="submitOneZone" id="submitOneZone" elementName="span">${cuenta}</t:
       zone>
   <div class="row">
     <div class="col-md-4">
       <h5>Con mixin en form (ajax)</h5>
5      <form t:id="submitOneForm2" t:type="form" t:zone="submitOneZone" t:mixins="submitOne"
       >
         <input t:type="submit" value="Sumar 1"/>
           </form>
     </div>
     <div class="col-md-4">
10     <h5>Con mixin en botón (ajax)</h5>
       <form t:id="submitOneForm3" t:type="form" t:zone="submitOneZone">
         <input t:type="submit" value="Sumar 1" t:mixins="submitOne"/>
       </form>
     </div>
15   <div class="col-md-4">
       <h5>Con mixin en enlace (ajax)</h5>
       <a t:type="eventlink" t:event="sumar1CuentaAjaxSubmitOne" t:zone="submitOneZone" t:
       mixins="submitOne">Sumar 1</a>
     </div>
   </div>
```

14.1.20 Patrón múltiples vistas de un mismo dato

Un proyecto grande contendrá muchos archivos de código fuente, poseer gran cantidad de archivos puede ser una molestia al tener que buscarlos o abrirlos. En el caso de las aplicaciones web puede darse el caso de que un mismo dato tenga un archivo diferente por cada forma de visualizarlo, para reducir el número de archivos en estos casos uso el siguiente patrón.

Puede que necesitemos mostrar un mismo dato de diferentes formas. Una posibilidad es crear una vista por cada forma diferente que se haya de mostrar el dato. Sin embargo, de esta forma tendremos que crear un archivo diferente por cada forma a visualizar, si esto mismo nos ocurre en múltiples datos nos encontraremos en la situación de que el número de archivos del proyecto crecerá suponiendo una pequeña molestia tener que trabajar con tantos, también y peor aún es que múltiples archivos relacionados no lo estarán salvo que

les demos una nomenclatura similar para mantenerlos ordenados por nombre y sean fáciles de encontrar si queremos abrir varios.

Tener tantos archivos puede ser una molestia que denomino de microgestión, esto es, tener muchos archivos pequeñitos. Para evitar microgestionar podemos tener una única vista que con un parámetro determine la forma de representar el dato, mientras que el contenido del archivo tenga alta cohesión me parece adecuado e incluso mejor ya que las diferentes vistas muy posiblemente serán parecidas con lo que quizá dupliquemos algo de código que será mejor tenerlo en un mismo archivo que en varios diferentes.

En Tapestry en una vista se pueden tener múltiples componentes Block cuya misión es agrupar otros componentes que como resultado de procesarse producirán el html. Por otra parte está el componente Delegate que indicándole en el parámetro to un componente Block lo procesa emitiendo el contenido html que generen los componentes que contenga. Teniendo en el código Java asociado al componente que mostrará el dato de diferentes formas un método con cierta lógica que devuelva un componente Block a visualizar podemos conseguir el objetivo.

Listado 14.35: PostComponent.tml

```
<!DOCTYPE html>
<t:container xmlns="http://www.w3.org/1999/xhtml" xmlns:t="http://tapestry.apache.org/
    schema/tapestry_5_4.xsd" xmlns:p="tapestry:parameter">

<t:delegate to="block"/>

<t:block id="excerptBlock">
  <article t:type="any" itemscope="" itemtype="http://schema.org/BlogPosting">
    <header><t:outputraw value="getTag('open')"/><a t:type="any" href="${post.url}"
    target="target" itemprop="sameAs">${post.title}</a><t:outputraw value="getTag('close
    ')"/></header>

    <p class="post-info">
      <span itemprop="dateModified" datetime="${data.get('microdataDate')}">${data.get('
    date')}</span>,
      <span>fuente <a t:type="any" href="${source.pageUrl}" target="target">${source.name
    }</a></span><t:if test="labels">,</t:if>
      <t:if test="labels">
        etiquetas
        <t:loop source="labels" value="label"><a t:type="pagelink" page="label" context="
    labelContext"><span itemprop="articleSection">${label.name}</span></a> </t:loop>
      </t:if>
    </p>

    <p itemprop="description" class="text-justify">
      ${contentExcerpt} [...]
    </p>
    <p>
      <a t:type="any" href="${post.url}" target="target" itemprop="sameAs">Leer artículo
    completo</a>
    </p>
```

```
  </article>
26 </t:block>

<t:block id="fullBlock">
  <article t:type="any" itemscope="" itemtype="http://schema.org/BlogPosting">
    <header><t:outputraw value="getTag('open')"/><a t:type="any" href="${post.url}"
    target="target" itemprop="sameAs">${post.title}</a><t:outputraw value="getTag('close
    ')"/></header>
31
    <p class="post-info" style="font-weight: bold;">
      <span itemprop="dateModified" datetime="${data.get('microdataDate')}">${data.get('
    date')}</span>,
      <span>fuente <a t:type="any" href="${source.pageUrl}" target="target">${source.name
    }</a></span><t:if test="labels">,</t:if>
      <t:if test="labels">
36        etiquetas
        <t:loop source="labels" value="label"><a t:type="any" href="${labelAbsoluteUrl}">
    <span itemprop="articleSection">${label.name}</span></a> </t:loop>
      </t:if>
    </p>

41    <p itemprop="description" class="text-justify">
      <t:outputraw value="content"/>
    </p>
  </article>
</t:block>
46
</t:container>
```

En la clase Java asociada al componente está el método getBlock que determina el bloque a mostrar. En este caso la lógica es muy sencilla, en base a un parámetro que recibe el componente (mode) indicando la vista del dato que se quiere se devuelve el componente Block adecuado. Las referencias a los componentes Block presentes en la vista se puede inyectar usando la anotación @Inject junto con @Component usando el mismo identificativo en la vista y en el nombre de la propiedad para la referencia del componente.

Listado 14.36: PostComponent.java

```
1 package info.blogstack.components;

3 ...

public class PostComponent {

  private DateTimeFormatter DATETIME_FORMATTER = DateTimeFormat.forPattern("EEEE, dd 'de'
     MMMM 'de' yyyy").withLocale(Globals.LOCALE);
8 private DateTimeFormatter MICRODATA_DATETIME_FORMATTER = DateTimeFormat.forPattern("
    yyyy-MM-dd'T'HH:mm");

  enum Mode {
```

```
    HOME, POST, ARCHIVE, NEWSLETTER, DEFAULT
  }
13
  private static int NUMBER_LABELS = 4;

  @Parameter
  @Property
18  private PostRecord post;

  @Parameter(value = "default", defaultPrefix = BindingConstants.LITERAL)
  @Property
  private Mode mode;
23
  @Property
  private LabelRecord label;

  @Inject
28  private MainService service;

  @Inject
  private LinkSource linkSource;

33  @Inject
  private Block excerptBlock;

  @Inject
  private Block fullBlock;
38
  public Object[] getContext() {
    return Utils.getContext(post, post.fetchParent(Keys.POST_SOURCE_ID));
  }

43  public Block getBlock() {
    switch (mode) {
      case HOME:
      case ARCHIVE:
      case POST:
48      case DEFAULT:
        return excerptBlock;
      case NEWSLETTER:
        return fullBlock;
      default:
53        throw new IllegalArgumentException();

    }
  }

58  public String getTag(String key) {
    Map<String, String> m = new HashMap<String, String>();
```

```
        m.put("h1:open", "<h1>");
        m.put("h1:close", "</h1>");
        m.put("h2:open", "<h2>");
63      m.put("h2:close", "</h2>");

        String tag = null;
        switch (mode) {
          case HOME:
68        case ARCHIVE:
          case NEWSLETTER:
          case DEFAULT:
            tag = "h2";
            break;
73        case POST:
            tag = "h1";
            break;
          default:
            throw new IllegalArgumentException();
78
        }

        String k = String.format("%s:%s", tag, key);
        return m.get(k);
83    }

      @Cached(watch = "post")
      public List<LabelRecord> getLabels() {
        return service.getLabelDAO().findByPost(post, NUMBER_LABELS, true);
88    }

      @Cached(watch = "post")
      public String getContentExcerpt() {
        AppPostRecord apost = post.into(AppPostRecord.class);
93      return apost.getContentExcerpt();
      }

      @Cached(watch = "post")
      public String getContent() {
98      AppPostRecord apost = post.into(AppPostRecord.class);
        return apost.getContent();
      }

      @Cached(watch = "post")
103   public Map<String, Object> getData() {
        AppPostRecord apost = post.into(AppPostRecord.class);
        Map<String, Object> datos = new HashMap<>();
        if (apost.getPublishdate() != null) {
          datos.put("date", DATETIME_FORMATTER.print(apost.getPublishdate()));
108       datos.put("microdataDate", MICRODATA_DATETIME_FORMATTER.print(apost.getPublishdate
```

```
    ()));
    }
    if (apost.getUpdatedate() != null) {
      datos.put("date", DATETIME_FORMATTER.print(apost.getUpdatedate()));
      datos.put("microdataDate", MICRODATA_DATETIME_FORMATTER.print(apost.getUpdatedate()
    ));
    }
    return datos;
  }

  public String getTarget() {
    return (mode == Mode.POST) ? null : "_blank";
  }

  public SourceRecord getSource() {
    return post.fetchParent(Keys.POST_SOURCE_ID);
  }

  public Object[] getLabelContext() {
    return Utils.getContext(label);
  }

  public String getLabelAbsoluteUrl() {
    return linkSource.createPageRenderLink("label", true, getLabelContext()).
    toAbsoluteURI();
  }
}
```

14.1.21 Servir recursos estáticos desde un CDN propio u otro como CloudFront

Un Content Delivery Network (CDN) no es más que un servidor, servidores o servicio dedicado a servir el contenido estático o actuar de cache para los clientes. Alguno de los motivos por los que podríamos querer usar un CDN en una aplicación son:

- Algunos servicios CDN están repartidos geográficamente por el mundo de modo que el contenido sea servido de un lugar más cercano al usuario esto hace que el tiempo que tarda en cargar un página o servirse el contenido sea menor.
- Descargar la tarea de servir al menos parte del contenido de la aplicación al CDN hará que no nos tengamos que preocupar de tener la capacidad para servirlo. Cuando se cargar una página se hacen varias peticiones al servidor para obtener el contenido como el html, imágenes, estilos, ... haciendo que los contenidos estáticos sean servidos por el CDN hará que el servidor tenga menos carga, dependiendo del número de usuarios de la aplicación o los picos de tráfico notaremos una mejoría.
- La alta fiabilidad de servicio que ofrecen.

Figura 14.4: Arquitectura no CDN (izquierda), Arquitectura CDN (derecha)

Amazon ClodFront es una de las opciones que podemos usar como CDN aunque perfectamente podemos usar una solución propia.

Para que el contenido estático se sirva del CDN debemos hacer que las URL de las imágenes y hojas de estilo se generen con la URL propia del CDN, al menos, deberemos cambiar el host de esas URL. No hay que hacer mucho más ya que CloudFront creo que se puede configurar para que cuando le lleguen las peticiones del contenido si no las tiene las delegue en la aplicación, una vez que las tiene cacheadas ya no necesita solicitárselas a la aplicación y las sirve él mismo.

Una de las cosas muy interesantes de Tapestry es que podemos modificar prácticamente cualquier comportamiento del mismo, esto es debido a que la mayor parte de sus funcionalidades son ofrecidas mediante servicios que podemos sobrescribir con los que nosotros proporcionemos, el contenedor de dependencias (IoC) lo hace muy fácil. Para modificar las URL de los recursos estáticos que son generados en Tapestry deberemos implementar la clase AssetPathConverter. Una implementación podría ser la siguiente:

```
1  package io.github.picodotdev.plugintapestry.misc;

3  ...

   public class CDNAssetPathConverterImpl implements AssetPathConverter {

     private String protocol;
8    private String host;
     private String port;
     private String path;

     private Map<String, String> resources = CollectionFactory.newMap();
13
```

```
  public CDNAssetPathConverterImpl(@Inject @Symbol(AppModule.CDN_DOMAIN_PROTOCOL) String
    protocol,
      @Inject @Symbol(AppModule.CDN_DOMAIN_HOST) String host,
      @Inject @Symbol(AppModule.CDN_DOMAIN_PORT) String port,
      @Inject @Symbol(AppModule.CDN_DOMAIN_PATH) String path) {
18
    this.protocol = protocol;
    this.host = host;
    this.port = (port == null || port.equals("")) ? "" : ":" + port;
    this.path = (path == null || path.equals("")) ? "" : "/" + path;
23  }

  @Override
  public String convertAssetPath(String assetPath) {
    if (resources.containsKey(assetPath)) {
28    return resources.get(assetPath);
    }
    String result = String.format("%s://%s%s%s%s", protocol, host, port, path, assetPath)
    ;
    resources.put(assetPath, result);
    return result;
33  }

  @Override
  public boolean isInvariant() {
    return true;
38  }
}
```

También deberemos añadir un poco de configuración al módulo de la aplicación para que se use esta nueva implementación. Esto se hace en el método serviceOverride de la clase AppModule.java, donde también en el método contributeApplicationDefaults configuramos los símbolos que se usarán al generar las URLs entre ellos el dominio del CDN.

```
1 package io.github.picodotdev.plugintapestry.services;

...

public class AppModule {
6
  private static final Logger logger = LoggerFactory.getLogger(AppModule.class);

  public static final String CDN_DOMAIN_PROTOCOL = "cdn.protocol";
  public static final String CDN_DOMAIN_HOST = "cdn.host";
11  public static final String CDN_DOMAIN_PORT = "cdn.port";
  public static final String CDN_DOMAIN_PATH = "cdn.path";

  ...
```

```
  public static void contributeServiceOverride(MappedConfiguration<Class, Object>
    configuration, @Local HibernateSessionSource hibernateSessionSource) {
    configuration.add(HibernateSessionSource.class, hibernateSessionSource);
    // Servicio para usar un CDN lazy, pe. con Amazon CloudFront
    configuration.addInstance(AssetPathConverter.class, CDNAssetPathConverterImpl.class);

    if (isServidorJBoss(ContextListener.SERVLET_CONTEXT)) {
      configuration.add(ClasspathURLConverter.class, new WildFlyClasspathURLConverter());
    }
  }

  public static void contributeApplicationDefaults(MappedConfiguration<String, Object>
    configuration) {
    ...

    configuration.add(CDN_DOMAIN_PROTOCOL, "http");
    configuration.add(CDN_DOMAIN_HOST, "s3-eu-west-1.amazonaws.com");
    configuration.add(CDN_DOMAIN_PORT, null);
    configuration.add(CDN_DOMAIN_PATH, "cdn-plugintapestry");
  }

  ...
}
```

Estás serían las URLs antiguas y nuevas con la implementación del AssetPathConverter:

- /PlugInTapestry/assets/meta/zbb0257e4/tapestry5/bootstrap/css/bootstrap.css
- /PlugInTapestry/assets/ctx/8a53c27b/images/tapestry.png
- /PlugInTapestry/assets/meta/z87656c56/tapestry5/require.js
- http://s3-eu-west-1.amazonaws.com/cdn-plugintapestry/PlugInTapestry/assets/meta/z58df451c/tapestry5/bootstra
- http://s3-eu-west-1.amazonaws.com/cdn-plugintapestry/PlugInTapestry/assets/ctx/8a53c27b/images/tapestry.png
- http://s3-eu-west-1.amazonaws.com/cdn-plugintapestry/PlugInTapestry/assets/meta/z87656c56/tapestry5/require

Así de simple podemos cambiar el comportamiento de Tapestry y en este caso emplear un CDN, esta implementación es sencilla y suficiente pero perfectamente podríamos implementarla con cualquier otra necesidad que tuviésemos. El cambio está localizado en una clase, son poco más que 46 líneas de código pero lo mejor es que es transparente para el código del resto de la aplicación, ¿que más se puede pedir?

14.1.22 Ejecución en el servidor de aplicaciones JBoss o WildFly

Los class loaders del servidor de aplicaciones JBoss/WildFly habitualmente han dado algún problema en la ejecución de las aplicaciones y la carga de clases. En versiones antiguas como la 4 de JBoss se podían producir conflictos entre las librerías de las aplicaciones y las librerías instaladas en el servidor ya que en JBoss se

buscaba las clases por defecto y primero en el class loader del servidor en vez de en el classloader de la aplicación (war). Ya en las últimas versiones como JBoss 7 y WildFly la forma de cargar las clases es más parecido al modelo habitual que se sigue en las aplicaciones Java EE y en servidores como Tomcat buscando primero en el directorio classes WEB-INF/classes y entre las librerías de la carpeta WEB-INF/lib del archivo war. Además, con la inclusión de JBoss Modules se puede seguir un esquema OSGi con lo que incluso podríamos usar simultáneamente en el servidor diferentes versiones de la misma librería.

Sin embargo, a pesar de seguir el esquema estándar de buscar las clases y usar OSGi para que Tapestry encuentre los archivos que necesita, como plantillas, imágenes, literales que pueden estar embebidos en los archivos jar de librerías es necesario hacer algunas modificaciones. En una guía de uso de Tapestry con JBoss se explica como conseguir hacer funcionar una aplicación Tapestry tanto en JBoss 7 como en WildFly 8. La solución consiste en proporcionar una clase para que encuentre correctamente los archivos que Tapestry necesita y esta clase será la que veremos en el siguiente ejemplo.

Con la clase que permite funcionar las aplicaciones Tapestry en JBoss/WildFly junto con un poco de configuración para el contenedor de dependencias definido en un módulo será suficiente. La clase es la siguiente:

Listado 14.37: WildFlyClasspathURLConverter.java

```
package io.github.picodotdev.plugintapestry.misc;

...

public class WildFlyClasspathURLConverter implements ClasspathURLConverter {

  private static final Logger logger = LoggerFactory.getLogger(
    WildFlyClasspathURLConverter.class);

  @Override
  public URL convert(final URL url) {
    if (url != null && url.getProtocol().startsWith("vfs")) {
      try {
        final URL realURL;
        final String urlString = url.toString();
        // If the virtual URL involves a JAR file,
        // we have to figure out its physical URL ourselves because
        // in JBoss 7.0.2 the JAR files exploded into the VFS are empty
        // (see https://issues.jboss.org/browse/JBAS-8786).
        // Our workaround is that they are available, unexploded,
        // within the otherwise exploded WAR file.
        if (urlString.contains(".jar")) {
          // An example URL:
          // "vfs:/devel/jboss-as-7.1.1.Final/standalone/deployments/myapp.ear/myapp.war/
    WEB-INF/\
          // lib/tapestry-core-5.3.2.jar/org/apache/tapestry5/corelib/components/"
          // Break the URL into its WAR part, the JAR part,
          // and the Java package part.
          final int warPartEnd = urlString.indexOf(".war") + 4;
          final String warPart = urlString.substring(0, warPartEnd);
```

```
29          final int jarPartEnd = urlString.indexOf(".jar") + 4;
            final String jarPart = urlString.substring(warPartEnd, jarPartEnd);
            final String packagePart = urlString.substring(jarPartEnd);
            // Ask the VFS where the exploded WAR is.
            final URL warURL = new URL(warPart);
34          final URLConnection warConnection = warURL.openConnection();
            final VirtualFile jBossVirtualWarDir = (VirtualFile) warConnection.getContent()
    ;
            final File physicalWarDir = jBossVirtualWarDir.getPhysicalFile();
            final String physicalWarDirStr = physicalWarDir.toURI().toString();
            // Return a "jar:" URL constructed from the parts
39          // eg.
            // "jar:file:/devel/jboss-as-7.1.1.Final/standalone/tmp/vfs/
    deployment40a6ed1db5eabeab/\
            // myapp.war-43e2c3dfa858f4d2/WEB-INF/lib/tapestry-core-5.3.2.jar!/org/apache/
    tapestry5/corelib/components/".
            final String actualJarPath = "jar:" + physicalWarDirStr + jarPart + "!" +
    packagePart;
            return new URL(actualJarPath);
44        } else {
            // Otherwise, ask the VFS what the physical URL is...
            final URLConnection connection = url.openConnection();
            final VirtualFile virtualFile = (VirtualFile) connection.getContent();
            realURL = VFSUtils.getPhysicalURL(virtualFile);
49        }
          return realURL;
        } catch (final Exception e) {
          logger.error("Unable to convert URL", e);
        }
54    }
      return url;
    }
  }
```

La configuración adicional para el contenedor de dependencias es para que Tapestry use esta nueva clase:

Listado 14.38: AppModule.java

```
1  package io.github.picodotdev.plugintapestry.services;

3  ...

   public class AppModule {
     ...
     public static void contributeServiceOverride(MappedConfiguration<Class, Object>
       configuration, @Local HibernateSessionSource hibernateSessionSource) {
8      configuration.add(HibernateSessionSource.class, hibernateSessionSource);

       if (isServidorJBoss(ContextListener.SERVLET_CONTEXT)) {
```

```
      configuration.add(ClasspathURLConverter.class, new WildFlyClasspathURLConverter());

    }
  }
  ...
  private static boolean isServidorJBoss(ServletContext context) {
    String si = context.getServerInfo();

    if (si.contains("WildFly") || si.contains("JBoss")) {
      return true;
    }

    return false;
  }
  ...
}
```

El ContextListener que nos permite acceder al ServletContext es el siguiente:

Listado 14.39: ContextListener.java

```
package io.github.picodotdev.plugintapestry.misc;

...

public class ContextListener implements ServletContextListener {

  public static ServletContext SERVLET_CONTEXT;

  @Override
  public void contextInitialized(ServletContextEvent sce) {
    SERVLET_CONTEXT = sce.getServletContext();
  }

  @Override
  public void contextDestroyed(ServletContextEvent sce) {
  }
}
```

Además hemos de incluir en el proyecto un par de librerías y usar al menos la versión 16 de guava si se incluye como dependencia en el war:

Listado 14.40: build.gradle

```
dependencies {
  ...
  compile 'com.google.guava:guava:16.0.1'
  providedCompile 'org.jboss:jboss-vfs:3.2.1.Final'
  runtime 'org.jboss.logging:jboss-logging:3.1.4.GA'
  ...
```

```
}
```

En la aplicación de ejemplo también deberemos actualizar la versión de guava al menos a la versión 16. Y esta clase y configuración es suficiente para que Tapestry sea compatible con el servidor de aplicaciones JBoss/-WildFly. Si no usamos lo indicado en este artículo al acceder la aplicación fallaría con una excepción.

14.1.23 Aplicación «standalone»

Apache Tapestry es un framework de desarrollo para aplicaciones o páginas web en el que habitualmente se emplea el lenguaje Java y se despliega en un servidor de aplicaciones como entorno de ejecución. Pero Tapestry es una pieza de software que se compone de diferentes partes algunas de las cuales pueden ser utilizadas fuera del contexto de una aplicación web. Este es el caso del contenedor de dependencias que proporciona IoC en Tapestry, podemos usarlo en una aplicación «standalone», es decir, en un programa que se inicia con el típico «public static void main(String[] args)» de las aplicaciones Java.

El contenedor de dependencias de Tapestry tiene algunas propiedades interesantes como que dos servicios pueden ser mutuamente dependientes y que se puede contribuir configuración a cualquier servicio para cambiar en cierta medida su comportamiento además de otras características que explico en el capítulo del Contenedor de dependencias (IoC). Para usarlo en una un programa que se ejecuta de la linea de comandos usando el main de una clase Java primeramente deberemos incluir en el proyecto la dependencia sobre tapestry-ioc, si usamos Gradle de la siguiente manera:

Listado 14.41: build-1.gradle

```
1 compile 'org.apache.tapestry:tapestry-core:5.4'
  compile 'org.apache.tapestry:tapestry-ioc5.4'
```

Una vez que tenemos la dependencia en el programa deberemos iniciar el contenedor IoC e indicarle los diferentes módulos que contendrán la definición de los servicios.

Listado 14.42: Main-1.java

```
1 RegistryBuilder builder = new RegistryBuilder();
  builder.add(TapestryModule.class, HibernateCoreModule.class, HibernateModule.class,
      BeanValidatorModule.class, TapestryOfflineModule.class, GeneratorModule.class);
3 builder.add(new SpringModuleDef("applicationContext.xml"));

  Registry registry = builder.build();
  registry.performRegistryStartup();
```

En este caso he usado Spring para la transaccionalidad e Hibernate para la persistencia. Después de esto tenemos la referencia al registro de servicios, podemos obtener cualquiera en base a la interfaz que implementa, en este caso el servicio que implementa la interfaz MainService.

Listado 14.43: Main-2.java

```
1 registry.getService(MainService.class);
```

Al final de la aplicación deberemos llamar al método shutdown del registro.

Listado 14.44: Main-3.java

```
registry.shutdown();
```

Otra cosa que nos puede interesar es poder generar contenido html usando el sistema de plantillas y componentes de Tapestry, ya sea en una aplicación «standalone» o en una aplicación web para enviar el contenido en un correo electrónico o quizá guardarlo en un archivo. Hay muchos sistemas de plantillas, cada framework suele tener uno propio o usar una solución específica como Thymeleaf pero la mayoría usa un [modelo push en vez de un modelo pull][blogbitix-31], en el caso de Tapestry se emplea el modelo pull que tiene algunas ventajas como explico en el artículo anterior. Si usamos una aplicación Tapestry usándolo también para generar el contenido de los correos o cierto contenido estático evitamos tener que aprender una segunda tecnología además de aprovechar todo el código reutilizable que posiblemente hemos desarrollado en algunos componentes. Para generar el contenido estático que generaría una página en Tapestry tenemos el módulo Tapestry Offline. Como no está en los repositorio de maven debemos descargarnos el jar e incluir la dependencia como un archivo.

Listado 14.45: build-2.gradle

```
compile files('misc/libs/tapestry-offline.jar')
```

Para generar una página de Tapestry fuera de una petición web y de un servidor de aplicaciones debemos usar el servicio OfflineComponentRenderer. Su uso sería el siguiente:

Listado 14.46: GeneratorServiceImpl.java

```
@Override
public File generatePage(String page, Object[] context, Map<String, String> params)
    throws IOException {
  File file = new File(to, getToPage(page, context, params).getPath());
  logger.info("Generating page «{}» ({}, {})...", page, file, params.toString());

  file.getParentFile().mkdirs();

  Writer w = new FileWriter(file);
  render(page, context, params, Globals.LOCALE, w);
  w.close();

  return file;
}

private void render(String page, Object[] context, Map<String, String> params, Locale
    locale, Writer writer) throws IOException {
  TypeCoercer coercer = Globals.registry.getService(TypeCoercer.class);
  OfflineComponentRenderer renderer = Globals.registry.getService("
    BlogStackOfflineComponentRenderer", OfflineComponentRenderer.class);

  EventContext activationContext = new ArrayEventContext(coercer, context);
  PageRenderRequestParameters requestParams = new PageRenderRequestParameters(page,
    activationContext, false);
```

```
  DefaultOfflineRequestContext requestContext = new DefaultOfflineRequestContext();
  for (Map.Entry<String, String> param : params.entrySet()) {
    requestContext.setParameter(param.getKey(), param.getValue());
24  }
  requestContext.setLocale(locale);

  renderer.renderPage(writer, requestContext, requestParams);
}
```

Tengo que decir que al generar la página fuera de una petición web tendremos alguna limitación como solo poder usar assets con el prefijo context. Pero esto por lo menos como explico en el caso de Blog Stack no me ha supuesto ningún problema.

Esto quizá no sea lo habitual pero en Blog Stack ambas posibilidades me han resultado de gran utilidad al desarrollar el proyecto. Las posibilidades son muchas por ejemplo podríamos usar alguna combinación de esto mismo con el microframework Spark si nuestra aplicación estuviese más orientada a una API, aunque también podríamos **??**.

14.2 Funcionalidades de otras librerías

Las funcionalidades que he comentado no son las únicas que puede necesitar una aplicación, hay otras funcionalidades que en determinados casos nos puede hacer falta resolver y que son proporcionadas por otras librerías.

14.2.1 Ansible y Docker

Una de las tareas que deberemos hacer cuando queramos poner la aplicación en producción es desplegarla en el servidor de aplicaciones. Para ello podemos hacer uso de las propias herramientas del servidor de aplicaciones pero también podemos hacer uso de herramientas como Ansible o Docker. Para esta tarea disponemos de varias opciones. En cualquier caso es recomendable que el proceso esté automatizado para evitar posibles errores humanos en algo que puede afectar al servicio que ofrece la aplicación, evitar hacer esta tarea repetitiva manualmente y para que el despliegue nos lleve el menor tiempo posible y de esta manera poder hacer varios despliegues en poco tiempo si nos fuese necesario.

14.2.2 Librerías JavaScript

Las aplicaciones web no solo son código en la parte servidor han de proporcionar soporte para la parte cliente, Tapestry lo entiende así y en el concepto de componente se incluye no solo código JavaScript sino también estilos CSS. Además se proporciona incorporada la librería Require JS que evita la polución JavaScript además carga de forma asíncrona de dependencias de los módulos y jQuery para la manipulación de elementos DOM, estilos y peticiones AJAX.

Con esta base se pueden usar librerías adicionales como Backbone y React con una filosofía similar a Tapestry para construir componentes en la parte cliente de la aplicación que pueden ser probados mediante pruebas unitarias con Jasmine y Sinon. En los siguientes dos artículos se muestra como usar todas estas librerías: Ejemplo lista de tareas con Backbone y React, Pruebas unitarias con Jasmine y Sinon. Por supuesto, podemos elegir usar otras librerías u otras que surjan en el futuro en el rápidamente cambiante ámbito JavaScript como por ejemplo Polymer que también comparte similitudes con Tapestry.

14.2.3 Interfaz REST

Tapestry no es el framework adecuado para desarrollar aplicaciones con únicamente una interfaz REST, para ello los frameworks basados en acciones son más adecuados por tener un mayor control de las URLs que se generan en la aplicación. Sin embargo, se puede usar RESTEasy junto para Tapestry para suplir las carencias en este ámbito. Otras opciones adecuadas pueden ser usar Spring MVC, Spark o Vert.x.

Ofrecer una API REST de una aplicación puede servir para que otras aplicaciones se integren y consuman los servicios de la nuestra. Es una forma de ofrecer los servicios mucho más sencilla y fácil de consumir que usando mensajes SOAP.

14.2.4 Interfaz RPC

En los modelos RPC las llamadas a métodos se hacen a través de la red de forma transparente aunque tendremos que tener en cuenta que se utilizando un medio no fiable y con un rendimiento menor que llamadas en la misma máquina que notaremos más si se usan muchas llamadas. SOAP es una forma de RPC en la que se utiliza XML, algunas críticas a SOAP son que el XML utilizado para la comunicación es complejo y los servicios SOAP no son fácilmente consumibles desde por ejemplo un navegador. Por otra parte, las API REST tratan de solventar algunas de las deficiencias de SOAP como por ejemplo estar expuestas como recursos fácilmente accesibles utilizando los mismos mecanismos de la web y un formato para el intercambio de datos como JSON más sencillo y fácilmente consumible que XML. Sin embargo, algunas críticas que se le están haciendo REST son:

- APIs asíncronas: el modelo RESTful de petición y respuesta no se adapta bien a un modelo donde hay necesidad de enviar datos de forma asíncrona evitando sondear continuamente el servidor con peticiones que consumen recursos de red y de servidor. El modelo asíncrono envía nuevos datos únicamente cuando estos se hacen disponibles.
- Orquestación y experiencia de la API: la granularidad de una API REST no se adapta correctamente a algunas situaciones haciendo necesario realizar varias peticiones HTTP lo que añade carga al cliente, servidor y la red. Orquestando APIs internas en el servidor y publicando una que esté adaptada a lo que necesitan los diferentes clientes supone un mejor rendimiento y simplicidad.
- SDKs vs APIs: los usuarios de las APIs finalmente las consumen desde un lenguaje de alto nivel como JavaScript, Python, Ruby, Java, PHP, C#, etc. con lo que los proveedores de las APIs necesitan ofrecer librerías cliente para algunos de estos lenguajes.

- Protocolos binarios: los formatos binarios son más eficientes que el texto plano, lo que es útil en dispositivos limitados como los utilizados en el internet de las cosas (IoT).
- Alta latencia: la sobrecarga que introduce el protocolo HTTP en cada petición no lo hace adecuado en situaciones en que una baja latencia es necesaria para proporcionar un rendimiento óptimo.

Por otra parte algunos otros puntos a favor de RPC son:

- Se tiene tipado seguro y puede enviar excepciones que puede ser manejadas con la misma infraestructura ofrecida por el lenguaje de programación usado.
- Si se hacen grandes volúmenes de llamadas y datos o hay requerimientos de ancho de banda se pueden usar protocolos de transporte más eficientes que HTTP.

Apache Thrift es una opción que podemos usar para definir una interfaz RPC, gRPC es otra. Perfectamente se puede usar una API RPC para uso interno y sobre ella definir una interfaz REST para consumo público.

Introducción y ejemplo de API RPC con Apache Thrift

14.2.5 Portlets

Los portales ofrecen una solución para los casos de uso de integración de aplicaciones, edición de contenido a modo de CMS, agregación de blogs, foros, colaboración entre personas, red social entre otros. La pieza fundamental de un portal en Java es un portlet. Desarrollar un portlet usando la API directamente no es simple, algunos frameworks que usaríamos para desarrollar aplicaciones y páginas web son usables para desarrollar portlets. Hay un módulo que permite desarrollar portletsmódulo que permite desarrollar portlets y en el siguiente artículo lo muestro usando Apache PlutoApache Pluto.

Portlets con el framework Apache Tapestry y Apache Pluto

14.2.6 Cache

Algunas partes de una página web pueden ser costosas de generar para evitar la carga para el sistema que puede suponer producir ese contenido para cada cliente podemos usar un sistema de cache mediante EhCache o Java Caching System. Perfectamente podemos crear un componente cache como queda demostrado en anterior enlace.

14.2.7 Plantillas

Las aplicaciones puede necesitar enviar correos electrónicos, Tapestry no permite generar el contenido los mensajes ya sea en texto plano o html usando su propio sistema de plantillas por lo que deberemos usar alguno de los muchos de los disponibles para Java. Uno de ellos es Thymeleaf, otro Mustache que además de poder usarlo en Java podemos usarlo como motor de plantillas en el navegador del cliente de forma que podemos reducir el número de herramientas que necesitamos conocer.

14.2.8 Informes

Es habitual que las aplicaciones generen algún tipo de informe o genere algún documento como salida. Una solución puede ser usar JasperReports que permiten generar informes con una calidad alta. Aunque no es una herramienta sencilla el esfuerzo de aprender a usarla merece la pena.

14.2.9 Gráficas

En los informes, correos electrónicos o en la propia aplicación web puede que necesitemos generar una representación gráfica de los datos. JFreeChart nos permite generar muchos tipos diferentes de gráficas desde el lenguaje Java.

14.2.10 Análisis estático de código

El compilador nos avisa de los errores en el código pero no analiza como está escrito. Si queremos que nuestro código no baje en calidad a medida que desarrollamos comprobando de forma automatizada las convenciones acordadas por el equipo u organización o imports innecesarios, variables asignadas y no usadas, posibles fallos, etc... podemos usar herramientas como PMD, checkstyle y CodeNarc.

14.2.11 Facebook y Twitter

Para integrarnos con Facebook una de las mejores librerías disponibles es RestFB, tiene pocas dependencias y es sencilla de utilizar. Para integrarnos con Twitter podemos usar la librería Twitter4j.

14.2.12 Fechas

La API para el tratamiento de fechas ofrecida en el JDK es criticada por mucha gente. Una de las razones es que no es fácil de usar, otra es que dificulta las pruebas unitarias. Por estas razones es habitual utilizar la librería JodaTime. Si te es posible utilizar esta librería probablemente sea mejor opción que usar la ofrecida en el actual JDK, en la versión de Java 8 se espera que se ofrezca una nueva API para las fechas que resuelva los problemas anteriores y quizá en ese momento JodaTime deje de considerarse tan a menudo como una buena y necesaria opción.

Capítulo 15

Notas finales

15.1 Comentarios y feedback

Animo a que me escribáis y estaré encantado de recibir vuestros comentarios aunque solo sea para decirme «gracias», «acabado de empezar a leer el libro», «me ha gustado» o críticas constructivas como «en el capítulo sobre X explicaría Y», «no me ha quedado claro la sección X, ¿como se hace Y?», «he encontrado un errata en la página X en la palabra Y» ... o cualquier idea o sugerencia que se os ocurra. Es una de las formas como me podéis «pagar» por el libro si queréis hacerlo y por el momento es suficiente recompensa para mí.

Las sugerencias prometo leerlas todas y tenerlas en cuenta en futuras actualizaciones. También si necesitas ayuda estaré encantado de proporcionarla a cualquier cosa que me preguntéis si después de buscar en Google, la documentación de Tapestry y los foros no encontráis respuesta, aunque tener en cuenta que mi tiempo es limitado y esto lo hago en mi tiempo libre por lo que puede que tarde en contestar, aún así intentaré dar siempre al menos alguna indicación.

También estoy dispuesto de escuchar alguna proposición (decente) que queráis hacerme. Para ello mi dirección de correo electrónico es:

pico.dev@gmail.com

15.2 Más documentación

Este libro espero que contenga todo lo necesario para empezar a usar el framework (y si le falta algo puedes comentármelo) y cubre los aspectos más comunes de todas las aplicaciones web pero hay partes que intencionadamente he dejado fuera. Para continuar aprendiendo sobre este framework lee la amplia sección de documentación del proyecto donde podrás encontrar:

- La API en formato Javadoc del proyecto, muy útil para conocer los servicios del framework.
- La referencia de los componentes, donde puedes ver los parámetros, tipos y descripción de cada uno de ellos así como un pequeño ejemplo de uso.
- Una guía de usuario que también puede ayudarte a empezar con el framework.
- Varios blogs del propio Howard Lewis Ship y otros commiters del proyecto donde suelen escribir información interesante.
- Otros libros escritos.
- Artículos, presentaciones en vídeo y wikis.
- Un sitio muy recomendable donde encontrar ejemplos es la aplicación JumpStart. Si necesitas ver un ejemplo completo y funcionando sobre algún aspecto de Tapestry muy probablemente lo encuentres aquí.
- También en mi blog podrás encontrar más entradas sobre este framework y una entrada específica en la que voy recogiendo la documentación que encuentro en internet.
- También en mi reposotorio de GitHub donde puedes encontrar el código fuente completo de los ejemplos que he escrito hasta el momento.

Los ejemplos pueden descargarse y probarse con los siguientes comandos en la terminal:

- Windows

```
1 git clone git://github.com/picodotdev/blog-ejemplos.git
2 # Si no se dispone de git, descargar el zip con el repositorio completo y
      descomprimir
  cd blog-ejemplos/PlugInTapestry/
  ./gradlew.cmd run
  # Abrir en el navegador con la URL indicada al final de la terminal
```

- Linux / Mac

```
1 git clone git://github.com/picodotdev/blog-ejemplos.git
  # Si no se dispone de git, descargar el zip con el repositorio completo y
      descomprimir
  cd blog-ejemplos/PlugInTapestry/
  ./gradlew run
5 # Abrir en el navegador con la URL indicada al final de la terminal
```

Terminal

```
 ______                  __             ____
/_  __/__ ____  ___ ___ / /_______ __  / __/
 / / / _ `/ _ \/ -_|_-</ __/ __/ // / /__ \
/_/  \_,_/ .__/\__/___/\__/_/  \_, / /____/
       /_/                    /___/  5.4.4 (development mode)


2019-04-19 21:02:01,890  INFO                                           org.hibernate.Version HHH000412: Hibernate Core {
5.1.16.Final}
2019-04-19 21:02:01,892  INFO                                   org.hibernate.cfg.Environment HHH000206: hibernate.proper
ties not found
2019-04-19 21:02:01,893  INFO                                   org.hibernate.cfg.Environment HHH000021: Bytecode provide
r name : javassist
2019-04-19 21:02:01,920  INFO                        org.hibernate.annotations.common.Version HCANN000001: Hibernate Comm
ons Annotations {5.0.1.Final}
2019-04-19 21:02:02,359  INFO                                   org.hibernate.dialect.Dialect HHH000400: Using dialect: o
rg.hibernate.dialect.H2Dialect
2019-04-19 21:02:03,229  INFO  springframework.scheduling.concurrent.ThreadPoolTaskExecutor Initializing ExecutorServic
e 'applicationTaskExecutor'
2019-04-19 21:02:03,483  INFO  ingframework.boot.actuate.endpoint.web.EndpointLinksResolver Exposing 2 endpoint(s) bene
ath base path '/actuator'
2019-04-19 21:02:03,600  INFO                     org.apache.coyote.http11.Http11NioProtocol Starting ProtocolHandler ["
https-jsse-nio-8443"]
2019-04-19 21:02:03,733  INFO                     org.apache.coyote.http11.Http11NioProtocol Starting ProtocolHandler ["
http-nio-8080"]
2019-04-19 21:02:03,755  INFO  org.springframework.boot.web.embedded.tomcat.TomcatWebServer Tomcat started on port(s):
8443 (https) 8080 (http) with context path ''
2019-04-19 21:02:03,758  INFO                      io.github.picodotdev.plugintapestry.Main Started Main in 4.378 secon
ds (JVM running for 4.981)
2019-04-19 21:02:03,767  INFO                      io.github.picodotdev.plugintapestry.Main
   ___  __         ____   ______              __
  / _ \/ /_ _____ _/  _/__/_  __/__ ____  ___ ___ / /_______ __
 / ___/ / // / _ `// // _ \/ / / _ `/ _ \/ -_|_-</ __/ __/ // /
/_/  /_/\_,_/\_, /___/_//_/_/  \_,_/ .__/\__/___/\__/_/  \_, /
            /___/                /_/                    /___/
2019-04-19 21:02:03,767  INFO                      io.github.picodotdev.plugintapestry.Main Application running
```

15.3 Ayuda

Si al usarlo tienes alguna duda y este libro no te ayuda a resolverla puedes consultar la documentación oficial del proyecto. Si aún así sigues sin resolverla puedes buscar en alguno de los archivos que contienen todos los correos enviados por otros usuarios, Apache Archive o nabble (en inglés), y finalmente después de haber leído como hacer preguntas de la forma correta preguntar en esas listas a otros usuario donde muy probablemente te den alguna solución o indicación.

También si quieres ver ejemplos prácticos sobre un tema relacionado con Tapestry puede acceder a la aplicación Jumstart, contiene muchos ejemplos en los que podrás ver el código fuente y probarlos funcionando.

15.4 Sobre el autor

Estoy titulado como Ingeniero en Informática y he trabajado en varias consultorías informáticas en el ámbito de Euskadi durante más de 10 años con las funciones de programador adquiriendo un conocimiento alto sobre muchas de las más actuales tecnologías de la plataforma Java y del desarrollo de aplicaciones web (aunque sigo sin considerarme un experto). En el presente estoy trabajando en una empresa dedicada al comercio electrónico usando el framework Grails lo que me ha permitido conocer otra opción además de las que ya tenía experiencia (JSP/Servlet, Struts, JSF y Tapestry).

Durante mi vida laboral dedicados al desarrollo de aplicaciones y páginas web he tenido varias oportunidades donde he podido usar Apache Tapestry y en todas he quedado ampliamente satisfecho con el resultado y los

buenos momentos que pasé mientras desarrollaba con este framework. La primera vez en la que lo usé en un proyecto real fue con la versión 3 en una aplicación de nombre PlugInRed GestorMensajes con la finalidad de administrar mensajes SMS para una conocida empresa de telecomunicaciones multinacional, a partir del nombre de la cual en su honor me he basado para crear el título de este libro, la siguiente fue en una aplicación de inventario de aplicaciones con Tapestry 4 para otra empresa dedicada a la distribución de alimentos.

15.5 Lista de cambios

Versión 1.0 / 2013-07-24

- Publicación inicial

Versión 1.0.1 / 2013-07-27

- Internacionalización en entidades de dominio
- Seguridad CSRF
- Mejor explicación de los mixins
- Versiones específicas de libros electrónicos (epub y mobi)

Versión 1.1 / 2013-11-24

- Añadida lista de cambios
- Solución al doble envío de peticiones (añadido código ejemplo)
- Ampliada sección convenciones para archivos properties de localización
- Explicado que hay un huevo de pascua con premio
- Solución al problema de seguridad CSRF (añadido código ejemplo)
- Revisar en que consiste CSRF y diferencia con XSS
- Añadida sección de posibles opciones en relaciones jerárquicas en bases de datos relacionales
- Migración de JPA a Hibernate. Ahora el código usa Hibernate en vez de JPA.
- Transacciones con anotación CommitAfter
- Transacciones con anotación propia Transactional
- Transacciones con Spring
- Integración con Spring
- Nueva portada

Versión 1.2 / 2014-01-17

- Portada con el mismo estilo que la presentación
- Varias correcciones en textos
- Añadida presentación

Versión 1.3 / 2014-08-29

- Ejecución en el servidor de aplicaciones JBoss o WildFly
- Página Dashboard
- Reescrito el inicio rápido
- Modelo «push» contra modelo «pull» en frameworks web
- Añadidos números de linea en los listados de código
- Reescrita sección Plantillas
- Servir recursos estáticos desde un CDN propio u otro como CloudFront
- Anotación Cached
- Reescrita sección Convenciones para archivos properties l10n
- Usar Tapestry de forma «standalone»
- Revisar principios de http://tapestry.apache.org/principles.html
- Revisado eventos de componente http://tapestry.apache.org/component-events.html
- Anotación Secure y seguridad con HTTPS
- Revisar capturas de pantalla (inicio rápido)
- Formato en HTML
- Reescrita sección Integración con Spring

Versión 1.4 / 2016-01-06

- Mejorados los listados de código permitiendo copiar y pegar no incluyendo los números de línea y sin espacios entre letras, aunque todavía no copia bien el formateo de los tabuladores.
- Añadida sección persistencia con jOOQ.
- Añadida sección máquina de estados finita (FSM) con Java 8.
- Añadida sección con ejemplo de multiproyecto con Gradle.
- Añadida sección con ejemplo de configuración de una aplicación en diferentes entornos con Spring Cloud Config.

- Añadida sección validar objetos con Spring Validation.
- Añadida sección guardar contraseñas usando Salted Password Hashing y otras formas correctas.
- Añadida sección productividad y errores de compilación.
- Añadida sección cómo trabajar con importes, ratios y divisas en Java.
- Añadida sección datos de sesión externalizados con Spring Session.
- Añadida sección librerías JavaScript.
- Añadida sección interfaz REST.
- Añadida sección patrón múltiples vistas de un mismo dato.
- Añadido en el ejemplo Spring Boot Actuator.
- Añadida sección Java para tareas de «scripting».
- Añadida sección interfaz RPC.
- Revisada sección forma de ejecución de una aplicación con Spring Boot, generando un archivo war o con servidor externo.
- Añadida sección información y métricas con Spring Boot Actuator.
- Añadida opción adicional para internacionalizar entidades de dominio.
- Añadidas opciones diferentes o alternativas a Tapestry en la plataforma Java.
- Actualizadas muchas imágenes.
- Actualización de la licencia a CC 4.0 y permitiendo uso comercial.
- Eliminada sección Anotación Transactional (la recomendación es usar Spring).
- Revisado diagramas modelos push y pull.
- Ampliada sección internacionalización con múltiples formas plurales e internacionalización en JavaScript.
- Modificado el ejemplo usando Spring Boot y actualización capítulos y código.
- Ampliada sección página de informe de error según excepción producida.

Versión 1.4.1 / 2016-02-14

- Varias correcciones de sintaxis.
- Cambios pequeños en algunas secciones.
- Revisada sección espacio de nombres p:parameter.
- Ampliada sección expansiones.

Versión 1.4.2 / 2019-04-21

- Añadida sección lanzar eventos desde JavaScript.
- Añadida sección casos de éxito.
- Añadida sección con webjars.
- Añadida sección actualizar versiones de assets incorporados.
- Añadida sección algunas diferencias con Servlets/JSP y Grails.
- Reescrita sección Internacionalización (i18n) en entidades de dominio.
- Revisada sección Mantenimiento de tablas con un CRUD.
- Capturas de imagen actualizadas.
- Varias correcciones en el código del ejemplo.

15.6 Sobre el libro

Este libro ha sido realizado completamente con software libre incluyendo LyX para componerlo, LibreOffice para la portada, GIMP e Inkscape para las imágenes, OpenJDK, eclipse y Tomcat para los ejemplos entre algunas otras herramientas y Arch Linux como sistema operativo.

15.7 Licencia

Este obra está bajo una licencia de Creative Commons Reconocimiento-CompartirIgual 4.0 Internacional.Para ver una copia de esta licencia, visita http://creativecommons.org/licenses/by-sa/4.0/.