ESTUDIO DE LOS SERVIDORES WEB OSGI

FAYDER ALFONSO FLÓREZ HERRERA
T00014525

Trabajo de Investigación Dirigida presentado al Programa de Ingeniería de Sistemas como requisito parcial para optar al título de
INGENIERO DE SISTEMAS

Giovanny Rafael Vasquez Mendoza
ASESOR

UNIVERSIDAD TECNÓLOGICA DE BOLÍVAR
FACULTAD DE INGENIERÍA
PROGRAMA DE INGENIERÍA DE SISTEMAS
CARTAGENA DE INDIAS
COLOMBIA
2011

AGRADECIMIENTOS

En primer lugar quisiera agradecer a mi asesor, Giovanny Rafael Vásquez Mendoza, por su laboriosa guía a través de todo este proceso; sin su ayuda, conocimiento y sugerencias el desarrollo de este Trabajo de Grado no hubiese sido el mismo. De la misma forma, agradezco a mis evaluadores: William Caicedo Torres y Edwin Puerta del Castillo; por permitirme y ayudarme sustentar este Trabajo de Grado desde la distancia.

Agradezco también a la empresa ZERO TECHNOLOGIES OÜ, radicada en Tallinn - Estonia, en la cual laboro hasta la fecha, por brindarme el espacio y las herramientas adecuadas para obtener el conocimiento necesario para el desarrollo de este Trabajo de Grado.

Finalmente, de todo corazón agradezco a mi familia por su interminable apoyo en las iniciativas y metas personales que he emprendido, incluyendo la realización de este Trabajo de Grado, por su comprensión y amor.

CONTENIDO

Página

ILUSTRACIONES 6

ABREVIATURAS 4

1. INTRODUCCIÓN 1

1.1 El Problema 3
1.1.1 Control de Visibilidad 4
1.1.2 El Classpath de Java 4
1.1.3 Implementación y Administración 5
1.1.4 Servidores Web para aplicaciones Java 5
1.2 OSGi como solución 6
1.2.1 La Alianza OSGi 7
1.3 El Framework OSGi 10
1.3.1 Capa de Seguridad 11
1.3.2 Capa Modular 13
1.3.3 Capa de Ciclo de Vida 23
1.3.4 Capa de Servicios 27

1.4 Implementaciones de la Especificación OSGi 32

2. CASOS DE ESTUDIO 34

2.1 Ambiente y Herramientas 34
2.2 Ventajas de la Especificación OSGi 37
2.3 Conociendo el Framework Apache Felix 38
2.4 'Hola Mundo' OSGi 41
2.4.1 Ciclo de Vida de un bundle 46
2.4.2 Apache Felix Web Console 48
2.4.3 'Hola Mundo OSGi' Web 51
2.4.4 'Hola Mundo OSGi' como Servicio 55
2.5 OSGi + MAVEN: El bundle plug-in para MAVEN . . . 65
2.6 OSGi + SPRING: SPRING-DM 75
2.7 OSGi + Tapestry: El Plug-in Tapestry OSGi 93
2.7 Ventajas de la Especificación OSGi 103
2.7.1 Modularidad 103
2.7.2 Control de Versiones y Actualización Dinámica . . 104
2.7.3 Extensibilidad y Adaptación 110

2.7.4 Transparencia y Laziness 114

3. CONCLUSIONES 117

FUENTES 121

GLOSARIO 122

ANEXO 1: Breve descripción de Casos de Estudio
ANEXO 2: Configuración de Herramientas en Linux
ANEXO 3: Ejemplos desarrollados en Apache Felix
ANEXO 4: Código fuente

ILUSTRACIONES

Ilustración Página

1. Ilustración 1: Capas del Framework OSGi 11
2. Ilustración 2: Capa Modular de la Especificación OSGi . . . 14
3. Ilustración 3: Red de Cargadores de Clases 20
4. Ilustración 4: Espacio de clases del Bundle A 21
5. Ilustración 5: Comando felix:lb 39
6. Ilustración 6: Archivo MANIFEST.MF de 'hola' 41
7. Ilustración 7: Clase utb.osgi.hola.HolaMundoOsgi 43
8. Ilustración 8: Creación de hola-1.0.jar 44
9. Ilustración 9: Maven plug-in maven-jar-plugin 45
10. Ilustración 10: Instalación de ejemplo 'Hola Mundo OSGi' . . . 45
11. Ilustración 11: Ejemplo 'Hola Mundo OSGi' en acción. . . . 46
12. Ilustración 12: Ciclo de Vida de un bundle 48
13. Ilustración 13: Instalación de Apache Felix HTTP Service . . . 49
14. *Ilustración 14: Apache Felix Web Console* 50
15. Ilustración 15: Cabeceras de Apache Felix Web Console . . . 51
16. Ilustración 16: Archivo MANIFEST.MF de 'holaWeb' . . . 52
17. Ilustración 17: Clase HolaMundoOsgiServlet de 'holaWeb' . . . 53
18. Ilustración 18: Clase HttpServiceTracker de 'holaWeb' . . . 54
19. Ilustración 19: Clase HolaMundoOsgiWeb de 'holaWeb' . . . 55
20. Ilustración 20: Creación de holaWeb-1.0.jar 55
21. Ilustración 21: Instalación de holaWeb-1.0.jar 56
22. Ilustración 22: Interfaz HolaMundoOsgiServicio 57
23. Ilustración 23: Clase HolaMundoOsgiServicioImpl de 'holaServicio' . . 57
24. Ilustración 24: Clase Activator de 'holaServicio' 58
25. Ilustración 25: Archivo MANIFEST.MF de 'holaServicio' . . . 59
26. *Ilustración 26: Bundle 'holaServicio' en Apache Felix Web Console* . . 60
27. Ilustración 27: Instalación de 'holaServicio' en repositorio maven local . . 61
28. Ilustración 28: Clase HolaMundoOsgiServlet de 'holaConsumidor' . . 62
29. Ilustración 29: Clase HttpServiceTracker de 'holaConsumidor'. . . . 63
30. Ilustración 30: Clase Activator de 'holaConsumidor' 64
31. Ilustración 31: Archivo MANIFEST.MF de 'holaConsumidor'. . . . 64
32. Ilustración 32: Archivo POM del proyecto 'hola' 67
33. Ilustración 33: Ejemplo de bundle plug-in para Maven . . . 69
34. Ilustración 34: Archivo POM del proyecto 'holaServicioMaven' . . 72
35. *Ilustración 35: Bundle 'holaServicioMaven' en Apache Felix Web Console* . 73
36. *Ilustración 36: Bundle plug-in para Maven de 'holaConsumidorMaven'* . . 74
37. Ilustración 37: Bundle 'holaConsumidorMaven' en Apache Felix Web Console . 75
38. Ilustración 38: Spring-DM bundles en Apache Felix Web Console . . 80
39. Ilustración 39: Diagrama de clases Caso de Estudio OSGi + Spring-DM. 80
40. Ilustración 40: Clase Person de 'spring-person-model' . . . 82
41. Ilustración 41: Clase PersonDAO de 'spring-person-model' . . . 83
42. Ilustración 42: Clase PersonDAOImpl de 'spring-person-model' . . 83
43. Ilustración 43: Clase PersonDataSource de 'spring-person-model' . . 84
44. Ilustración 44: Archivo applicationContext.xml de 'spring-person-model' . 85
45. Ilustración 45: Archivo osgiContext.xml de 'spring-person-model' . . 85
46. Ilustración 46: Bundle Plug-in para Maven de 'spring-person-model' . . 86
47. Ilustración 47: Archivo osgiContext.xml de 'spring-person-view' . . 87

48. Ilustración 48: Clase PersonConsole de ‘spring-person-view’ . . . 88
49. Ilustración 49: Archivo applicationContext.xml de ‘spring-person-view’ . 89
50. Ilustración 50: Clase Activator de ‘spring-person-view’ . . . 89
51. Ilustración 51: Cabeceras MANIFEST de ‘spring-person-view’ . . 90
52. Ilustración 52: Bundle ‘spring-person-model’ en Apache Felix Web Console . 91
53. Ilustración 53: Bundle ‘spring-person-view’ en Apache Felix Web Console . 91
54. Ilustración 54: Bundle ‘spring-person-view’ en Apache Felix . . . 92
55. Ilustración 55: Clase PersonDAOFactory de ‘tapestry-person-model’ . . 97
56. Ilustración 56: Clase PersonDAOImpl de ‘tapestry-person-model’ . . 97
57. Ilustración 57: Versión 1 de archivo Index.tml de ‘tapestry-person-view’ . 98
58. Ilustración 58: Versión 1 de clase Index de ‘tapestry-person-view’ . . 99
59. Ilustración 59: Plantilla ShowAll.tml de ‘tapestry-person-view’ . . 99
60. Ilustración 60: Clase ShowAll.java de ‘tapestry-person-model’. . . . 100
61. Ilustración 61: Clase ComponentModule de ‘tapestry-person-view’ . . 101
62. Ilustración 62: Archivo component.xml de ‘tapestry-person-view’ . . 101
63. Ilustración 63: Página Index de Ejemplo OSGi + Tapestry . . . 102
64. Ilustración 64: Página ShowAll de Ejemplo OSGi + Tapestry . . . 102
65. Ilustración 65: Paquetes exportados de ‘tapestry-person-model’ versión 1.0.0 . 106
66. Ilustración 66: Bundle ‘tapestry-person-model’ con versiones 1.0.0 en Apache Felix Web Console 106
67. Ilustración 67: Bundle ‘tapestry-person-view’ importa versiones 1.0.0 de los paquetes exportados por el bundle ‘tapestry-person-model’ 106
68. Ilustración 68: Constructor de la Clase PersonDataSource del bundle ‘tapestry-person-model’ versión 2.0 107
69. Ilustración 69: Paquetes exportados de ‘tapestry-person-model’ versión 2.0.0. 107
70. Ilustración 70: Versión de dependencia ‘tapestry-person-model’ del bundle ‘tapestry-person-view’ actualizada a 2.0 108
71. Ilustración 71: Bundle ‘tapestry-person-view’ importa los paquetes exportados por ‘tapestry-person-model’ versión 2.0 108
72. Ilustración 72: Página ShowAll de ‘tapestry-person-view’ actualizada con ‘tapestry-person-model’ versión 2.0 109
73. Ilustración 73: Archivo Index.tml de ‘tapestry-person-view’ con cambios para actualización dinámica 110
74. Ilustración 74: Clase Index de ‘tapestry-person-view’ con cambios para actualización dinámica 110
75. Ilustración 75: Página Index de ‘tapestry-person-view’ después de actualización dinámica. 110
76. Ilustración 76: Servicio Log Service de Apache Felix Web Console . . 112
77. Ilustración 77: Instalando Apache Felix Log Service para Apache Felix Web Console 112
78. Ilustración 78: Log Service instalado para Apache Felix Web Console . . 113
79. Ilustración 79: Detalle del bundle ‘tapestry-person-view’ en Apache Felix Web Console 115

ABREVIATURAS

API *Application Programming Interface* (Interfaz de Programación de Aplicaciones)

HTTP *Hypertext Transport Protocol* (Protocolo de Transferencia de Hipertexto)

IDE *Integrated Development Environment* (Entorno de Desarrollo Integrado)

JAR *Java Archive*

JRE *Java Runtime Environment* (Máquina Virtual de Java)

JVM *Java Virtual Machine* (Máquina Virtual de Java)

MVC *Model-View-Controller* (Modelo-Vista-Controlador)

POM *Project Object Model*

POJO *Plain Old Java Object*

POO Programación Orientada a Objetos

URI *Uniform Resource Identifier*

URL *Uniform Resource Locator*

WAR *Web Application Archive*

SINTAXIS

En algunos partes durante este Trabajo de Grado, se utilizará una sintaxis especial para describir aspectos de la especificación OSGi. Esta sintaxis está basada en los siguientes símbolos:

*	Repetición del elemento anterior, cero o más veces, e.g. `(',' elemento ) *`
+	Repetición, una o más veces
?	Elemento previo es opcional
(…)	Agrupamiento
'…'	Literal
|	Disyunción (Or)
[…]	Conjunto (por lo menos uno)
..	Lista, e.g. 1..5, es la lista 1 2 3 4 5

dígito	::= [0..9]
alfa	::= [a..zA..Z]
alfa-numérico	::= alfa | dígito
token	::= (alfa-númerico | '_' | '-') +
número	::= dígito+
parámetro	::= directiva | atributo
directiva	::= token ':=' argumento
atributo	::= token '=' argumento

CAPÍTULO
1. INTRODUCCIÓN

Un módulo, en el área del desarrollo de software, se puede definir como una parte auto-suficiente de un sistema mucho mayor, y las características de un buen módulo están dadas por tener alta cohesión, es decir que todos los elementos del módulo tienen la tarea de solucionar o brindar una funcionalidad en común, y un bajo acoplamiento, es decir que los elementos del módulo dependen poco o nada de otros elementos que existan en un módulo diferente.

Actualmente la modularidad en Java está definida por la noción de "paquetes", los cuales son agrupaciones de clases. Estas clases a su vez, al igual que los métodos en ellas, tienen modificadores que controlan su visibilidad (`public, protected, private`). Para que cierta parte del código sea visible de un paquete a otro, este debe ser declarado `public,` lo que hace que dicho código sea accesible a cualquier otro paquete, lo cual conlleva a exponer detalles de la implementación.

Incluso si el diseño de una aplicación hace que esta esté idealmente organizada, es decir, que todas las clases e interfaces están idealmente organizadas en paquetes sin que exista la falla expuesta en el párrafo anterior, que existan capas funcionales en la aplicación, que se utilice una herramienta para la Inyección de Dependencias tal como Spring, o incluso que el proyecto como tal esté dividido en diferentes proyectos de compilación; al final toda la aplicación tiene que ser implementada en un único archivo WAR, y esto no es realmente modularidad.

La especificación OSGi resuelve el problema de modularidad de la plataforma Java, y además presenta ciertas ventajas con respecto a los servidores Web para aplicaciones Java no-OSGi comúnmente utilizados en la actualidad, lo cual se demostrará en esta tesis.

Este trabajo de investigación tiene como objetivo principal demostrar mediante casos de

estudio las ventajas que tienen las implementaciones de la especificación OSGi sobre los actuales servidores Web no OSGi.

Como objetivos específicos de este trabajo de investigación, se tienen:

- Contribuir a la documentación que existe sobre los servidores Web OSGi, en especial en el idioma español, mediante la revisión bibliográfica y descripción de la especificación OSGi.
- Validar mediante casos de estudio que los servidores Web OSGi son compatibles con tecnologías ampliamente usadas hoy en día, tales como: Maven, Spring y Tapestry5.
- Identificar las ventajas que tienen los servidores Web OSGi sobre los servidores Web tradicionales.

Se ha definido demostrar la compatibilidad de la implementación OSGi con tres tecnologías específicas: Maven, Spring y Tapestry5, debido a que resulta inconveniente demostrar la compatibilidad de la implementación OSGi con un gran número de tecnologías similares. Se considera de mejor conveniencia para este trabajo de investigación, escoger una muestra limitada pero significativa de tecnologías muy usadas actualmente en el desarrollo de software.

En cuanto al procedimiento para alcanzar los objetivos anteriormente expuestos, en primera instancia se definirá el ambiente de desarrollo y las herramientas a utilizar durante este Trabajo de Grado. Luego se procederá a identificar y describir gradualmente los casos de estudio sobre la compatibilidad de la especificación OSGi con Maven, Spring y Tapestry, así como también sobre las ventajas y características que la implementación OSGi tiene sobre los servidores Web tradicionales. Por último, estas demostraciones servirán de insumo para exponer las conclusiones de este trabajo de investigación.

1.1 El Problema

El éxito de la plataforma Java no se puede negar, pero tampoco se puede negar que no soporta el desarrollo de sistemas modulares, más allá de la encapsulación orientada a objetos.

La mayoría de grandes proyectos presentan la necesidad de implementar un repertorio de técnicas para compensar la falta de modularidad en Java, pero estas técnicas poseen errores inherentes los cuales tienen un efecto negativo en el ciclo de vida de una aplicación. Es posible conseguir modularidad en Java, pero es más difícil de lo que debería ser.

Los archivos Java o JAR[1] son vistos usualmente como la unidad de modularidad en Java, aunque en realidad sea una ilusión de modularidad. Una vez un JAR es desplegado en un servidor de aplicaciones, todo el contenido de dicho JAR queda expuesto en el espacio de clases de una aplicación y por lo tanto cualquier clase pública en el JAR puede ser vista y usada por cualquier otra clase que exista en el espacio de clases, derrotando así cualquier noción de modularidad.

Los servidores Web tradicionales para aplicaciones Java, aquellos que no implementan la especificación OSGi, no tienen la capacidad de manejar diferentes módulos de una misma aplicación al mismo tiempo, es decir que no pueden manejar diferentes archivos WAR[2] como parte de una sola aplicación. Esto se debe a que dichos servidores Web dedican un *Classpath* diferente a cada archivo WAR, y por lo tanto son tratados como aplicaciones diferentes. Además, este tipo de servidores Web tampoco tienen la capacidad de actualizar aplicaciones “en caliente”, es decir sin necesidad de reiniciar el servidor Web.

Se puede considerar a la especificación OSGi como la capa de modularidad que le hacía falta a la plataforma Java, entendiéndose como modularidad cuando el código de un software está dividido en partes lógicas que representan diferentes aspectos. A continuación se presentan algunos de los factores que dificultan la modularidad y la actualización

[1] Java ARchive
[2] Web application ARchive

dinámica en la plataforma Java.

1.1.1 Control de Visibilidad

Los modificadores de visibilidad de la plataforma Java (`private`, `public` y `protected`) son una solución para el encapsulamiento orientado a objetos y no para la modularidad como tal. Ante la necesidad de que cierto código sea visible a un paquete diferente, se hace necesario que dicho código sea visible a todos los paquetes, utilizando el modificador `public`. Esto significa que la plataforma Java obliga a decidir entre:

1. Debilitar la estructura lógica de la aplicación teniendo en el mismo paquete clases no relacionadas, y así evitar la exposición de API privada.
2. Mantener la estructura lógica de la aplicación usando múltiples paquetes, a expensas de exponer API privada

Con la especificación OSGi es posible empaquetar una aplicación en diferentes archivos JAR independientes y declarar exactamente que parte del código, siendo una clase la mínima parte, es accesible desde cada JAR y mantener esa restricción de visibilidad.

1.1.2 El *Classpath* de Java

Se puede afirmar que el *Classpath* de Java inhibe las buenas prácticas de modularidad, debido a los problemas de manejo de versiones y dependencias que tiene. Cuando el *Classpath* de Java intenta solucionar las dependencias de las librerías y componentes que hacen parte de una aplicación, este no presta ninguna atención a las versiones del código y simplemente toma la primera versión que encuentre. Por esto, el proceso de configuración del *Classpath* de Java a veces se vuelve una tarea de prueba y error, hasta que la Máquina Virtual (MV) de Java deje de quejarse sobre clases faltantes. Es común tener diferentes versiones de la misma clase, librería o componente en el *Classpath*, y eso normalmente conlleva a un conflicto de versiones.[3]

[3] *Cf.* HALL, Richard S- et ál. OSGi in Action – Creating Modular Applications in Java. Manning, 2010, p. 6.

En las implementaciones de la especificación OSGi es muy difícil encontrar el error `NoSuchMethodError` y que se deba a una incorrección en el *Classpath* de la implementación OSGi. OSGi es de gran ayuda en estos casos al asegurarse de que las dependencias de las aplicaciones sean satisfechas antes de permitir ejecutar el código.

La especificación OSGi también provee un poderoso manejo de versiones. Cada aplicación puede tener definida explícitamente las versiones de las dependencias que necesita, lo que da la posibilidad de tener diferentes versiones de la misma dependencia en un mismo ambiente, pero la aplicación solamente haría uso de la versión que le ha sido definida.

1.1.3 Implementación y Administración

A la plataforma Java también le falta soporte en cuanto a la implementación y administración de aplicaciones, lo cual dificulta la actualización de aplicaciones una vez han sido implementadas. Si se considera por ejemplo el requerimiento de soportar un mecanismo de *plug-in* dinámico, la única manera de lograr ese requerimiento es usando *classloaders*, los cuales tienden a producir errores y además es programación de bajo nivel. Los *classloaders* nunca fueron destinados a ser una herramienta común para los desarrolladores, pero muchos de los sistemas actuales requieren de su uso.[4]

La especificación OSGi provee la capacidad de empaquetar una aplicación en archivos JAR lógicamente independientes e implementar solamente aquellas partes estrictamente necesarias. La modularidad de OSGi está preparada para proveer un mecanismo de extensibilidad o *plug-in* más conveniente, incluyendo soporte para el dinamismo en tiempo de ejecución.

1.1.4 Servidores Web para aplicaciones Java

Una de las exigencias que demandan los procesos de desarrollo y aplicación de software es la instalación y actualización de manera rápida e invisible al usuario. En el mundo de Java

[4] *Ibíd.*, p. 7.

los servidores Web como Apache Tomcat, Glassfish y JBoss son algunos de los más usados y convenientes a la hora de implementar y brindar servicios Web para aplicaciones Java. Las versiones de estos servidores de aplicaciones que no implementan la especificación OSGi deben ser detenidos en caso de que alguna aplicación que hospeden, necesite ser actualizada.

La especificación OSGi provee un modelo de componentes más completo y dinámico que los actuales ambientes más usados comúnmente. Un servidor Web que implemente la especificación OSGi puede contener aplicaciones y componentes que pueden ser instalados, re-iniciados, actualizados, detenidos o des-instalados remotamente (si así se quiere) sin necesidad de re-iniciar el servidor Web, haciendo cualquiera de estos pasos completamente invisible al usuario; pero además, por proveer una capacidad modular mayor, permite la actualización o extensión de aplicaciones de una forma más rápida.

La especificación OSGi está lista para ser tenida en cuenta en el desarrollo empresarial y a gran escala, resolviendo muchas de las desventajas que la plataforma Java y sus servidores de aplicaciones poseen actualmente.

1.2 OSGi como solución

La plataforma de servicios OSGi está compuesta en dos partes: el *Framework* y los servicios estándar. El *Framework* es la implementación que provee la funcionalidad de OSGi y los servicios estándar definen API re-usables para tareas comunes de administración y seguimiento.

En el pasado “OSGi” era el acrónimo en inglés de “*Open Services Gateway Initiative*”. Después del lanzamiento de la tercera especificación, la Alianza OSGi definió oficialmente el nombre de “OSGi” como una marca.

La especificación OSGi para el Framework y los servicios estándar es manejada por la Alianza OSGi. La especificación OSGi se encuentra actualmente en su 4ta revisión, y es

usada ampliamente en industrias como la automotriz, tecnología móvil, aplicaciones de escritorio y más recientemente aplicaciones empresariales.

1.2.1 La Alianza OSGi

La Alianza OSGi es una corporación sin ánimo de lucro fundada por Ericsson, IBM, Motorola y Sun Microsystems, entre otros, en Marzo de 1999. Su misión es crear especificaciones abiertas para la distribución de servicios en redes locales y dispositivos.

Se define así misma como un consorcio mundial de innovadores de tecnología que adelanta un proceso maduro y probado para asegurar inter-operabilidad de aplicaciones y servicios basados en su plataforma de integración de componentes, y líder del estándar para la siguiente generación de servicios de Internet para hogares, autos, teléfonos móvil, escritorios y demás ambientes. [5]

La Alianza OSGi provee especificaciones, referencias de implementación, casos de prueba y certificación para fomentar un valioso ecosistema inter-industria. La adopción de la plataforma reduce el tiempo de salida al mercado y los costos de desarrollo porque la integración de módulos pre-compilados y pre-probados reduce los costos de mantenimiento y provee constantes oportunidades por la posibilidad de actualizar o brindar servicios y aplicaciones dinámicamente. La idea es brindar una arquitectura común y abierta para proveedores de servicios, desarrolladores, proveedores de software y de equipos, para desarrollar, desplegar y administrar servicios de una manera muy coordinada.

Los siguientes son algunos de los beneficios que la Alianza OSGi describe sobre la Plataforma de Servicios OSGi: [6]

- Menos Complejidad: Desarrollar con tecnología OSGI significa desarrollar *bundles*. Los *bundles* esconden su contenido, es decir la lógica del negocio, y se comunican entre sí a través de servicios bien definidos. Al esconder el contenido interno y se adquiere más libertad para realizar cambios posteriores, lo que reduce el número de *bugs*.

[5] OSGi Alliance | About: http://www.osgi.org/About/HomePage
[6] OSGi Alliance | About / Benefits of Using OSGi: http://www.osgi.org/About/WhyOSGi

- Re-uso: El modelo de componentes OSGi hace que sea muy fácil usar componentes de terceros en una aplicación. Un número creciente de proyectos de código abierto proveen archivos JAR listos para OSGi. Sin embargo, librerías comerciales también están comenzando a estar disponibles como *bundles*.
- Fácil Implementación: La tecnología OSGi no es solo un estándar para componentes en sí, sino que también especifica como son instalados y administrados. Esta API de administración estandarizada hace que sea muy fácil integrar la tecnología OSGi en sistemas existentes y futuros.
- Actualización Dinámica: El modelo de componentes OSGi es un modelo dinámico. Los bundles pueden ser instalados, iniciados, detenidos, actualizados y des-instalados sin tener que parar todo el sistema. Muchos programadores de Java no creen en que esto se pueda hacer de manera confiable y por lo tanto inicialmente no lo usan en modo de producción. Sin embargo, después de usar el modelo en modo de desarrollo por un tiempo, la mayoría se da cuenta de que de hecho funciona y reduce los tiempos de implementación.
- Adaptación: El modelo de componentes OSGi está diseñado para permitir la "mezcla" y compatibilidad de componentes. Esto requiere que las dependencias de los componentes necesiten ser especificadas y requiere que los componentes "vivan" en un ambiente en donde sus dependencias opcionales no siempre estén disponibles. El servicio de registro de OSGi es un registro dinámico en donde los *bundles* pueden registrar, obtener y escuchar servicios. Este modelo de servicios dinámico permite a los *bundles* descubrir que capacidades están disponibles en el sistema y adaptarse a las funcionalidades que pueden proveer. Esto hace que el código sea flexible a cambios.
- Transparencia: Los *Bundles* y servicios son "ciudadanos" de primera clase en el ambiente OSGi. La API de administración provee acceso al estado interno de un *bundle* así como también a como está relacionado con otros bundles. Por ejemplo, la mayoría de *frameworks* provee una línea de comando que muestra este estado interno. Las aplicaciones pueden ser detenidas parcialmente para depurar cierto problema. En vez de soportar miles de líneas de *logging* y largos periodos de re-inicio, las aplicaciones OSGi a menudo pueden ser depuradas a través de línea de

comando.

- Control de Versiones: La tecnología OSGi soluciona uno de los grandes problemas de los JAR. El problema consiste en que si se tiene una librería A que funciona con una librería B; versión=2, y se necesita implementar una librería C, pero esta librería C funciona sólo con la librería B; versión=3, entonces la librería C no se podría implementar en la aplicación. En el ambiente OSGi todos los *bundles* están cuidadosamente versionados y sólo aquellos bundles que pueden colaborar entre sí, están conectados en el mismo "espacio de clases". Esto permite que ambos *bundles* A y C funcionen con sus propias librerías. A pesar de que no es recomendado diseñar sistemas con este conflicto, es útil tener un *Framework* que lo soporte.
- Simple: La API de OSGi es sorprendentemente simple. Su núcleo (paquetes principales) es sólo un paquete y menos de 30 clases o interfaces. Este paquete es suficiente para escribir *bundles*, instalarlos, iniciarlos, detenerlos y des-instalarlos, e incluye todas las clases de seguridad y *listeners* que se necesitan. Hay pocas API que proveen tanta funcionalidad, para ser tan pequeñas.
- Rápido: Una de las principales responsabilidades del *Framework* OSGi es la de cargar clases desde los *bundles*. En Java tradicional, los JAR son completamente visibles y están puestos linealmente en una lista. Buscar una clase significa buscar a través de esa lista (a menudo bastante larga). En contraste, OSGi pre-conecta los *bundles* y sabe exactamente que *bundle* provee cual clase. Este ahorro en la búsqueda significa más velocidad
- *Laziness*: La evaluación perezosa es buena en el software y la tecnología OSGi tiene muchos mecanismos para hacer cosas solamente cuando sea realmente necesario. Por ejemplo, los *bundles* pueden ser iniciados desde el inicio del servidor OSGi o pueden ser configurados para iniciar solo cuando otro *bundle* le necesite. Los servicios pueden ser registrados pero solamente creados cuando son usados. Estas especificaciones han sido mejoradas muchas veces para permitir esta clase de escenarios perezosos que pueden ahorrar costos en el tiempo de ejecución.
- No invasivo: Las aplicaciones (*bundles*) en OSGi pueden virtualmente usar cualquier funcionalidad de la MV sin que OSGi les restrinja. Una buena práctica en

OSGi es escribir POJOs, y por esta razón no existen interfaces especiales para los servicios OSGi, incluso un objeto cadena o `String` puede ser un servicio OSGi. Esta estrategia hace que el código de una aplicación sea más fácil de llevar a otro ambiente.

Portabilidad: Una de las metas originales de Java es la de poder ejecutarse en cualquier ambiente. Obviamente no es posible ejecutar cualquier código en cualquier lugar porque las capacidades de las MV difieren. Una MV de un teléfono móvil seguramente no soportará las mismas librerías que una computadora que ejecuta aplicaciones bancarias. OSGi tiene dos cosas en cuenta, primero, la API de OSGi no debe usar clases que no están disponibles en todos los ambientes, y segundo, un *bundle* no debe iniciar si contiene código que no está disponible en el ambiente de ejecución.

1.3 El Framework OSGi

El *Framework* OSGi forma el núcleo de las Especificaciones de la Plataforma de Servicios de OSGi, el cual provee un Framework Java de propósito general y seguro que soporta el despliegue de aplicaciones conocidas como *bundles*.

Las aplicaciones que implementan la especificación OSGi pueden descargar e instalar OSGi *bundles*, y removerlos en cualquier momento. El *Framework* maneja la instalación y actualización de *bundles* de manera dinámica y escalable. Esta especificación permite la existencia de múltiples implementaciones del *Framework*, por ejemplo: Apache Felix, Eclipse Equinox y Eclipse Virgo.

Un modelo de programación consistente facilita a los desarrolladores de *bundles* hacer frente a problemas de escalabilidad, el cual es un aspecto muy importante porque el *Framework* tiene la intención de funcionar en dispositivos con características de hardware diferentes. El *Framework* OSGi permite a los *bundles* seleccionar una implementación correcta en tiempo de ejecución a través del Registro de Servicios. Dichos *bundles* registran nuevos servicios, reciben notificaciones sobre el estado de servicios, o buscan servicios

disponibles para adaptarse a las características del dispositivo en el que se encuentran. Esta característica del *Framework* permite que *bundles* sean instalados para agregar funcionalidad o modificar y actualizar *bundles* existentes sin necesidad de reiniciar el sistema.

Conceptualmente el *Framework* tiene 4 capas:

- Capa de Seguridad
- Capa Modular
- Capa de Ciclo de Vida
- Capa de Servicios

Ilustración 1: Capas del Framework OSGi

1.3.1 Capa de Seguridad

Esta capa es una capa opcional, basada en arquitectura de Seguridad de Java 2[7] y provee la infraestructura necesaria para desplegar y administrar aplicaciones que deben ejecutarse en ambientes muy detalladamente controlados[8].

La Plataforma de Servicios OSGi puede autenticar código por ubicación y por firma. Existen servicios definidos en capas de alto nivel que administran los permisos que estás

[7] Java 2 Security Architecture Version 1.2, Sun Microsystems, Marzo 2002
[8] *Cf.* OSGi Alliance, The. OSGi Service Platform Core Specification. Release 4, Version 4.2, 2009, p. 11.

asociados a la unidad de código a autenticar. Estos servicios son:

- Servicio Administrador de Servicios (*Permission Admin Service*): Administra los permisos basado en ubicación absoluta.
- Servicio Condicional Administrador de Servicios (*Conditional Permission Admin Service*): Administra los permisos basado en un modelo condicional, en donde las condiciones revisan ubicación o firmante.

El *Framework* OSGi hace uso de los permisos de Java 2, y asocia cada *bundles* con un conjunto de permisos específicos. Durante tiempo de ejecución, los permisos son obtenidos a través del Administrador de Seguridad. Hay un conjunto de permisos llamados "Permisos Implícitos" (*Implied Services*), los cuales son concedidos a los *bundles* por defecto. Estos permisos son necesarios para la operación normal de los *bundles*. Estos Permisos Implícitos se listan a continuación[9]:

- Permiso de Archivo para LEER (*READ*), ESCRIBIR (*WRITE*) y ELIMINAR (*DELETE*)
- Permiso de Propiedad para LEER paquetes en `org.osgi.framework.*`
- Permiso de Administrador para ejecutar las acciones llamadas *RESOURCE, METADATA, CLASS* y *CONTEXT*.

Otro conjunto de permisos son denominados "Permisos Basados en Filtros" (*Filter Based Permissions*). Un ejemplo de este tipo de permisos es cuando a un *bundle* se le da el permiso de administrar otros *bundles*. Esto se hace usando un filtro para el nombre del permiso. Por ejemplo, un *bundle* puede tener le permiso de obtener todos los servicios registrados por *bundles* que vengan de una ubicación específica:

```
ServicePermission("(location=https://www.ejemplo.com/*)", GET)
```

Esto provee un modelo muy conveniente porque permite que operadores permitan que un grupo de *bundles* colaboren de manera estrecha sin requerir espacios de nombres para servicios, paquetes o *bundles* que sean *ad hoc*. Más específicamente, el filtro puede contener:

[9] *Ibíd*., p. 23.

- ID: El ID de un *bundle*
- Ubicación
- Firmante
- Nombre simbólico

Entrar en detalle sobre como se deben firmar los JAR para que sean compatibles con la Capa de Seguridad en OSGi, se sale del propósito de este Trabajo de Grado, en especial porque ningún caso de estudio que realizaremos hará uso extenso de esta capa.

1.3.2 **Capa Modular**

La Capa Modular define el concepto modular de OSGi: *bundle*, que es simplemente un JAR con metadata extra. Un *bundle* contiene los archivos `class`, los recursos y un archivo metadata como se muestra en la Ilustración 2: Capa Modular de la Especificación OSGi. Sin embargo, existe un tipo MIME reservado para los *bundles* de OSGi, que puede ser usado para distinguir *bundles* de JAR normales[10]:

```
application/vnd.osgi.bundle
```

Los Bundles son mejores que los JAR ya que dan la posibilidad de declarar explícitamente aquellos paquetes dentro del JAR son externamente visibles (i.e. Paquetes Exportados) y aquellos paquetes de los cuales este depende (i.e Paquetes Importados). El principal beneficio de declarar paquetes exportados e importados es que el *framework* OSGi puede administrar y verificar la consistencia de los bundles automáticamente; este proceso se llama “Resolución de bundles”, y consiste en relacionar paquetes exportados con paquetes importados.

[10] *Cf.* OSGi Alliance, The. OSGi Service Platform Core Specification. Release 4, Version 4.2, p. 27.

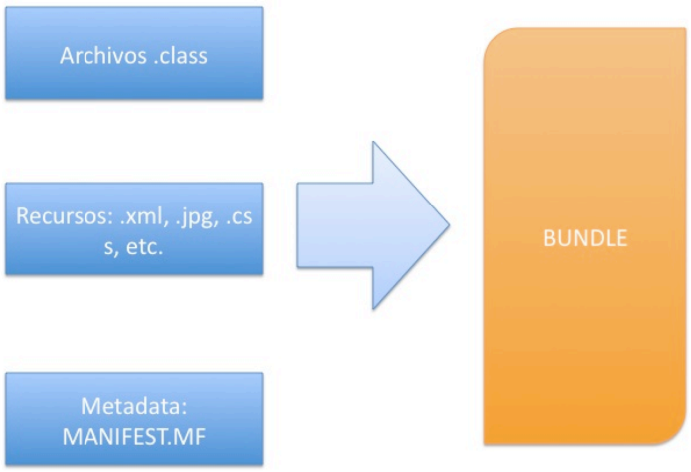


Ilustración 2: Capa Modular de la Especificación OSGi

Cabeceras de un *bundle*

Un *bundle* contiene información descriptiva sobre si mismo en un archivo llamado MANIFEST, el cual normalmente se encuentra en la ruta `META-INF/MANIFEST.MF`. Todas las implementaciones de la especificación OSGi debe cumplir con las siguientes características:

- Procesar la parte principal del archivo MANIFEST. Secciones individuales del mismo son solamente usadas durante la verificación de firma del *bundle*.
- Ignorar cabeceras desconocidas. El desarrollador puede incluir cabeceras adicionales, que no sean reconocidas por OSGi, pero estas no afectan el funcionamiento del *bundle*.
- Ignorar atributos y directivas desconocidas

Las cabeceras del MANIFEST deben seguir estrictamente una serie de reglas definidas en *OSGi IANA Mime Type,* http://www.iana.org/assignments/media-types/application/vnd.osgi.bundle. Los nombres reservados de las cabeceras del MANIFEST de OSGi están definidos en *OSGi Header Name Spage Registry, http://www.osgi.org/headers*.

Cada cabecera del MANIFEST tiene su propia sintaxis, pero comúnmente siguen el

siguiente patrón:

```
Cabecera ::=cláusula (‘,’ cláusula ) *
Cláusula ::= path (‘;’ path ) *
                  (‘;’ parámetro ) *
```

Un parámetro puede ser una directiva o un atributo. Una directiva es una instrucción que tiene una semántica implícita para el *Framework*. Un atributo es usado para propósitos de comparación.

Las cabeceras definidas por la especificación OSGi se listan a continuación, con un valor de ejemplo para cada una entre paréntesis. La mayoría de ellas son opcionales[11].

- `Bundle-ActivationPolicy (lazy):` Especifica como el *`Framework`* debe activar el *`bundle`* una vez iniciado.
- `Bundle-Activator (com.foo.bar.Activator):` Especifica el nombre de la clase a usar para iniciar y detener el *`bundle`*.
- `Bundle-Category (osgi, test, nursery):` Contiene una lista de catearías separada por comas.
- `Bundle-Classpath (/jar/http.jar,.):` Define una lista separada por comas de rutas a archivos JAR o directorios (dentro del *bundle*) que contienen clases o recursos. El punto (‘.’) especifica el directorio raíz del *bundle*, el cual es la ruta por defecto.
- `Bundle-Copyright (OSGi © 2002):` Contiene especificación sobre derechos de autor del *bundle*.
- `Bundle-Description (Network Firewall):` Define una corta descripción sobre el *bundle*.
- `Bundle-DocURL (`http://www.foo.com/Firewall/doc`):` Contiene una URL en donde se encuentra documentación sobre el *bundle*.
- Bundle-Icon (/icons/foo-logo.png;size=64): Provee una lista de URLs que contienen iconos que representan al *bundle*. Se puede definir el atributo *size*, el cual

[11] *Cf.* OSGi Alliance, The. OSGi Service Platform Core Specification. Release 4, Version 4.2, p. 28.

indica el tamaño del icono en pixeles. Las URL son interpretadas como relativas, excepto si se especifica un schema.

- `Bundle-License (http://www.opensource.org/license):` El propósito de esta cabecera es la de automatizar parte del procesamiento de licencias requerido por varias organizaciones, como por ejemplo, aceptar una licencia antes de usar un *bundle*. Esta cabecera es puramente informativa, y no debe ser procesada por el *Framework* OSGi.
- `Bundle-Localization (OSGI-INF/L10n/bundle):` Contiene la ubicación en donde se encuentran archivos de localización. El valor por defecto es OSGI-INF/L10n/bundle, por ejemplo: OSGI-INFO/L10n/bundle_de.properties.
- `Bundle-ManifestVersion (2)`: Define la versión de la especificación OSGi que el *bundle* implementa. El valor 1 significa que el *bundle* sigue la especificación OSGi 3.0 y el valor 2 para la especificación OSGi 4.0. Un poco confuso, pero así lo ha decidido la Alianza OSGi.
- `Bundle-Name (Firewall)`: Esta cabecera define un nombre 'amigable' y legible para el *bundle*.
- `Bundle-NativeCode (/lib/http.DDL:osname=QNX;osversion=3.1):` Contiene una especificación de librerías de código nativo que se encuentran en el *bundle*.
- `Bundle-RequiredExecutionEnvironment (CDC-1.0/Foundation-1.0):` Contiene una lista separada por comas de ambientes de ejecución que deben estar presentes en la Plataforma de Servicios.
- `Bundle-SymbolicName (com.foo.bar)`: Esta cabecera, junto con la cabecera `Bundle-Version` identifican de manera única a un *bundle*, y por lo tanto es una cabecera requerida. Su valor debe acogerse a la convención de nombres de dominio (e.g. `utb.osgi.hola`). El *Framework* produce un error durante la instalación de un *bundle* con un `Bundle-SymbolicName` (el cual es *case sensitive*) y `Bundle-Version` idénticos a los de un *bundle* ya existente.
- `Bundle-UpdateLocation (`http://www.example.com/firewall/bundle.jar`):` Especifica una URL en donde se encuentra la actualización para el *bundle*.
- `Bundle-Vendor (OSGi Alliance):` Contiene una descripción o nombre

sobre el proveedor del *bundle*.

- `Bundle-Version (1.1)`: Esta cabecera específica la versión del *bundle*; es opcional y su valor por defecto es '0.0.0'.
- `Dynamic-ImportPackage (com.foo.bar.*)`: Contiene una lista separada por comas de nombres de paquetes a importar de manera dinámica cuando sean necesarios.
- `Export-Package (org.osgi.util.tracker; version=1.3)`: Contiene la declaración sobre paquetes a exportar, es decir que son públicos.
- `Fragment-Host (org.eclipse.swt;bundle-version="[3.0.0,4.0.0)")`: Define el *bundle* huésped de este fragmento.
- `Import-Package (org.osgi.util.tracker)`: Esta cabecera define uno o más paquetes (proveídos por otros *bundles*) que este *bundle* requiere.
- Require-Bundle (com.acme.chess): Hace explícito los paquetes que este *bundle* necesita importar de otros *bundles* para su funcionamiento correcto.

Versiones

Para describir versiones, se usa la siguiente sintaxis:

```
version     ::=
      major( '.' minor ( '.' micro ( '.' Qualifier) ? ) ? ) ?
major       ::= número
minor       ::= número
micro       ::= número
qualifier   ::= ( alfa-numérico '_' | '-' ) +
```

Un token de versión no puede contener espacios en blanco, y tiene el valor por defecto de 0.0.0.

Las versiones también se pueden definir en rangos, de la siguiente manera:

```
version-range     ::= interval | atleast
interval          ::= (  '[' | '(' ) floor ',' ceiling (']' | ')' )
atleast           ::= version
floor             ::= version
```

```
ceiling          ::= version
```

Si una versión sencilla, de hecho es interpretada como el rango [versión, ∞), y cuando la versión no ha sido definida, el rango por defecto es [0.0.0, ∞). El uso de la coma en el rango de versiones requiere que este se encuentre entre comillas. Por ejemplo:

```
Import-Package: com.acme.foo;version="[1.23, 2)",
      com.acme.bar;version="[4.0, 5.0)"
```

Sintaxis de Filtros

Los filtros son usados para permitir la descripción de una restricción de manera muy concisa. La sintaxis de un filtro está basada en la representación de filtros de LDAP definida en *A String Representation of LDAP Search Filters, RFC 1960, UMich, 1996,* http://www.ietf.org/rfc/rfc1960.txt. Esta sintaxis está dada por la siguiente gramática[12]:

```
filter      ::= '(' filter-comp ')'
filter-comp ::= and | or | not | operation
and         ::= '&' filter-list
or          ::= '|' filter-list
not         ::= '!' filter
filter-list ::= filter | filter filter-list
operation   ::= simple | present | substring
simple      ::= attr filter-type value
filter-type ::= equal | approx | greater-eq | less-eq
equal       ::= '='
approx      ::= '~='
greater-eq  ::= '>='
less-eq     ::= '<='
present     ::= attr '=*'
substring   ::= attr '=' initial any final
initial     ::= () | value
any         ::= '*' star-value
star-value  ::= () | value '*' star-value
final       ::= () | value
value       ::= <see text>
attr        ::= <see text>
```

La Capa de Ciclo de Vida define como los bundles son dinámicamente instalados y administrados en el *framework* OSGi. Esta capa define con precisión las operaciones de ciclo de vida de un *bundle*, esto es: instalación, actualización, iniciación, detenido y des-instalación. Estas operaciones permiten dinámicamente administrar y evolucionar una aplicación de una manera bien definida. Esto significa que los bundles pueden ser

[12] *Cf.* OSGi Alliance, The. OSGi Service Platform Core Specification. Release 4, Version 4.2, p. 33.

agregados o removidos con seguridad sin re-iniciar la aplicación. Así mismo esta capa define como los bundles obtienen acceso al contexto de ejecución, el cual les provee la capacidad de interactuar con el *framework* OSGi y los servicios que provee durante la ejecución.

La Capa de Servicio contiene el "Registro de Servicios" de OSGi. Un *bundles* puede crear un servicio y registrarlo en el registro de servicios y de esta forma otros *bundles* pueden obtener o escuchar dicho servicio. Los servicios son dinámicos, esto quiere decir que un *bundle* puede decidir retirar su servicio del registro de servicios mientras otros *bundles* estén utilizando dicho servicio. Estos *bundles* deben asegurarse de no seguir usando el servicio que ha sido retirado del registro. El objetivo de este dinamismo es poder instalar y des-instalar *bundles* sobre la marcha y que los otros *bundles* sean capaces de adaptarse a estos cambios. Aunque el registro de servicios acepta cualquier objeto como servicio, la mejor práctica es registrar estos objetos usando interfaces y así la implementación del cliente.

Arquitectura del Cargador de Clases

Muchos *bundles* pueden compartir una misma Máquina Virtual, en donde "esconden" o comparten paquetes con otros *bundles*. El mecanismo que hace esto posible es el Cargador de Clases de Java. Cada *bundle* tiene su propio cargador de clases, y junto con los cargadores de clases de los otros *bundles*, crean red de delegación[13].

[13] *Cf.* OSGi Alliance, The. OSGi Service Platform Core Specification. Relese 4, Version 4.2, p. 36.

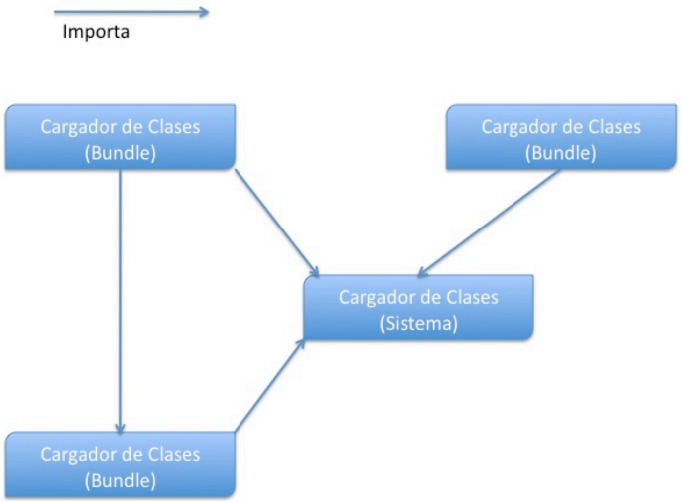


Ilustración 3: Red de Cargadores de Clases

El Cargador de Clases puede cargar recursos de:

- El *Classpath* de Arranque: Contiene los paquetes java.*, y sus implementaciones.
- El *Classpath* del *Framework*: El *Framework* normalmente tiene un Cargador de Clases por separado.
- El Espacio de *Bundles*: Consiste en el JAR asociado con el *bundle*, además de cualquier otro JAR relacionado con el mismo, como los fragmentos.

Un "espacio de clases" son todas las clases accesibles desde un cargador de clases de un *bundle*, por lo que el espacio de clases de un *bundle* puede tener clases provenientes de:

- El Cargador de Clases del padre
- Paquetes importados
- Bundles requeridos
- El Classpath del *bundle* (paquetes privados)
- Fragmentos

Un espacio de clases debe ser consistente, es decir que no existan dos clases con el mismo nombre completo para evitar *Class Cast Exceptions*. Sin embargo, espacios de clases separados en una plataforma OSGi pueden tener el mismo nombre completo. La capa de modularización soporta un modelo en donde múltiples versiones de la misma clase pueden

ser cargadas en la misma Máquina Virtual.

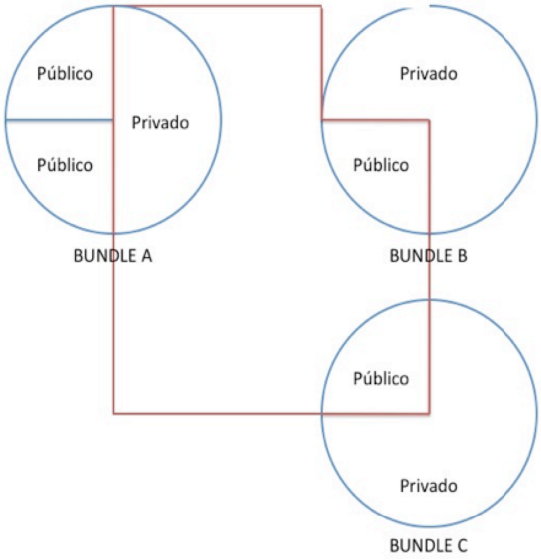


Ilustración 4: Espacio de clases del Bundle A

Antes de que un *bundle* sea usado, el *Framework* debe resolver las restricciones que existan sobre compartimiento de paquetes, es decir las restricciones sobre paquetes importados que los *bundles* necesiten para su funcionamiento.

Proceso de Resolución

Las siguientes cabeceras proveen la información necesaria para la resolución de *bundles*[14]:

- *Bundle-ManifestVersion:* Un *bundle* de expresar la versión de la sintaxis del manifiesto OSGi a través de esta cabecera. Para las versiones del *Framework* OSGi 1.3 en adelante, la versión para esta cabecera es '2', a menos q una versión futura acepte números mayores. Esta cabecera es opcional y obviamente tiene un valor por defecto de '2'.
- *Bundle-SymbolicName:* Esta cabecera es obligatoria, y junto con la cabecera *Bundle-Version*, le permite al *Framework* reconocer un *bundle* de manera única. Esto significa que dos *bundles* con igual *symbolic name* y *version* son tratados como el mismo, y la instalación de un *bundle* con los mismos valores para estas cabeceras

[14] *Cf.* OSGi Alliance, The. OSGi Service Platform Core Specification. Release 4, Version 42, p. 39.

que un *bundle* pre-existente debe fallar. El Framework permite definir los siguientes parámetros para esta cabecera:

- *singleton:* Indica que un *bundle* solamente puede tener una sola versión resuelta, y tiene un valor por defecto de 'false'. El Framework debe resolver máximo una versión del *bundle*, en caso de que existan varias instaladas. Los *bundles* Singleton no afectan la resolución de *bundles* no Singleton con el mismo *symbolic name*.
- *fragment-attachement:* Define como fragmentos pueden ser adheridos, y acepta los siguientes valores:
  - *always*: Fragmentos pueden ser adheridos en cualquier momento, siempre y cuando el *bundle* huésped pase el proveso de resolución exitosamente
  - *never*: Ningún fragmento es permitido
  - *resolve-time*: Fragmentos pueden ser adheridos solamente durante el momento de resolución del *bundle* huésped.

⁀ *Bundle-Version:* Es una cabecera opcional y tiene un valor por defecto de '0.0.0'.

⁀ *Import-Package*: Define las restricciones sobre los paquetes a importar. Esta cabecera permite definir los siguientes atributos:

- *resolution:* Indica si la resolución de un paquete es obligatoria u opcional, para permitir la instalación del *bundle*
- *version:* Consiste en un rango para seleccionar las versiones de los paquetes importados que el *bundle* acepta.
- *bundle-symbolic-name:* Define el *bundle symbolic name* del *bundle* del cual importar los paquetes.
- *bundle-version*: Consiste en un rango para seleccionar la versión del *bundle* "exportador", del cual importar los paquetes.

⁀ *Export-Package:* Esta cabecera, al contrario que *Import-Package*, permite exportar uno o más paquetes, para el uso de otros *bundles*. Se permite exportar el mismo paquete varias veces, pero con atributos diferentes. Los atributos o directivas que se pueden definir en esta cabecera son:

- *uses:* Una lista separada por comas de paquetes que son usados por el paquete que está siendo exportado. Si se usa la coma en el valor de esta directiva, entonces este valor debe estar entre comillas dobles. Por lo tanto, el *bundle* que importe el paquete exportado, también debe importar los paquetes definidos en esta lista.
- *mandatory:* Es una lista separada por comas de atributos. Si se usan comas para definir el valor de un atributo, este valor debe estar encerrado en comillas dobles. Un *bundle* que importe un paquete con esta directiva, debe especificar los mismos atributos, para que el proceso de resolución sea exitoso.
- *include*: Una lista separada por comas de nombres de clases que deben ser visibles para el *bundle* que importa el paquete. El uso de una coma en el valor, requiere que esté encerrado entre comillas dobles.
- *exclude:* Una lista separada por comas de nombres de clases que deben ser invisibles para el *bundle* que importa el paquete.
- *version:* La versión del paquete en cuestión. El valor por defecto es '0.0.0'
- *bundle-symbolic-name:* El *bundle symbolic name* del *bundle* que exporta el paquete.
- *bundle-version*: Es la versión del *bundle* que exporta el paquete.

Un ejemplo de una definición correcta para la cabecera Import-Package es:

```
Import-Package: com.example.foo; com.example.bar;\
version="[1.23,1.24]";resolution:=mandatory
```

Un ejemplo de una definición correcta para la cabecera Export-Package es:

```
Export-Package: com.example.foo;com.example.bar;version=1.23
```

1.3.3 **Capa de Ciclo de Vida**

La Capa de Ciclo de Vida provee una *API* para controlar la seguridad y operaciones de ciclo de vida de los *bundles*. Esta capa se basa en la Capa Modular y en la Capa de Seguridad.

Una vez el *Framework* ha sido creado, debe estar en el estado *INSTALLED*. En este estado, el *Framework* no está activo y no existe un *Bundle Context* válido. Desde este punto el *Framework* puede navegar su ciclo de vida a través de los siguientes métodos[15]:

- *init*: Si el *Framework* no está activo, este método traslada el *Framework* hacia el estado *STARTING*
- *start*: Se asegura de que el *Framework* se encuentra en estado *ACTIVE*. Este método puede ser llamado solamente sobre el *Framework*, ya que en este punto no hay *bundles* ejecutándose.
- *update*: Detiene el *Framework*.
- *stop*: Traslada el *Framework* hacia el estado *RESOLVED* después de pasar por el estado *STOPPED.*
- *uninstall:* No debe ser llamado, ya que lanza una excepción.

Antes de que el *Framework* pueda ser usado, este debe ser inicializado, y eso se lleva a cabo a través del método *init*. Si el *Framework* ha sido inicializado, significa que es operacional, pero ninguno de los *bundles* que contiene se encuentra activo. Esto se refleja en el estado *STARTING*, que es cuando los *bundles* pueden ser instalados. Los *bundles* existentes deben estar todos en el estado *INSTALLED*. En este estado, el *Framework* se ejecuta en el nivel inicial 0.

El proceso de cierre o apagado puede ser iniciado invocando el método *stop*. Una vez este proceso es iniciado, el *Framework* entra en el estado STOPPING. Todos los *bundles* que estén activos son detenidos y libera todo los recursos (como hilos y cargadores de clases). Luego el *Framework* entra en el estado RESOLVED y destruye el *Bundle Context*.

Bundles

Un *bundle* representa un archivo JAR que puede ser ejecutado en el *Framework* OSGi. La Capa de Ciclo de vida maneja el ciclo de vida de un *bundle*, es decir: instalación, actualización y des-instalación.

[15] *Cf.* OSGi Alliance, The. OSGi Service Platform Core Specification. Release 4, Version 42, p. 92.

Un *bundle* es iniciado a través de su *Bundle Activator*, el cual es señalado en el manifest con la cabecera *Bundle-Activator*. La clase que se señala en esta cabecera debe implementar la interfaz `BundleActivator`. Esta interfaz tiene un método `start()` y otro `stop()` los cuales son usados por el programador para registrar el *bundle* junto con otros recursos que sean necesarios. Durante la activación de un *bundle*, este recibe un instancia del *Bundle Context*, el cual le brinda al *bundle*:

- Acceso a información sobre el resto del *Framework*
- Posibilidad de instalar otros *bundles*
- Acceso al Registro de Servicios

El *Framework* le asigna a cada *bundle* un número que lo identifica de manera única, durante toda la "vida" de ese *bundle* en el *Framework*. Este identificador se asigna de forma ascendente a los *bundles* cuando estos son instalados. Un *bundle* también tiene una ubicación definida en el *Framework*, el cual es la URL del JAR. Y como ya se ha dicho anteriormente, un *bundle* también posee un *Symbolic Name* y una versión, ambos definidas por el programador, los cuales, juntos, identifican de manera única y global al *bundle*.

Un *bundle* puede estar en alguno de estos estados, durante su ciclo de vida en el *Framework*:

- *INSTALLED:* El *bundle* has sido instalado exitosamente.
- *RESOLVED:* Todas las clases que el *bundle* necesita se encuentran disponibles. Este estado significa que el *bundle* está listo para ser iniciado o que ha sido detenido.
- *STARTING:* El *bundle* está siendo iniciado y el método `BundleActivator.start` será invocado.
- *ACTIVE:* El *bundle* ha sido exitosamente activado y se está ejecutando. El método `BundleActivator.start` ha sido llamado exitosamente.
- *STOPPING:* El *bundle* está siendo detenido y el método `BundleActivator.stop` será invocado.
- *UNINSTALLED:* El *bundle* ha sido desinstalado.

Cuando un *bundle* es instalado, es también almacenado persistentemente en el *Framework* y permanece allí hasta que sea explícitamente desinstalado.

Los recursos de un *bundle* pueden venir de diferentes fuentes. Pueden venir del mismo JAR, de fragmentos, paquetes importados o del *Classpath* del *bundle*, y cada caso requiere una estrategia de búsqueda diferente.

El *Bundle Context*

La relación entre el *Framework* y los *bundles* instalados en él es realizado por los objetos del *Bundle Context*. Un objeto *BundleContext* representa el contexto de ejecución de un único *bundle* en la Plataforma de Servicios de OSGi[16].

Un objeto *BundleContext* es creado por el *Framework* cuando un *bundle* es iniciado, y es único para cada *bundle*. El *bundle*, por su parte, puede utilizar este objeto para:

- Instalar nuevos *bundles*
- Obtener información sobre otros *bundles*
- Obtener un área de almacenamiento persistente en el ambiente OSGi
- Obtener servicios registrados en el Registro de Servicios
- Registrar servicios
- Suscribirse o dar de baja de eventos transmitidos por el *Framework*

El *System Bundle*

El *Framework* como tal es representado como un *bundle*, y se le da el nombre de *System Bundle*. A través de este, el *Framework* puede registrar servicios que pueden ser usados por otros *bundles*. Ejemplos de estos servicios son Administración de Paquetes y Administración de Permisos (*Package Admin* y *Permission Admin*).

El *System Bundle* es mostrado en la lista de *bundles* instalados, y siempre tiene como ID el valor cero (0), y a diferencia de los otros *bundles*, el *System Bundle* no se puede desinstalar

[16] *Cf.* OSGi Alliance, The. OSGi Service Platform Core Specification. Release 4, Version 42, p. 108.

y si se detiene, todo el *Framework* es detenido[17].

Eventos

La Capa de Ciclo de Vida del *Framework* OSGi tiene los siguientes tipos de Eventos:

- *BundleEvent:* Reporta cambios en el ciclo de vida de los *bundles*
- *FrameworkEvent:* Reporta que el *Framework* ha sido iniciado, si el nivel de inicio ha sido cambiado, si los paquetes han sido refrescados o si se ha encontrado un error.

Eventos que tengan un tipo desconocido son ignorados.

Permiso de Administración

El Permiso de Administración o *Admin Permission*, es un permiso usado para conceder el derecho de administrar el *Framework* con la opción de restringir este derecho a un conjunto dado de *bundles*, llamados *targets*. Por ejemplo, un "Operador" puede otorgarle a un *bundle* el derecho de solamente manejar *bundles* que tengan un "firmante" con nombre "ACME":

```
org.osgi.framework.AdminPermision(
      "(signer=\*, o=ACME, c=co)",…)
```

El primer argumento de `AdminPermission` es una expresión regular que indica el *bundle* o conjunto de *bundles* sobre los cuales el permiso aplica, y el segundo argumento es el nombre del permiso en sí.

1.3.4 Capa de Servicios

La Capa de Servicios OSGi define un modelo colaborativo y dinámico que se encuentra muy integrado con la Capa de Ciclo de Vida. El modelo consiste en "publicar-encontrar y enlazar". Un servicio es un objeto Java que se encuentra registrado en el Registro de Servicios a través de una o más interfaces. Los *bundles* pueden registrar servicios, buscarlos o recibir notificaciones cuando existe algún cambio en el Registro de Servicios.

En la Plataforma de Servicios de OSGi, los *bundles* son construidos alrededor de un

[17] *Cf.* OSGi Alliance, The. OSGi Service Platform Core Specification. Release 4, Version 42, p. 113.

conjunto de servicios disponibles en el Registro de Servicios. OSGi tiene definidos servicios comunes y básicos disponibles una vez el *Framework* ha sido instalado.

Las dependencias que existan entre un *bundle* que brinde un servicio y los *bundles* que consuman este servicio, son manejadas por el *Framework*. Por ejemplo, cuando un *bundle* es detenido, todos los servicios registrados por ese *bundle* deben ser dados de baja por el *Framework* automáticamente.

En general, los servicios registrados son referenciados a través de objectos `ServiceReference`. Esto evita dependencias dinámicas innecesarias entre *bundles* cuando un *bundle* solamente necesita saber sobre un servicio, pero no necesita usarlo.

En la práctica, un desarrollador crea un objeto "servicio" al implementar una interfaz, que contiene la especificación de los métodos públicos, y registra esta interfaz como servicio en el Registro de Servicios. Una vez un *bundle* tiene un objeto "servicio" registrado bajo el nombre de una interfaz, el servicio asociado puede ser adquirido por otros *bundles* usando el nombre de la interfaz. El *Framework* también permite registrar servicios bajo nombres de clases.

Para que un *bundle* pueda usar un objeto servicio e invocar sus métodos, este debe primero obtener un objeto `ServiceReference`. La interfaz `BundleContext` tiene definido dos métodos que un *bundle* puede invocar para obtener objetos `ServiceReference`[18]:

- `getServiceReference(String)`: Este método retorna un objeto `ServiceReference` a un objeto servicio que implementa y está registrado bajo el nombre de la interfaz definida por el argumento `String`. Si existen múltiples objetos para ese servicio, el objeto servicio con el mayor *SERVICE_RANKING* es devuelto. Si también existe un empate en el *SERVICE_RANKING*, entonces el objeto con el menor *SERVICE_ID* (el servicio que fue registrado primero) es devuelto.
- `getServiceReferences(String, String)`: El primer argumento es el

[18] *Cf.* OSGi Alliance, The. OSGi Service Platform Core Specification. Release 4, Version 42, p. 129.

nombre de la interfaz del servicio y el segundo argumento es un filtro. Este método retorna una arreglo de objetos `ServiceReference` que cumplan las siguientes condiciones:

- Implementen y están registrados bajo el nombre de interfaz dado por el primer argumento.
- Satisfagan la búsqueda bajo el filtro dado por el segundo argumento.

Ambos métodos devuelven `null` en caso de no encontrar ningún objeto servicio, y también requieren que el *bundle* que los invoque tenga el permiso `ServicePermission[ServiceReference, GET]`.

El objeto `BundleContext` es usado para obtener el objeto servicio real. Si un *bundle* obtiene un objeto servicio, ese *bundle* se convierte en dependiente del Ciclo de Vida del servicio registrado. Esta dependencia es rastreada por el objeto `BundleContext` usado para obtener el objeto servicio, y es una razón por la cual hay que ser muy cuidadosos al compartir objetos `BundleContext` entre *bundles*.

El método `BundleContext.getService(ServiceReference)` devuelve un objeto que implementa las interfaces definidas en el argumento `ServiceReference`.

Los siguientes son los 2 Eventos de Servicios[19]:

- *ServiceEvent*: Reporta cambios en el Registro de Servicios y tiene los siguientes tipos:
  - *REGISTERED*: Un servicio ha sido registrado
  - *MODIFIED*: Las propiedades de un servicio han sido modificadas
  - *UNREGISTERING:* Un servicio está en proceso de ser dado de baja del registro
- *ServiceListener*: Es llamado por un *ServiceEvent* cuando un objeto servicio ha sido registrado o modificado, o está en el proceso de dar de baja del registro.

[19] *Cf.* OSGi Alliance, The. OSGi Service Platform Core Specification. Release 4, Version 42, p. 131.

Un *bundle* que use un objeto servicio debe registrar un objeto `ServiceListener` para rastrear la disponibilidad del objeto servicio, y tomar medidas apropiadas cuando el objeto servicio esté siendo dado de baja del registro.

Referencias Muertas

El *Framework* debe administrar las dependencias entre los *bundles*. Sin embargo, esta administración está restringida a estructuras del *Framework*. Los *bundles* deben observar eventos generados por el *Framework* para limpiar y remover "referencias muertas".

Una referencia muerta es una referencia a un objeto Java que pertenece al cargador de clases de un *bundle* que ha sido detenido o que está asociado con un objeto servicio que ha sido dado de baja del registro. Java estándar no provee medios genéricos para limpiar referencias muertas, y los desarrolladores de *bundles* deben analizar su código cuidadosamente para asegurarse de que las referencias muertas son eliminadas.

Las referencias muertas son potencialmente peligrosas porque dificultan el trabajo del recolector de basura de Java para "recoger" clases e instancias de *bundles* que han sido detenidos. Esto puede resultar en un incremento significativo de uso de memoria y puede causar fallas a la hora de actualizar librerías de código nativo. Los *bundles* que utilicen servicios, deberían también usar el "Rastreador de Servicios" o *Service Tracker*[20].

Los desarrolladores de servicios pueden minimizar las consecuencias de las referencias muertas siguiendo los siguientes mecanismos:

- Implementar objetos servicios utilizando la interfaz `ServiceFactory`. Los métodos de esta interfaz facilitan el rastreo de *bundles* que utilizan su servicios.
- Los objetos servicio que se brinden a otros *bundles* deben usar un puntero a la implementación del objeto servicio real, así cuando el objeto servicio se vuelva inválido, el puntero toma un valor nulo.

Fábrica de Servicios (*Service Factory*)

[20] *Cf.* OSGi Alliance, The. OSGi Service Platform Core Specification. Release 4, Version 42, p. 132.

Una Fábrica de Servicios permite la personalización del objeto servicio que es devuelto cuando un *bundle* invoca el método `BundleContext.getService(ServiceReference)`.

A menudo, el objeto servicio que es registrado por un *bundle* es devuelto directamente. Sin embargo, si el objeto servicio que está registrado implementa la interfaz `ServiceFactory`, el *Framework* debe invocar métodos de este objeto para crear un único objeto servicio por cada *bundle* que solicite el servicio. Cuando cierto *bundle* deje de usar el objeto servicio, por ejemplo si dicho *bundle* ha sido detenido, el *Framework* debe notificar al objeto `ServiceFactory`.

Los objetos `ServiceFactory` ayudan a manejar las dependencias entre *bundles* que no son estrictamente manejadas por el *Framework*. Al enlazar un objeto servicio devuelto al *bundle* que lo solicita, el servicio puede ser notificado cuando ese *bundle* deja de usar el servicio, y así liberar recursos asociados a ese servicio.

La interfaz `ServiceFactory` define los siguientes métodos[21]:

- `getService(Bundle, ServiceRegistration)`: Este método es llamado por el *Framework* si existe una llamada al método `BundleContext.getService(ServiceReference)` y además se cumple que:
  - El argumento `ServiceReference` se refiere a un objeto servicio que implementa la interfaz `ServiceFactory`.
  - El conteo de uso del objeto servicio por parte del *bundle* es cero, es decir que actualmente el *bundle* no tiene dependencias con el objeto servicio.
- `ungetService(Bundle, ServiceRegistration, Object)`: Este método es invocado por el *Framework* si existe una llamada al método `BundleContext.ungetService`, y además se cumple que:
  - El argumento `ServiceReference` se refiere a un objeto servicio que implementa la interfaz `ServiceFactory.`
  - El conteo de uso del objeto servicio por parte del *bundle* es cero, es decir que actualmente el *bundle* no tiene dependencias con el objeto servicio.

[21] *Cf.* OSGi Alliance, The. OSGi Service Platform Core Specification. Release 4, Version 42, p. 134.

Para que un *bundle* libere un objeto servicio, debe remover la dependencia dinámica del *bundle* que registró el servicio. La interfaz `BundleContext` define un método para liberar objetos servicios llamado `ungetService(ServiceReference)`. Este método devuelve falso si el *bundle* no tiene dependencias con el objeto servicio falso si el *bundle* no tiene dependencias con el objeto servicio, y verdadero en caso contrario.

Remover Servicios

La interfaz `ServiceRegistration` define el método `unregister()` para remover objetos servicios del Registro de Servicios. El argumento `ServiceReference` para este objeto `ServiceRegistration` no puede ser usado para obtener un objeto servicio que haya sido removido del registro. El hecho de que este método exista en el objeto `ServiceRegistration` asegura que solamente el *bundle* que tenga este objeto puede remover el servicio que tenga asociado. Sin embargo, un servicio puede ser removido del registro por un *bundle* diferente al que lo registró, y esto se logra cuando un *bundle* le pasa a otro el objeto `ServiceRegistration`.

Después de que el método `ServiceRegistration.unregister` se ejecuta exitosamente, el objeto servicio debe ser completamente removido del Registro de Servicios del *Framework*. Por lo tanto, objetos ServiceReference obtenidos para ese servicio no puede ser usado nuevamente para obtener el objeto servicio. Invocar el método `BundleContext.getService` debe devolver `null`. Si existen *bundles* con referencias a este objeto servicio, estos deben ser notificados a través del tipo `ServiceEvent.UNREGISTERING`. Una vez un *bundle* recibe este evento, este debe liberar el objeto servicio y cualquier otro recurso asociado, de manera que el objeto servicio pueda ser recolectado por el Recolector de Basura de Java[22].

1.4 Implementaciones de la Especificación OSGi

Las siguientes implementaciones de la especificación OSGi R4.0 son las más utilizadas

[22] *Cf.* OSGi Alliance, The. OSGi Service Platform Core Specification. Release 4, Version 4.2, p. 135.

actualmente:

- Eclipse Equinox[23]: Es la implementación de Eclipse. El objetivo general del proyecto Equinox es ser una comunidad OSGi de primera clase y fomentar la visión de Eclipse como un paisaje de bundles. La última versión estable a la fecha es 3.6.
- Apache Felix[24]: Esta es la implementación de la especificación OSGi bajo la licencia Apache. Esta será la implementación utilizada durante este Trabajo de Grado. La última versión estable a la fecha es 3.0.7.
- Eclipse Virgo[25]: Previamente era un proyecto de SpringSource bajo el nombre de 'SpringSource dm', pero en 2010 pasó a ser de Eclipse. Virgo implementa la consola de Equinox, y el contenedor de servlets Tomcat; además del *framework* Spring y Spring DM. Esto significa que Virgo soporta bundles OSGi así como también aplicaciones JAR normales, en especial las aplicaciones que implementan el *framework* Spring. La última versión estable a la fecha es 2.1.0.RELEASE.

[23] http://www.eclipse.org/equinox/
[24] http://felix.apache.org/site/index.html
[25] http://www.eclipse.org/virgo/

CAPÍTULO
2. CASOS DE ESTUDIO

Este capítulo contiene el contenido central de este Trabajo de Grado, ya que a continuación empezará la demostración y alcance de los objetivos propuestos. En primera instancia se definirá el ambiente y las herramientas que serán utilizadas durante este Trabajo de Grado como medio para alcanzar los objetivos del mismo. En segunda instancia, se realizarán ejemplos básicos tipo 'Hola Mundo' para comenzar la familiarización con las características básicas del *Framework* OSGi. Por último, se procederá a desarrollar los casos de estudio demostrando la compatibilidad de OSGi con 3 tecnologías específicas: Maven, Spring y Tapestry5, así como también se identificarán y demostrarán las ventajas de la especificación OSGi.

2.1 Ambiente y Herramientas

Antes de comenzar con la definición y configuración de las herramientas que se utilizarán durante este Trabajo de Grado, es importante definir el ambiente (i.e. Sistema Operativo) en el cual dichas herramientas estarán configuradas, ya que la instalación y configuración de dichas herramientas difiere de un Sistema Operativo a otro. El sistema operativo que se utilizará durante este Trabajo de Grado es basado en Unix. En el Anexo 1 se explica la configuración de estas herramientas dentro de un ambiente Unix: Ubuntu Linux[26] específicamente.

Debido a que este Trabajo de Grado trata básicamente sobre tecnología Java, es natural y necesario hacer uso de la Máquina Virtual Java, o JVM por sus siglas en inglés. La versión de la instancia de Máquina Virtual Java que se utilizará en este Trabajo de Grado, o JVM por sus siglas en inglés, es la JRE 6[27].

[26] www.ubuntu.com
[27] http://www.oracle.com/technetwork/java/javase/downloads/index.html

Uno de los puntos a demostrar durante este Trabajo de Grado, es la compatibilidad de OSGi con 3 tecnologías Java en específico: Maven, Spring y Tapestry5.

- Maven[28]: Es una herramienta de software destinada a facilitar la administración y la automatización del desarrollo de proyectos. Maven utiliza un archivo XML llamado POM (acrónimo de Project Object Model) para describir el proyecto de software que se está desarrollando, sus dependencias (JARs o bundles) y componentes, y el orden de construcción. Maven descarga dinámicamente librerías de Java y plug-ins Maven de uno o más repositorios. Maven obtiene librerías del Repositorio Central de Maven y otros repositorios Maven. La última versión estable a la fecha es 3.0.3.
- Spring Framework[29]: Este *framework* se ha vuelto popular en la comunidad Java como una alternativa, re-emplazo, o incluso adición al modelo Enterprise JavaBean. Las funcionalidades principales de este *framework* pueden ser usadas en cualquier aplicación Java, pero existen extensiones para desarrollar aplicaciones web con Java EE. La última versión estable a la fecha de este *framework* es 3.0.3. El Spring Framework comprende muchos módulos los cuales proveen un diverso rango de servicios, tales como:
    - Contenedor de Control de Inversión: Es responsable de manejar ciclos de vida de los objetos: crear objetos, llamar métodos de inicialización y configurar objetos conectándolos entre ellos. Los objetos creados por el contenedor son llamados *beans*. Generalmente, el contenedor es configurado cargando archivos XML que contienen las definiciones de los *beans* los cuales proveen la información requerida para crear los *beans*. Los objetos pueden ser obtenidos a través de la 'Inyección de Dependencias'.
    - Acceso de Datos: El *framework* de acceso de datos de Spring soporta otros *frameworks* como JDBC, Hibernate, JDO, JPA, entre otros. Para todos estos *frameworks*, Spring provee:
        - Manejo de Recursos: Obtención y liberación de recursos de la base de datos.
        - Manejo de Excepciones: Traduce excepciones relacionadas al acceso

[28] http://maven.apache.org/
[29] http://www.springsource.org/

de datos a la jerarquía de acceso de datos de Spring.

- Manejo de Transacciones
  - Manejo de Transacciones: La abstracción que provee esta parte del *framework* es capaz de manejar transacciones locales y globales, transacciones anidadas y puntos de seguridad, y funcionar en casi todos los ambientes de Java. También soporta las siguientes estrategias de manejo de transacciones: con conexión JDBC, mapeo objeto-relacional, bases de datos orientadas a objetos, etc.

Tapestry5[30]: Destinado a crear aplicaciones web en Java que sean dinámicas, robustas y altamente escalables. Tapestry divide una aplicación web en un conjunto de páginas, cada una construida desde componentes. Esto provee una estructura consistente, permitiéndole al *Framework* asumir responsabilidad de aspectos claves como: la construcción de URL, almacenamiento persistente en el cliente o en el servidor, validación de formas, internacionalización, y reporte de excepciones. Desarrollar aplicaciones con Tapestry incluye crear plantillas HTML, y combinando estas plantillas con código Java. En Tapestry, las aplicaciones se crean en términos de objetos y específicamente no en términos de URL. Tapestry provee una verdadera orientación a objetos al desarrollo de aplicaciones Web. Durante el desarrollo de este Trabajo de Grado, se usarán los términos 'Tapestry y 'Tapestry5' de manera indistinta. La última versión estable a la fecha es 5.2.4.

Vale la pena señalar que en este Trabajo de Grado no se tiene la intención de dar una detallada descripción sobre otras tecnologías diferentes a OSGi, sino mostrar la integración de estas tecnologías con OSGi.

El Entorno de Desarrollo, o IDE por sus siglas en inglés, que se utilizará durante este Trabajo de Grado será *Eclipse IDE for Java Developers* versión Helios[31]. Eclipse IDE es un ambiente de desarrollo de software implementado principalmente en Java y usado para desarrollar aplicaciones Java. Eclipse IDE emplea el uso de *plug-ins* para proveer todas sus funcionalidades, y de hecho usa la implementación OSGi llamada 'Equinox'. Además

[30] http://www.tapestry.apache.org/
[31] http://www.eclipse.org

Eclipse IDE es gratuito y es uno de los más populares, sino el más popular, ambiente de desarrollo para aplicaciones Java.

Como se indicó en la Introducción, la implementación OSGi que se utilizará será Apache Felix versión 3.0.7[32]. La elección sobre cual implementación usar es más de gustos que de aspectos técnicos, ya que las implementaciones más usadas hoy en día brindan prácticamente los mismos servicios. Sin embargo, Apache Felix es ligeramente más estricto en su interpretación de la implementación OSGi, así que un *bundle* que funcione en Apache Felix, funcionará en Eclipse Equinox o Eclipse Virgo[33].

Debido a que el objetivo general de este Trabajo de Grado es el de demostrar mediante casos de estudio las ventajas que tienen las implementaciones de la especificación OSGi sobre los actuales servidores Web no OSGi, es necesario que se defina un servidor Web no OSGi en específico el cual sirva de referencia comparativa durante dichos casos de estudio. El servidor Web no OSGi que se utilizará será Apache Tomcat en su versión 6.0.32. La principal razón para elegir Apache Tomcat es debido a su amplio y extenso uso dentro de la comunidad de desarrollo Web en Java.

Definidas entonces las herramientas de las que se hará uso durante este Trabajo de Grado, no queda más sino comenzar a ver más de cerca la especificación OSGi y lo que puede hacer, alcanzando uno por uno los objetivos propuestos para este Trabajo de Grado.

2.2 Ventajas de la Especificación OSGi

En la Introducción de este Trabajo de Grado se encuentra una lista de beneficios y ventajas que la misma Alianza OSGi ha definido e identificado sobre la especificación OSGi . De esta lista, se han escogido las siguientes ventajas como las más representativas e interestantes para estudiar:

- Modularidad
- Control de Versiones

[32] http://felix.apache.org/site/index.html
[33] Neil Barlett, OSGi in Practice, Estados Unidos: 2009, p. 28

- Actualización Dinámica
- Extensibilidad
- Adaptación
- Transparencia
- *Laziness*

Esto quiere decir que por cada una de estas ventajas, habrá un caso de estudio en donde se demuestre su presencia en la implementación de la especificación OSGi, y se compare con su ausencia en el servidor para aplicaciones Web tradicional escogido previamente: Apache Tomcat.

2.3 Conociendo el Framework Apache Felix

Antes de comenzar con el primer ejemplo OSGi, se hará una breve introducción a la implementación OSGi Apache Felix y su manejo básico. Lo primero es descargar la implementación, la cual se encuentra en la página Web http://www.felix.apache.org/. El lugar donde el archivo sea descomprimido se denominará `FELIX_HOME` de ahora en adelante.

La principal forma de interactuar con el *Framework* es a través de la interfaz de usuario por línea de comando de Apache Felix, llamada Apache Felix Gogo. Para iniciar el *Framework* se debe ejecutar la siguiente línea de comando desde `FELIX_HOME`: `java -jar bin/felix.jar`.

Como se ha indicado anteriormente, un *bundle* es el término de OSGi para llamar a un componente dentro del *Framework* OSGi. Un *bundle* es un archivo JAR normal pero que además tiene un archivo de configuración llamado MANIFEST. Un *bundle* puede proveer alguna funcionalidad específica o servicio que otros *bundles* pueden usar; los *bundles* solo pueden usar la funcionalidad de otros *bundles* a través de paquetes y servicios compartidos.

Cuando el *Framework* ha sido iniciado, automáticamente instala e inicia todos los *bundles* que se encuentren en la carpeta `FELIX_HOME/bundle`. Por defecto, esta carpeta contiene

bundles relacionados con la interfaz de usuario para interactuar con el *Framework* por línea de comando. Los *bundles* que son instalados en el *Framework* son copiados en una carpeta caché para ejecuciones subsecuentes: `FELIX_HOME/felix-cache`.

Después de iniciar el *Framework* se puede utilizar el comando `help` para ver la lista de comandos disponibles, y `help <nombre-de-comando>` para obtener ayuda sobre algún comando en específico.

La distribución del *Framework* Apache Felix viene por defecto con cuatro *bundles*:

- *Gogo Runtime*: Provee la funcionalidad central de comandos
- *Gogo Shell*: Es la interfaz de usuario por línea de comando
- *Gogo Command*: El conjunto básico de comandos
- *Bundle Repository*: Provee el servicio de manejo de repositorio de *bundles*.

El comando `felix:lb` muestra la lista de *bundles* que se encuentren en el *Framework*, con su número identificador, el estado, el nivel en el *Framework* en el que se encuentran, el nombre y la versión de cada *bundle*. La Ilustración 5 muestra el resultado de ejecutar este comando:

```
fayder-mac:felix-framework-3.0.7 fayder$ java -jar bin/felix.jar
----------------------------
Welcome to Apache Felix Gogo

g! felix:lb
START LEVEL 1
   ID|State      |Level|Name
    0|Active     |    0|System Bundle (3.0.7)
    1|Active     |    1|Apache Felix Bundle Repository (1.6.2)
    2|Active     |    1|Apache Felix Gogo Command (0.6.1)
    3|Active     |    1|Apache Felix Gogo Runtime (0.6.1)
    4|Active     |    1|Apache Felix Gogo Shell (0.6.1)
g!
```

Ilustración 5: Comando felix:lb

Antes de instalar *bundles*, es importante entender como los *bundles* son manualmente desplegados en el *Framework*. Primeramente los *bundles* son instalados y luego son inicializados. Para instalar *bundles* se usa el comando `felix:install` seguido de la dirección o URL donde se encuentra el *bundle* a instalar. Por ejemplo: `felix:install`

`file:/url/bundle.jar`.

Una vez un *bundle* ha sido instalado, puede ser iniciado con el comando `felix:start` seguido del identificador del *bundle*. En el *Framework* Apache Felix todos los *bundles* tienen un número identificador único.

El comando `felix:stop` es usado para detener un *bundle* y el comando `felix:uninstall` es usado para remover un *bundle* de la carpeta `FELIX_HOME/felix-cache`. Los *bundles* pueden ser actualizados usando el comando `felix:update`.

Para detener el *Framework* se usa el comando `stop 0`; aquellos *bundles* que hayan sido instalados serán automáticamente cargados la próxima vez que se inicie el *Framework*.

Con estos comandos previamente explicados, podemos avanzar y realizar un primer ejemplo de una aplicación OSGi muy sencilla. La página Web de Apache Felix provee la documentación sobre el *Framework*: http://felix.apache.org/site/documentation.html.

2.4 *'Hola Mundo' OSGi*

Código Fuente: Proyecto 'hola'

Folder con ejemplo instalado: felix-hola

En este Caso de Estudio se demuestran las funcionalidades básicas del *framework* Apache Felix, y además se demuestra la instalación de un *bundle* OSGi y se describe su ciclo de vida dentro del *framework*. Este *bundle* simplemente imprime la cadena "Hola Mundo OSGi" en la consola de Apache Felix.

Un ejemplo tipo 'Hola Mundo' es lo bastante sencillo y útil para demostrar las funcionalidades básicas del *Framework* Apache Felix. La Ilustración 6 muestra el contenido del previamente nombrado archivo MANIFEST, el cual hace que un archivo JAR sea un *bundle* OSGi, que se usará en este ejemplo.

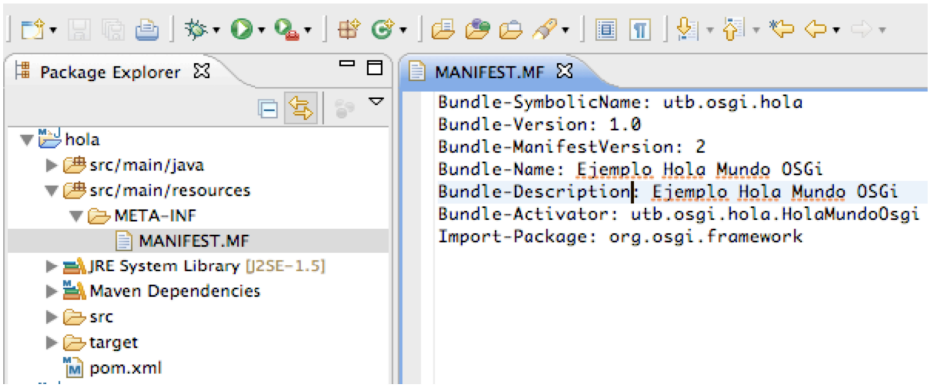


Ilustración 6: Archivo MANIFEST.MF de 'hola'

El archivo MANIFEST es un archivo de texto que consiste de unas cabeceras seguidas por valores que proveen información descriptiva sobre el *bundle*. A continuación se recuerda el significado o propósito de estos encabezados:

- `Bundle-SymbolicName`: Esta cabecera, junto con la cabecera `Bundle-Version` identifican de manera única a un *bundle*, y por lo tanto es una cabecera requerida. Su valor debe acogerse a la convención de nombres de dominio (e.g.

`utb.osgi.hola`). El *Framework* produce un error durante la instalación de un *bundle* con un `Bundle-SymbolicName` (el cual es *case sensitive*) y `Bundle-Version` idénticos a los de un *bundle* ya existente.

- `Bundle-Version`: Esta cabecera específica la versión del *bundle*; es opcional y su valor por defecto es '0.0.0'.
- `Bundle-ManifestVersion`: Define la versión de la especificación OSGi que el *bundle* implementa. El valor 1 significa que el *bundle* sigue la especificación OSGi 3.0 y el valor 2 para la especificación OSGi 4.0. Puede ser un poco confuso, pero así lo ha decidido la Alianza OSGi.
- `Bundle-Name`: Esta cabecera define un nombre 'amigable' y legible para el *bundle*.
- `Bundle-Description`: Define una descripción corta sobre el *bundle*.
- `Bundle-Activator`: Define básicamente el nombre completo de la clase que será usada para iniciar y detener el *bundle*.
- `Import-Package`: Esta cabecera define uno o más paquetes (proveídos por otros *bundles*) que este *bundle* requiere. En este caso, la clase definida en la cabecera `Bundle-Activator` necesita ciertas dependencias del paquete `org.osgi.framework,` y como este paquete lo provee otro *bundle*, es necesario importarlo, como se verá más adelante.

Existen muchas más cabeceras para el archivo MANIFEST, pero con estas bastan para seguir con el ejemplo. Vale la pena resaltar que solamente con la cabecera `Bundle-SymbolicName` un JAR se convierte automáticamente en un *bundle* OSGi, pero como se ha definido también la cabecera `Bundle-Activator`, es necesario realizar otros pasos más para tener un ejemplo que funcione.

Un *Bundle Activator* es una clase en el *bundle* que implementa la interfaz `org.osgi.framework.BundleActivator,` el cual define los métodos `start()` y `stop()` los cuales son invocados cuando un *bundle* es iniciado y detenido, respectivamente[34]. La clase `HolaMundoOsgi` de este ejemplo contiene el código fuente mostrado en la Ilustración

[34] *Cf.* HALL, Richard S. et ál. OSGi in Action – Creating Modular Applications in Java. Manning, 2012, p. 32.

7.

HolaMundoOsgi.java

```
package utb.osgi.hola;

import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;

/**
 * Hola Mundo OSGi!
 *
 */
public class HolaMundoOsgi implements BundleActivator
{

    public void start(BundleContext context) throws Exception {
        System.out.println("Hola Mundo OSGi!");
    }

    public void stop(BundleContext context) throws Exception {
        System.out.println("Adiós Mundo OSGi!");
    }

}
```

Ilustración 7: Clase utb.osgi.hola.HolaMundoOsgi

En este sencillo ejemplo, el método `start()` imprime la cadena 'Hola Mundo OSGi!' cuando el *bundle* es iniciado, como se mostrará más adelante. De igual forma, cuando el *bundle* es detenido este invoca el método `stop()` el cual imprime la cadena "Adiós Mundo OSGi!". Podemos notar que ambos métodos reciben como argumento un objeto tipo `BundleContext` que puede ser usado para interactuar con el contenedor OSGi (i.e. Apache Felix). No es necesario usar este argumento para este ejemplo.

El siguiente paso consiste en crear el *bundle*, es decir el archivo JAR. Como se dijo anteriormente, una de las herramientas que se usará durante este Trabajo de Grado es Maven, y su uso será a partir de este momento, en este caso para generar o crear el *bundle*. Para generar el archivo JAR se utiliza el comando de Maven `mvn clean package`.

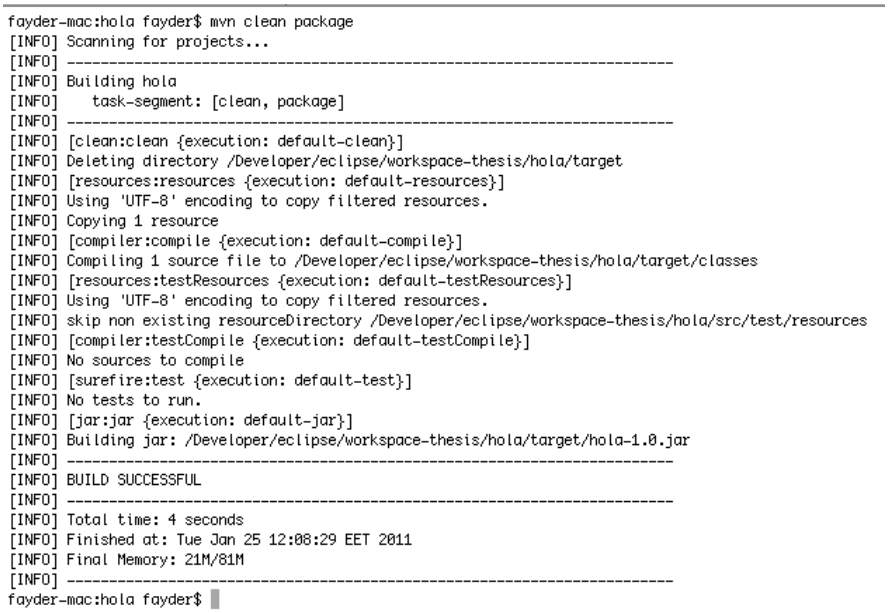


Ilustración 8: Creación de hola-1.0.jar

A manera de aclaración, OSGi funciona perfectamente sin la utilización de Maven, simplemente que esta herramienta facilita mucho el manejo de dependencias de un proyecto, así como la compilación y empaquetamiento del mismo, como se verá en este Trabajo de Grado. En el anterior caso, el comando `mvn package` lo que hace es compilar y ejecutar *Tests*, si existe alguno, y luego empaquetar el proyecto en un archivo JAR o WAR.

En el archivo POM del proyecto `hola` es necesario realizar la configuración necesaria para que Maven tome el archivo MANIFEST que se ha creado dentro de la carpeta `src/main/resources/META-INF` , ya que de otra manera Maven crearía su propio archivo MANIFEST. Para esto, en el POM del proyecto debe existir el siguiente plugin:

```
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-jar-plugin</artifactId>
            <version>2.3.1</version>
            <configuration>
                <archive>
                    <manifestFile>
                        src/main/resources/META-INF/MANIFEST.MF
                    </manifestFile>
                </archive>
            </configuration>
        </plugin>
    </plugins>
</build>
```

Ilustración 9: Maven *plug-in* maven-jar-plugin

Aquí se puede comenzar a observar como trabajan juntos Maven y OSGi, sin embargo más adelante se demostrará de forma más completa la compatibilidad de Maven y OSGi.

El siguiente paso es entonces instalar el *bundle* `hola-1.0.jar` en Apache Felix. Para eso, se inicia el *Framework* Apache Felix, y luego se ejecuta el comando `felix:install` seguido de la ubicaciónn del `JAR hola-1.0.jar`. Luego de instalar el *bundle*, se ejecuta el comando `felix:lb` para ver la lista de *bundles* instalados en el *Framework*. El resultado se muestra en la Ilustración 10.

```
fayder-mac:felix-framework-3.0.7 fayder$ java -jar bin/felix.jar
----------------------------
Welcome to Apache Felix Gogo

g! felix:install file:/Developer/eclipse/workspace-thesis/hola/target/hola-1.0.jar
Bundle ID: 6
g! felix:lb
START LEVEL 1
   ID|State      |Level|Name
    0|Active     |    0|System Bundle (3.0.7)
    1|Active     |    1|Apache Felix Bundle Repository (1.6.2)
    2|Active     |    1|Apache Felix Gogo Command (0.6.1)
    3|Active     |    1|Apache Felix Gogo Runtime (0.6.1)
    4|Active     |    1|Apache Felix Gogo Shell (0.6.1)
    6|Installed  |    1|Ejemplo Hola Mundo OSGi (1.0.0)
g!
```

Ilustración 10: Instalación de ejemplo 'Hola Mundo OSGi'

En este momento el *bundle* de ejemplo se encuentra en un estado de *Installed*, solo es ejecutar el comando `felix:start 6`,

donde el número 6 es el ID del *bundle*, para ver en consola la frase 'Hola Mundo OSGi'. De igual manera si se ejecuta el comando `felix:stop` se verá la frase 'Adiós Mundo OSGi!'. Si se ejecuta el comando `felix:lb` se podrá ver que el nuevo estado del *bundle* es *Resolved*, como se muestra en la Ilustración 11.

```
g! start 6
Hola Mundo OSGi!
g! stop 6
Adi?s Mundo OSGi!
g! felix:lb
START LEVEL 1
   ID|State      |Level|Name
    0|Active     |    0|System Bundle (3.0.7)
    1|Active     |    1|Apache Felix Bundle Repository (1.6.2)
    2|Active     |    1|Apache Felix Gogo Command (0.6.1)
    3|Active     |    1|Apache Felix Gogo Runtime (0.6.1)
    4|Active     |    1|Apache Felix Gogo Shell (0.6.1)
    6|Resolved   |    1|Ejemplo Hola Mundo OSGi (1.0.0)
g!
```

Ilustración 11: Ejemplo 'Hola Mundo OSGi' en acción

Hasta este momento se ha visto que un *bundle* en Apache Felix puede tener los siguientes estados: *Installed, Active, Resolved*. Es importante conocer que significa exactamente estos estados, lo cual se relaciona con el Ciclo de Vida de un *bundle* en el contenedor OSGi.

2.4.1 Ciclo de Vida de un *bundle*

Como se vió en el ejemplo anterior, una vez se ha instalado un *bundle* de manera exitosa, su estado es *Installed*. Luego de ser iniciados exitosamente, el siguiente estado es *Active*. Esto significa que todos las dependencias del *bundle* están disponibles en el *Framework* y han sido resueltas exitosamente para el *bundle* en cuestión (paquetes importados y requeridos). Sin embargo, existe un estado entre *Installed* y *Active*, llamado *Starting*, pero normalmente el proceso de iniciado es tan rápido que no se alcanza a percibir. En caso de que las dependencias del *bundle* no puedan ser resueltas, su estado será *Installed* y el *Framework* advertirá sobre las dependencias que no se han encontrado disponibles.

También se vió que cuando un *bundle* es detenido exitosamente su estado pasó a *Resolved*.

El estado *Resolved* significa precisamente que ese *bundle* no está activo, pero sus dependencias han sido resueltas, es decir que no hay problemas con ese *bundle*, simplemente no está activo. Vale la pena aclarar que de hecho existe un estado entre *Active* y *Resolved*, el cual es *Stopping*, pero al igual que el estado *Starting*, no es fácil percatarse de tal estado debido a la rapidez en que le *bundle* pasa de un estado a otro.

Si se ejecuta el comando `felix:uninstall` seguido del ID del *bundle* que se quiere desinstalar, el *bundle* pasará al estado *Uninstalled*, el cual es un estado simbólico pues lo que realmente sucede es que ese *bundle* es removido completamente del *Framework*.

Existen otro par de comandos en Apache Felix que intervienen en el ciclo de vida de un *bundle*. El comando `felix:resolve` verifica que el *bundle* indicado no tenga problemas con sus dependencias y no significa un cambio de estado para el *bundle*, si el *bundle* se encuentra en estado *Active*. El ciclo de vida de un *bundle* se puede ver en la Ilustración 12. El comando `felix:update`, aunque no está en la Ilustración 12, se usa para actualizar un *bundle* de forma dinámica. Más adelante se describirá más ejemplos usando los comandos de ciclo de vida.

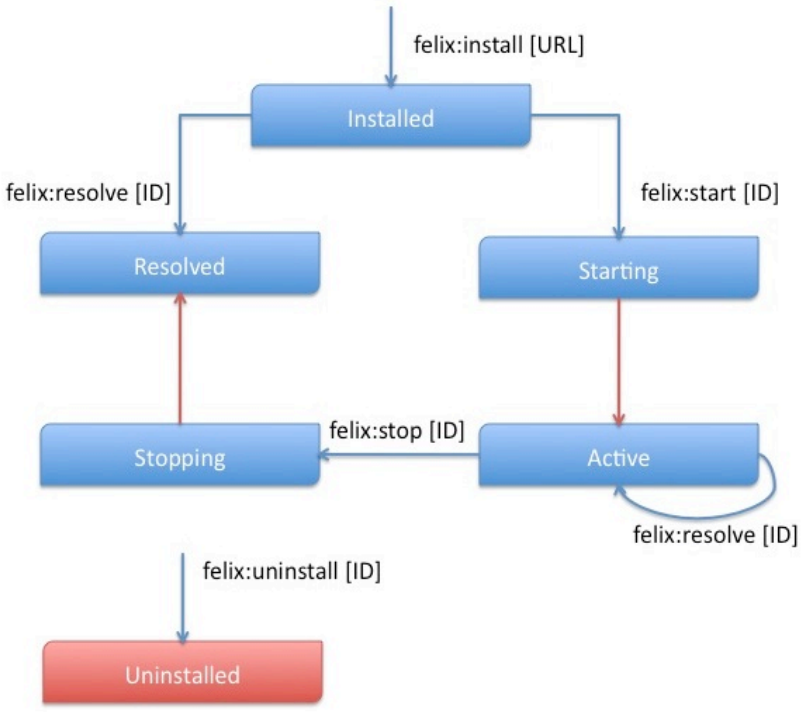


Ilustración 12: Ciclo de Vida de un *bundle*

Vale la pena aclarar que la Ilustración 12, muestra un Ciclo de Vida de un *bundle* exitoso, es decir cuando sus dependencias están disponibles en el *Framework* y han sido satisfechas.

2.4.2 Apache Felix Web Console

Folder con ejemplo instalado: felix-webconsole

El *Framework* Apache Felix posee una serie de 'sub-proyectos' los cuales agregan funcionalidades al *Framework*. El sub-proyecto llamado *Apache Felix Web Console*[35] es una herramienta que permite inspeccionar y administrar *bundles* OSGi a través de una interfaz Web. Es decir que podemos utilizar esta extensión como alternativa a la interfaz de línea de comando que se ha visto hasta ahora.

Esta extensión tiene una sola dependencia que es requerida: Una implementación de la especificación del servicio HTTP OSGi. Existen diferentes opciones que se pueden utilizar y una de las implementaciones dada por el mismo *Framework* se llama *Apache Felix HTTP*

[35] http://felix.apache.org/site/apache-felix-web-console.html

Service Jetty[36]. Este *bundle* es una implementación de la Especificación de Servicio HTTP de OSGi, es cual está descrito en el *OSGi Compendium*[37], y provee una manera de registrar *servlets* y recursos en un contendor de Servlets, y asignarles una URI. La extensión *Apache Felix Web Console* necesita de este *bundle* debido a que está implementado como un *servlet*.

El proceso para instalar *Apache Felix HTTP Service Jetty* es muy parecido al del *bundle* de ejemplo 'Hola Mundo OSGi':

```
fayder-mac:felix-framework-3.0.7 fayder$ java -jar bin/felix.jar
Hola Mundo OSGi!
----------------------------
Welcome to Apache Felix Gogo

g! felix:install http://apache.osuosl.org//felix/org.apache.felix.http.jetty-2.2.0.jar
Bundle ID: 6
g! start 6
g! [INFO] Started jetty 6.1.x at port(s) HTTP:8080
lb
START LEVEL 1
   ID|State      |Level|Name
    0|Active     |    0|System Bundle (3.0.7)
    1|Active     |    1|Apache Felix Bundle Repository (1.6.2)
    2|Active     |    1|Apache Felix Gogo Command (0.6.1)
    3|Active     |    1|Apache Felix Gogo Runtime (0.6.1)
    4|Active     |    1|Apache Felix Gogo Shell (0.6.1)
    5|Active     |    1|Ejemplo Hola Mundo OSGi (1.0.0)
    6|Active     |    1|Apache Felix Http Jetty (2.2.0)
g!
```

Ilustración 13: Instalación de *Apache Felix HTTP Service*

Como se muestra en la Ilustración 13, se usa el comando `felix:install` seguido de una URL que provee el *bundle* que se necesita, y luego el comando `felix:start` seguido del ID del nuevo *bundle*. `La línea [INFO] Started jetty 6.1.x at port(s) HTTP:8080` indica que *Jetty* está esperando solicitudes HTTP en el puerto 8080.

El siguiente paso es instalar el *bundle Apache Felix Web Console*, el cual se realiza de la misma manera. Para verificar que se ha instalado de manera correcta los *bundles* que se neceistan, en la URL http://localhost:8080/system/console y se debe obtener un resultado similar al mostrado en la Ilustración 14.

[36] http://felix.apache.org/site/apache-felix-http-service.html
[37] http://www.osgi.org/Release4/Download

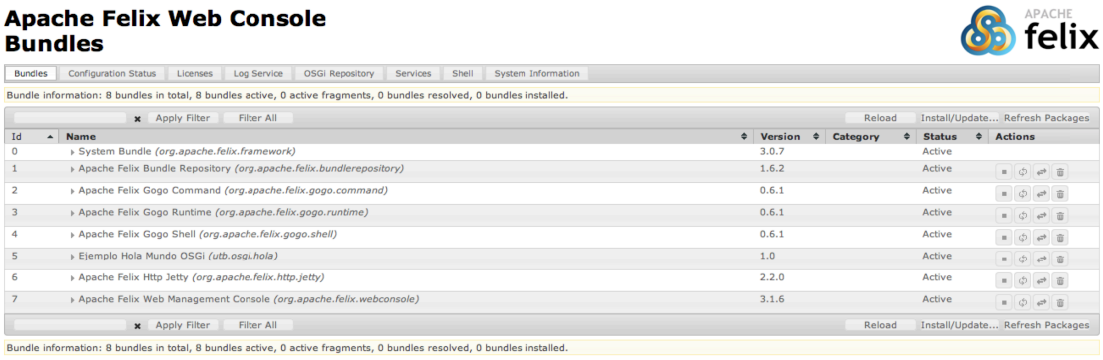


Ilustración 14: ***Apache Felix Web Console***

Si se presta atención a la interfaz de línea de comando, se pude notar que Apache Felix muestra una serie de excepciones. Estas excepciones están relacionadas a dependencias que tiene *Apache Felix Web Console* y que no fueron encontradas dentro del *Framework*; sin embargo, estar excepciones son opcionales y por eso *Apache Felix Web Console* funciona normalmente al menos para lo que lo se necesita en este momento. En el *OSGi Compendium* se describe que la implementación de un *bundle* que utilice paquetes opcionales debe estar preparado para el caso en que estos paquetes no se encuentren disponibles, es decir, que una excepción puede ser mostrada cuando exista una referencia no satisfecha a algún paquete o clase[38].

La extensión *Apache Felix Web Console*, hace mucho más fácil el manejo y administración de *bundles* dentro del *Framework* Apache Felix. Una de las ventajas de esta extensión es que podemos ver con mucha claridad y detalle la configuración OSGi de cada *bundle*; es decir, podemos ver los paquetes que son importados y/o exportados, las versiones, resoluciones, los servicios que un *bundle* en particular brinda, etc. Para ver esta información solo basta con hacer clic en el *bundle* deseado. Por ejemplo, si hacemos clic sobre el *bundle Apache Felix Web Console*, en la parte de 'Manifest Headers', veremos algo parecido a la Ilustración 15.

[38] OSGI Alliance, The. OSGi Service Platform Core Specification. Release 4, Version 4.2, 2009, p. 46

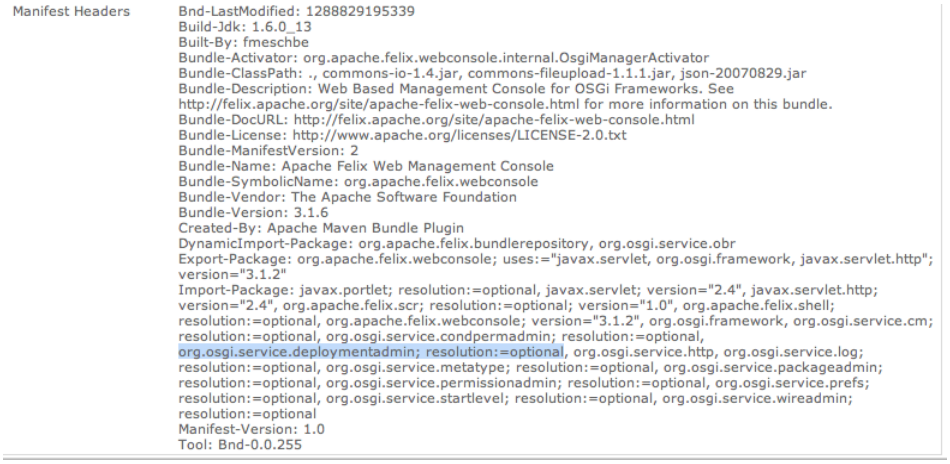


Ilustración 15: Cabeceras de *Apache Felix Web Console*

Según la interfaz de línea de comando, una de las clases faltantes en el *Framework* para *Apache Felix Web Console* es la clase `org.osgi.service.deploymentadmin.DeploymentException`. Como se muestra en la Ilustración 15, parte de la cabecera `Import-Package` de *Apache Felix Web Console* es la dependencia `org.osgi.service.deploymentadmin; resolution:=optional`. Debido a que esta dependencia opcional no se encuentra disponible en el *Framework,* Apache Felix muestra un error tipo `NoClassDefFoundError`, pero aún así *Apache Felix Web Console* funciona normalmente para lo que lo se necesita en este ejemplo.

2.4.3 'Hola Mundo OSGi' Web

Código Fuente: Proyecto 'holaWeb'
Folder con ejemplo instalado: felix-holaWeb

En esta ocasión, se demostrará el mismo ejemplo anterior llamado 'Hola Mundo OSGi', pero esta vez el mensaje se leerá en un navegador Web, aprovechando las mejoras que se le han hecho al *Framework* Apache OSGi.

El primer paso, de nuevo, es el de definir el contenido del archivo MANIFEST.MF. En la Ilustración 16 se puede observar que esta vez existen más paquetes a importar, o dependencias, que con el ejemplo 'Hola Mundo OSGi'. Naturalmente, si se va a desarrollar una aplicación Web en Java, se necesita como mínimo del paquete `javax.servlet` y `javax.servlet.http`.

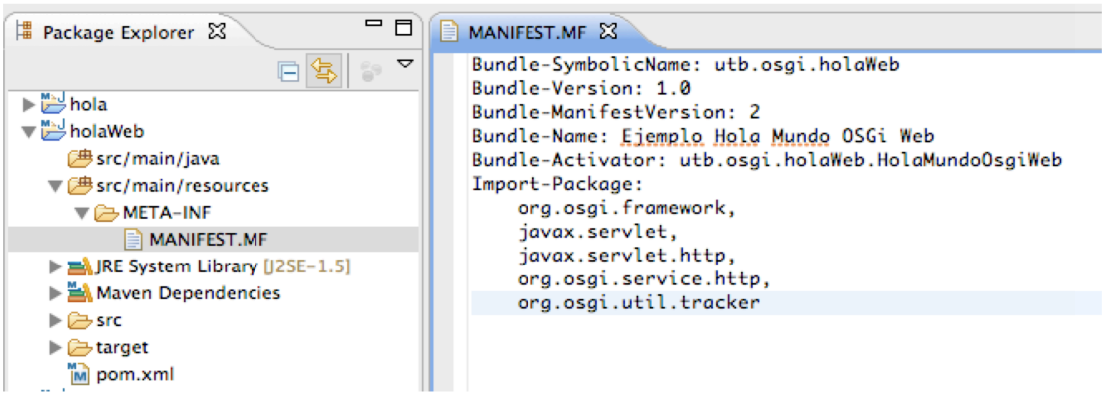


Ilustración 16: Archivo MANIFEST.MF de 'holaWeb'

En el caso de aplicaciones Web OSGi, también se tiene una dependencia a `org.osgi.service.http`, ya que es este bundle el que registra los *servlets* en el *Framework*. Con respecto al paquete `org.osgi.util.tracker`, más delante de hablará de él.

El siguiente paso es desarrollar el *servlet* que imprimirá la oración 'Hola Mundo OSGi Web' en el navegador. La Ilustración 17 muestra como luce esta clase, que se ha denominado `HolaMundoOSGiServlet`, que lo único que hace es sobre-escribir u `Override` el método `doGet()` y este escribe la oración 'Hola Mundo OSGi Web'.

HolaMundoOsgiServlet.java

```
package utb.osgi.holaWeb;

import java.io.IOException;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class HolaMundoOsgiServlet extends HttpServlet{

    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException{
        response.setContentType("text/html");
        response.getWriter().print("<h3> Hola Mundo OSGi Web! </h3>");
    }
}
```

Ilustración 17: Clase HolaMundoOsgiServlet de 'holaWeb'

La siguiente clase a desarrollar se ha denominado `HttpServiceTracker`. Esta clase tiene una referencia a `HttpService`, el cual es el servicio de OSGi que le permite a los *bundles* dinámicamente registrar o dar de baja recursos y *servlets*, es decir que permite relacionar solicitudes URI con carpetas, archivos HTML o con *servlets*.

La clase `HttpServiceTracker`, también extiende la clase `ServiceTracker`, lo que le permite rastrear los servicios registrados en un ambiente OSGi, en el caso de que estos sean actualizados o des-instalados. En este caso, permite rastrear o encontrar el servicio `HttpService`. El método `addingService()` relaciona el *servlet* `HolaMundoOsgiServlet`

con el URI http://localhost:8080/holamundoosgi. De la misma forma, el método `removedService()` rompe la relación que fue establecida anteriormente. La Ilustración 18 muestra esta clase.

HttpServiceTracker.java

```
package utb.osgi.holaWeb;

import javax.servlet.ServletException;

public class HttpServiceTracker extends ServiceTracker {

    public HttpServiceTracker(BundleContext context) {
        super(context, HttpService.class.getName(), null);
    }

    @Override
    public Object addingService(ServiceReference reference) {
        HttpService service = (HttpService) context.getService(reference);

        try {
            service.registerServlet("/holamundoosgi", new HolaMundoOsgiServlet(), null, null);
        } catch (ServletException e) {
            e.printStackTrace();
        } catch (NamespaceException e) {
            e.printStackTrace();
        }

        return service;
    }

    @Override
    public void removedService(ServiceReference reference, Object service) {
        HttpService httpService = (HttpService) service;
        httpService.unregister("/holamundoosgi");
        super.removedService(reference, service);
    }

}
```

Ilustración 18: Clase HttpServiceTracker de 'holaWeb'

Ahora se necesita de un lugar en donde iniciar el `HttpServiceTracker`, y el mejor lugar es una clase que implemente `BundleActivator`, y para este caso a esta clase se le ha llamado `HolaMundoOsgiWeb`. Como se ve en la Ilustración 19, durante la activación del *bundle*, en el método `start()`, también se inicia el objeto `HttpServiceTracker`, y de igual forma se cierra en el método `stop()`.

Ahora, simplemente queda empaquetar el proyecto e instalarlo en el *Framework* Apache Felix. Resulta útil recordar que para empaquetar el proyecto, se utiliza el comando de Maven `mvn clean package`, como se muestra en la Ilustración 20.

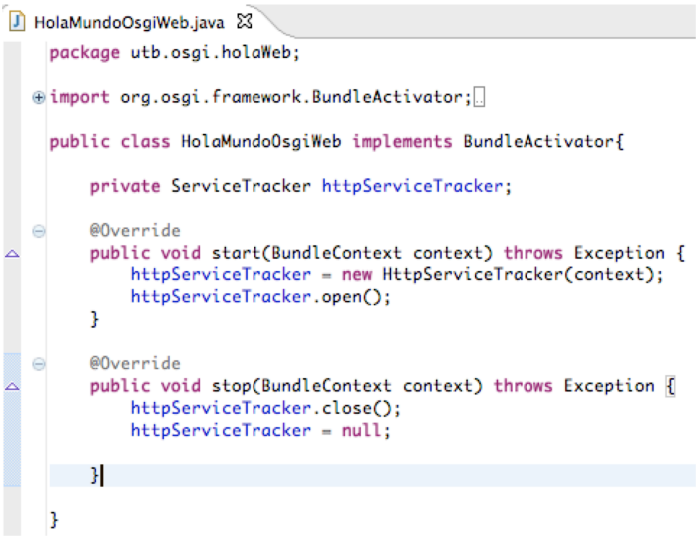


Ilustración 19: Clase HolaMundoOsgiWeb de 'holaWeb'

```
fayder-mac:holaWeb fayder$ mvn clean package
[INFO] Scanning for projects...
[INFO] ------------------------------------------------------------------------
[INFO] Building holaWeb Maven Webapp
[INFO]    task-segment: [clean, package]
[INFO] ------------------------------------------------------------------------
[INFO] [clean:clean {execution: default-clean}]
[INFO] Deleting directory /Developer/eclipse/workspace-thesis/holaWeb/target
[INFO] [resources:resources {execution: default-resources}]
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] Copying 1 resource
[INFO] [compiler:compile {execution: default-compile}]
[INFO] Compiling 3 source files to /Developer/eclipse/workspace-thesis/holaWeb/target/classes
[INFO] [resources:testResources {execution: default-testResources}]
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] skip non existing resourceDirectory /Developer/eclipse/workspace-thesis/holaWeb/src/test/resources
[INFO] [compiler:testCompile {execution: default-testCompile}]
[INFO] No sources to compile
[INFO] [surefire:test {execution: default-test}]
[INFO] No tests to run.
[INFO] [jar:jar {execution: default-jar}]
[INFO] Building jar: /Developer/eclipse/workspace-thesis/holaWeb/target/holaWeb-1.0.jar
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 3 seconds
[INFO] Finished at: Sun Feb 13 20:39:16 EET 2011
[INFO] Final Memory: 22M/81M
[INFO] ------------------------------------------------------------------------
fayder-mac:holaWeb fayder$
```

Ilustración 20: Creación de holaWeb-1.0.jar

Para instalar el *bundle* `holaWeb-1.0.jar` en el *Framework*, se puede hacer uso de la interfaz Web *Apache Felix Web Console* de la que actualmente se dispone. Simplemente hay que dirigirse a la URL http://localhost:8080/system/console/bundles, dar clic en 'Install/Update', navegar donde se encuentra el archivo JAR que se quiere instalar y por último hacer clic en el botón 'Install

or Update' que se encuentra en el dialogo, tal como se muestra en la Ilustración 21.

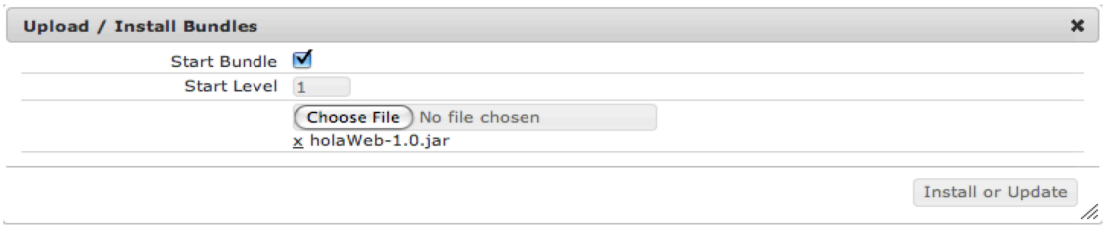


Ilustración 21: Instalación de holaWeb-1.0.jar

Para verificar que todo ha marchado bien hay que dirigirse a la URL http://localhost:8080/holamundoosgi, y se debe ver la oración 'Hola Mundo OSGi Web' en el navegador.

2.4.4 'Hola Mundo OSGi' como Servicio

Código Fuente: Proyectos 'holaServicio' y 'holaConsumidor'
Folder con ejemplos instalados: felix-service

En los anteriores ejemplos se vieron ejemplos bastante sencillos de aplicaciones en el *Framework* Apache Felix, y algunas de sus características básicas como la instalación de *bundles*, el ciclo de vida de un *bundle*, el archivo MANIFEST.MF y las extensiones a Apache Felix como el *Apache Felix Web Console*.

Este Caso de Estudio muestra como se puede publicar o registrar un servicio en el ambiente OSGi, y a su vez como se puede consumir ese servicio. Esto quiere decir que se tendrán 2 *bundles*, uno que provea el servicio y otro que lo consuma. Este ejemplo permitirá comenzar a ver como trabaja la modularidad en el ambiente OSGi.

En OSGi, así como también en las buenas prácticas de Programación Orientada a Objetos (POO), se recomienda programar para una interfaz en vez de para una implementación; por eso se definirá una interfaz por medio de la cual otros *bundles* puedan consumir los servicios que ofrece esta interfaz.

Para simplificar las cosas, se seguirá implementando el ejemplo tipo 'Hola Mundo'. En esta

ocasión se tiene una interfaz llamada `HolaMundoOsgiServicio`, la cual a su vez tiene 2 métodos: `getHola()` y `getAdios()`, así como se ve en la Ilustración 21.

HolaMundoOsgiServicio.java

```
package utb.osgi.holaServicio;

public interface HolaMundoOsgiServicio {

    String getHola();

    String getAdios();

}
```

Ilustración 22: Interfaz HolaMundoOsgiServicio

La clase que implementa esta interfaz, se ha llamado `HolaMundoOsgiServicioImpl`. Como se vé en la Ilustración 23, esta clase regresa la cadena 'Hola Mundo OSGi Servicio' en el método `getHola()`, y la cadena 'Adios Mundo OSGi Servicio' en el método `getAdios()`. Normalmente es mejor usar nombres en inglés para los nombres de clases, métodos y recursos en general, pero se seguirá haciendo una excepción por ser ejemplos muy sencillos.

HolaMundoOsgiServicioImpl.java

```
package utb.osgi.holaServicio.impl;

import utb.osgi.holaServicio.HolaMundoOsgiServicio;

public class HolaMundoOsgiServicioImpl implements HolaMundoOsgiServicio{

    public String getHola() {
        return "Hola Mundo OSGi Servicio";
    }

    public String getAdios() {
        return "Adios Mundo OSGi Servicio";
    }

}
```

Ilustración 23: Clase HolaMundoOsgiServicioImpl de 'holaServicio'

Si se compara la Ilustración 22 y la Ilustración 23, se puede notar que la interfaz y la clase que la implementa se encuentran en paquetes diferentes. Además de ser una buena práctica de POO, la de mantener interfaces y sus respectivas implementaciones en paquetes diferentes, en OSGi esta buena práctica toma relevancia porque permite exportar y hacer público solamente el paquete que contiene la interfaz, y de esta manera asegurar un nivel de visibilidad conveniente al no exponer el paquete de la implementación.

Como ya se sabe, para que otros *bundles* puedan usar cierto servicio, este debe estar registrado en el Registro de Servicios del *Framework*, y esto se hace a través de una clase que implemente `BundleActivator`, o en otras palabras un *Bundle Activator*.

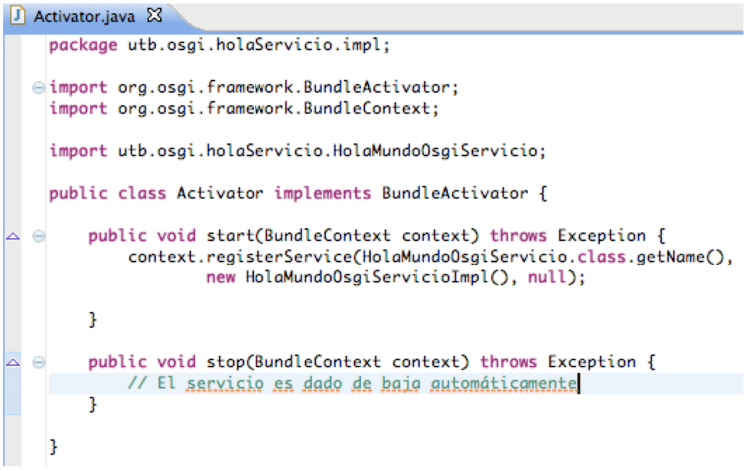


Ilustración 24: Clase Activator de 'holaServicio'

Esta es la primera vez que se usa el argumento `BundleContext context` del método `start()`, y lo se hace para registrar el servicio `HolaMundoOsgiServicio` en el Registro de Servicios de Apache Felix a través del método `registerService()`, al cual se le pasa como argumentos el nombre del servicio y una instancia de `HolaMundoOsgiServicioImpl`; también se pasa un argumento `null` en vez del conjunto de propiedades que el método espera como 3er argumento, pero que en este caso no es necesario. En el método `stop()` no es necesario colocar nada, debido a que el servicio es removido del Registro de Servicios automáticamente cuando el *bundle* es detenido. Si se hubiese registrados otro tipo de recursos entonces si necesitaría removerlos manualmente.

Por último queda por definir el contenido del archivo MANIFEST.MF, el cual se puede ver en la Ilustración 25. La diferencia entre este MANIFEST y los que se han desarrollado en ejemplos anteriores es que esta vez se hace uso de la cabecera `Export-Package`, para exportar el paquete `utb.osgi.holaServicio` y así permitir a otros *bundles* puedan ver y hacer uso de la interfaz `HolaMundoOsgiServicio`. De esta manera, si *bundle* importa el paquete `utb.osgi.holaServicio` y hace uso de la interfaz `HolaMundoOsgiServicio`, este

bundle obtendría una instancia de la clase `HolaMundoOsgiServicioImpl`, debido a que en el Registro de Servicios de OSGi el nombre de dicha interfaz se encuentra relacionada con esta implementación. Puede que se repita mismo nombre de cierta interfaz en el Servicio de Registro de OSGi, pero normalmente tendrían una versión diferente (`Bundle-Version`). En caso de tener la misma versión, el *Framework* determinaría que se trata de la misma interfaz. Más adelante se hablará sobre el Control de Versiones en OSGi.

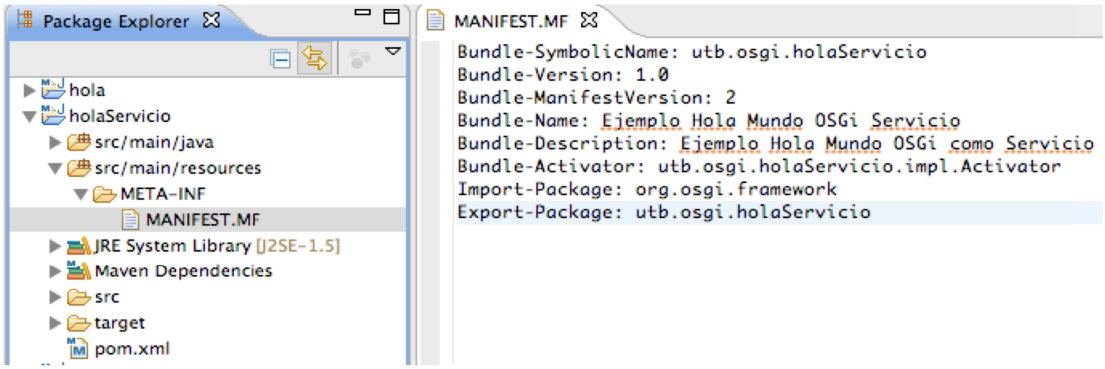


Ilustración 25: Archivo MANIFEST.MF de 'holaServicio'

Ahora solo resta empaquetar el proyecto en un archivo JAR, lo cual se hace de la misma forma en la que se ha venido haciendo: usando el comando `mvn clean package`, e instalarlo en el *Framework*. Para comprobar que el servicio `HolaMundoOsgiServicio` ha sido registrado en el Registro de Servicios de Apache Felix, se puede hacer clic sobre el nombre del *bundle* en la interfaz Web, o *Apache Felix Web Console*, y allí se puede ver que el servicio ha sido registrado satisfactoriamente, tal como se muestra en la Ilustración 26. También vale la penar notar en la misma ilustración, que el paquete `utb.osgi.holaServicio` esta siendo exportado satisfactoriamente.

El siguiente paso para completar este ejemplo es el de desarrollar un *bundle* que consuma el servicio `HolaMundoOsgiServicio`, el cual se ha denominado `holaConsumidor`. Para este caso se empleará un ejemplo similar a `holaWeb`, pero en este caso el mensaje que se leerá en el navegador Web, provendrá del servicio `HolaMundoOsgiServicio`. Esto significa que este *bundle* debe importar en su archivo MANIFEST.MF el paquete `utb.osgi.holaServicio`, y como se está usando MAVEN, es ideal tener acceso a esta dependencia por medio del archivo POM, como el resto de dependencias en los anteriores

ejemplos.

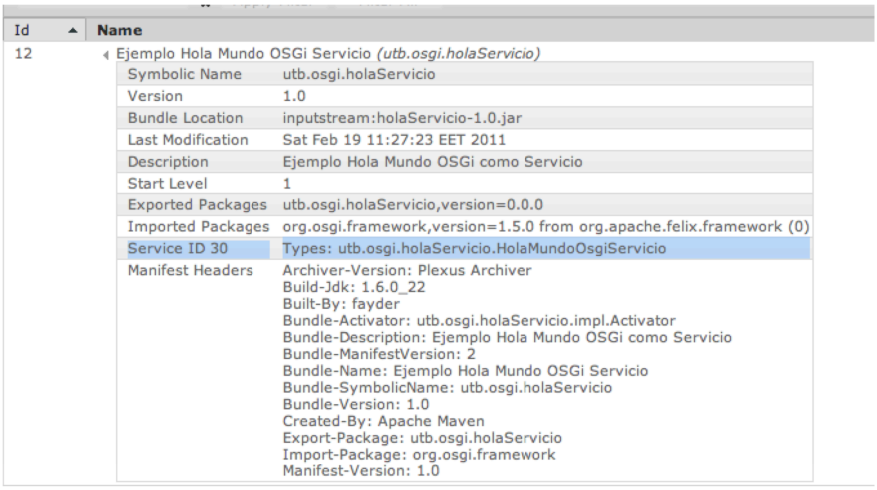


Ilustración 26: *Bundle* 'holaServicio' en *Apache Felix Web Console*

Normalmente las dependencias descritas dentro del archivo POM de un proyecto MAVEN, se encuentran disponibles en repositorios, sean públicos o privados. En este caso, para hacer que los archivos JAR que se han desarrollado, como `holaServicio`, estén disponibles al archivo POM, lo que se hace es instalar estos archivos JAR en el repositorio local, es decir la máquina donde se trabaja. Para hacer esto solo basta con ejecutar el comando `mvn clean install`, en la carpeta principal del proyecto que se quiera instalar en el repositorio local. Para el caso de `holaServicio`, la ejecución de este comando tiene un resultado similar al mostrado en la Ilustración 27.

```
fayder-mac:holaServicio fayder$ mvn clean install
[INFO] Scanning for projects...
[INFO] ------------------------------------------------------------------------
[INFO] Building holaServicio
[INFO]    task-segment: [clean, install]
[INFO] ------------------------------------------------------------------------
[INFO] [clean:clean {execution: default-clean}]
[INFO] Deleting directory /Developer/eclipse/workspace-thesis/holaServicio/target
[INFO] [resources:resources {execution: default-resources}]
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] Copying 1 resource
[INFO] [compiler:compile {execution: default-compile}]
[INFO] Compiling 3 source files to /Developer/eclipse/workspace-thesis/holaServicio/target/classes
[INFO] [resources:testResources {execution: default-testResources}]
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] skip non existing resourceDirectory /Developer/eclipse/workspace-thesis/holaServicio/src/test/resources
[INFO] [compiler:testCompile {execution: default-testCompile}]
[INFO] No sources to compile
[INFO] [surefire:test {execution: default-test}]
[INFO] No tests to run.
[INFO] [jar:jar {execution: default-jar}]
[INFO] Building jar: /Developer/eclipse/workspace-thesis/holaServicio/target/holaServicio-1.0.jar
[INFO] [install:install {execution: default-install}]
[INFO] Installing /Developer/eclipse/workspace-thesis/holaServicio/target/holaServicio-1.0.jar to /Users/fayder/.m2/repository/utb/osgi/holaServicio/1.0/holaServicio-1.0.jar
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 5 seconds
[INFO] Finished at: Sat Feb 19 15:43:07 EET 2011
[INFO] Final Memory: 22M/81M
[INFO] ------------------------------------------------------------------------
fayder-mac:holaServicio fayder$
```

Ilustración 27: Instalación de 'holaServicio' en repositorio maven local

Ahora se peude colocar en el POM a `holaServicio` como dependencia de `holaConsumidor`, pero aclarando que el *scope* de esta dependencia es *provided*, es decir, que será proveída externamente, en este caso por el *Framework* Apache Felix. La ventaja que se obtiene con esto, es que se puede utilizar la API de `holaServicio` de una manera más limpia, en las clases del proyecto `holaConsumidor`. Para leer más sobre el POM de Maven, esta es la guía oficial: http://maven.apache.org/pom.html.

Como se recordaía, el ejemplo 'Hola Mundo OSGi Web' simplemente imprimía un código HTML con la oración 'Hola Mundo OSGi Web!' en el objeto `HttpServletResponse`. Esta vez lo que se hará será hacer uso del servicio `HolaMundoOsgiServicio` para obtener la cadena que se quiere imprimir, como lo se puede observar en la Ilustración 28.

```java
package utb.osgi.holaConsumidor;

import java.io.IOException;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.osgi.util.tracker.ServiceTracker;

import utb.osgi.holaServicio.HolaMundoOsgiServicio;

public class HolaMundoOsgiServlet extends HttpServlet{

    private ServiceTracker serviceTracker;

    public HolaMundoOsgiServlet(ServiceTracker serviceTracker) {
        this.serviceTracker = serviceTracker;
    }

    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {

        HolaMundoOsgiServicio servicio = (HolaMundoOsgiServicio) serviceTracker.getService();

        response.setContentType("text/html");
        if (servicio != null) {
            response.getWriter().println("<h3>" + servicio.getHola() + "</h3>");
        }
    }
}
```

Ilustración 28: Clase HolaMundoOsgiServlet de 'holaConsumidor'

En este ejemplo también se necesita relacionar el *servlet* con una URI, y esto lo se hace en la clase `HttpServiceTracker`. Como se ve en la Ilustración 29, el constructor de esta clase además de tomar un objeto tipo `BundleContext` como parámetro, también toma un objeto tipo `ServiceTracker`, el cual luego se pasa como argumento a la clase `HolaMundoOsgiServlet` en donde, como se vió anteriormente, es usado para obtener el servicio que se requiere. Pero esto significa que este objeto llamado `holaMundoServicioTracker` debe ser inicializado desde la clase que llama a `HttpServiceTracker`, y esa clase es la clase `Activator`.

```java
package utb.osgi.holaConsumidor;

import javax.servlet.ServletException;

import org.osgi.framework.BundleContext;
import org.osgi.framework.ServiceReference;
import org.osgi.service.http.HttpService;
import org.osgi.service.http.NamespaceException;
import org.osgi.util.tracker.ServiceTracker;

public class HttpServiceTracker extends ServiceTracker{

    private ServiceTracker holaMundoServicioTracker;

    public HttpServiceTracker(BundleContext context, ServiceTracker holaMundoServicioTracker) {
        super(context, HttpService.class.getName(), null);
        this.holaMundoServicioTracker = holaMundoServicioTracker;
    }

    @Override
    public Object addingService(ServiceReference reference) {
        HttpService httpService = (HttpService) context.getService(reference);

        try {
            httpService.registerServlet("/holamundo_osgi_servicio",
                    new HolaMundoOsgiServlet(holaMundoServicioTracker), null, null);
        } catch (ServletException e) {
            e.printStackTrace();
        } catch (NamespaceException e) {
            e.printStackTrace();
        }

        return httpService;
    }

    @Override
    public void removedService(ServiceReference reference, Object service) {
        HttpService httpService = (HttpService) service;
        httpService.unregister("/holamundo_osgi_servicio");
        super.removedService(reference, service);
    }
}
```

Ilustración 29: Clase HttpServiceTracker de 'holaConsumidor'

En la clase `Activator` se inicializan ambas referencias tipo `ServiceTracker`, una referencia para encontrar el servicio `HttpService` y otra para encontrar el servicio `HolaMundoOsgiServicio`. El servicio `HttpService` es registrado en el Registro de Servicios por el *bundle* 'HttpServiceBundle', y por su parte el servicio `HolaMundoOsgiServicio` es registrado por el *bundle* 'holaServicio'. Luego de inicializados, se llama al método `open()` para que en efecto puedan encontrar los servicios que se requieren. En el método `stop()` de la clase `Activator`, se invoca el método `close()` para cada referencia tipo `ServiceTracker`, de manera que dejen de rastrear servicios cuando el *bundle* es detenido.

```java
package utb.osgi.holaConsumidor;

import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;
import org.osgi.util.tracker.ServiceTracker;

import utb.osgi.holaServicio.HolaMundoOsgiServicio;

public class Activator implements BundleActivator{

    private ServiceTracker holaMundoServicioTracker;

    private ServiceTracker httpServiceTracker;

    public void start(BundleContext context) throws Exception {
        holaMundoServicioTracker =
            new ServiceTracker(context, HolaMundoOsgiServicio.class.getName(), null);

        httpServiceTracker = new HttpServiceTracker(context, holaMundoServicioTracker);

        holaMundoServicioTracker.open();
        httpServiceTracker.open();

    }

    public void stop(BundleContext context) throws Exception {
        httpServiceTracker.close();
        holaMundoServicioTracker.close();
    }

}
```

Ilustración 30: Clase Activator de 'holaConsumidor'

En el archivo MANIFEST.MF, el cual se puede apreciar en la Ilustración 31, uno de los paquetes que hacer parte de la cabecera `Import-Package` es el de `utb.osgi.holaServicio`, como es de esperar.

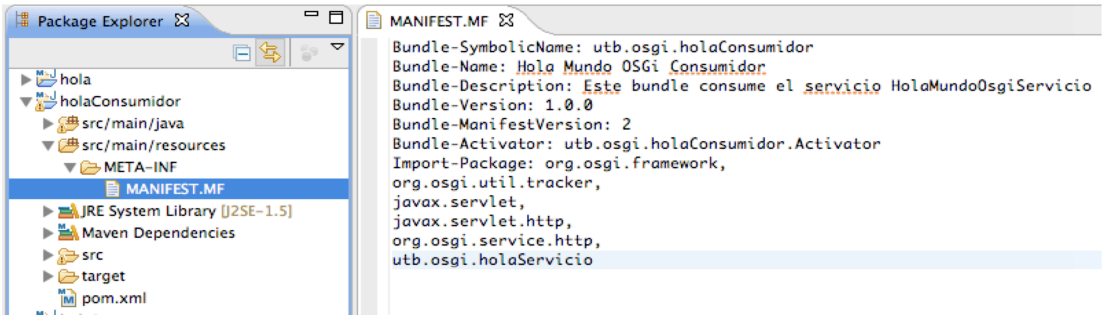


Ilustración 31: Archivo MANIFEST.MF de 'holaConsumidor'

Por último, solamente queda empaquetar el proyecto e instalarlo en el *Framework* apache Felix. Para empaquetar el proyecto, se ejecuta el comando `mvn clean package`, estando en la carpeta raíz del mismo; y para instalarlo se puede utilizar la interfaz de línea de comando y ejecutar `felix:install file:[URL o ruta]`, o bien se puede utilizar la interfaz gráfica

de *Apache Felix Web Console*. Cuando el *bundle* se ha instalado correctamente, se debe ver la frase 'Hola Mundo OSGi Servicio' en el navegador, en la ruta http://localhost:8080/holamundo_osgi_servicio.

Los *bundles* de OSGi normalmente dependen de servicios, sin importar el tamaño, los cuales publican y/o consumen. En este caso, este último ejemplo consume el servicio `HttpService` y `HolaMundoOsgiServicio`. El primero es proveído por el *bundle* 'HTTP Service Bundle', y el segundo por el *bundle* 'holaServicio'. También vale la pena anotar, que aunque se ha desarrollado aplicaciones Web, no se ha hecho uso de archivos WAR, sino solamente archivos JAR.

2.5 OSGi + MAVEN: El *bundle plug-in* para MAVEN

Código Fuente: Proyecto 'holaServicioMaven' y 'holaConsumidorMaven'

Folder con ejemplos instalados: felix-maven

Este Caso de Estudio demuestra la integración y compatibilidad de OSGi con la herramienta Maven, y como los *bundles* OSGi se benefician no solamente de las funcionalidades que brinda Maven para aplicaciones no OSGi, sino que Maven también brinda funcionalidades específicas para aplicaciones OSGi que facilitan el desarrollo de *bundles* OSGi.

La mayoría de desarrolladores considera a Maven como una herramienta de construcción o *Build Tool*, es decir, una herramienta usada para construir o empaquetar código fuente. Otros podrían referirse a Maven como una herramienta de administración de proyectos Java. La realidad es que Maven, además de brindar herramientas para la construcción o empaquetamiento, también puede generar reportes, páginas Web, y facilitar la comunicación entre desarrolladores. La definición basada en la página de Apache Maven[39] dice que Maven es una herramienta de administración de proyectos que consiste en un *Project Object Model* POM, un conjunto de estándares, un ciclo de vida del proyecto, un sistema de manejo de dependencias, y la capacidad de agregar *plug-ins*[40].

Hasta ahora se ha usado Maven básicamente para 3 aspectos. El primer aspecto es el uso que se ha hecho del archivo POM para manejar las dependencias de los proyectos Java que se han desarrollado hasta ahora, así como también ciertas características de dichos proyectos, tales como la versión, el nombre, el tipo de empaquetamiento (JAR) y la librería de Java (JVM 6). Por ejemplo, la Ilustración 32 muestra el archivo POM del proyecto 'hola', el primer Caso de Estudio que se desarrolló en este Trabajo de Grado, el cual se denominó 'Hola Mundo OSGi'.

[39] http://maven.apache.org/

[40] *Cf.* O'BRIEN, Tim, *et ál.* Maven Definitive Guide. United States of America: Sonatype, 2008, p. 1.

Ilustración 32: Archivo POM del proyecto 'hola'

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>utb.osgi</groupId>
    <artifactId>hola</artifactId>
    <version>1.0</version>
    <packaging>jar</packaging>

    <name>hola</name>
    <developers>
        <developer>
            <id>T00014525</id>
            <name>Fayder Alfonso Florez Herrera</name>
            <email>fayder.florez@gmail.com</email>
        </developer>
    </developers>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    </properties>

    <dependencies>
    <!-- For BundleActivator -->
        <dependency>
            <groupId>org.osgi</groupId>
            <artifactId>org.osgi</artifactId>
            <version>3.0.0</version>
            <scope>provided</scope>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-jar-plugin</artifactId>
                <version>2.3.1</version>
                <configuration>
                    <archive>
                        <manifestFile>
                            src/main/resources/META-INF/MANIFEST.MF
                        </manifestFile>
                    </archive>
                </configuration>
```

```
            </plugin>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <configuration>
                    <source>1.6</source>
                    <target>1.6</target>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

El segundo aspecto para destacar es que se ha usado Maven para empaquetar los proyectos en archivos JAR usando el comando `mvn clean package`. También se ha usado Maven

para instalar dependencias en el repositorio local usando el comando `mvn clean install`. Sin embargo, es el tercer aspecto el que realmente comienza a mostrar la integración entre OSGi y Maven, y se trata del manejo del archivo MANIFEST.MF. Lo que se ha venido haciendo es colocar el archivo MANIFEST.MF de cada proyecto en la ruta `src/main/resources/META-INF` respectivamente, y definir en el archivo POM de cada proyecto dicha ruta, tal como se ve en la Ilustración 32. De esta manera, cuando se empaqueta el proyecto, Maven no crea su propio archivo MANIFEST.MF sino que toma el que le se ha indicado en el archivo POM.

La definición de la ruta del archivo MANIFEST.MF dentro del archivo POM de un proyecto Maven es una pequeña muestra de la integración de OSGi y Maven. Para realmente comenzar a ver esta integración a una mayor escala, existe el llamado '*Bundle Plug-in* para Maven'[41]. Este *plug-in* está basado en una herramienta desarrollada por Peter Kriens llamada BND[42], y se utiliza para hacerla trabajar específicamente con la estructura de proyectos de Maven.

Para comenzar a tener una idea sobre el *bundle plug-in* para Maven, imagínese que se tienen 2 paquetes:

- `utb.osgi.maven.api`
- `utb.osgi.maven.impl`

También supongase que se tiene una clase `Activator`, la cual está en el paquete `utb.osgi.impl`. Como se puede ver en la Ilustración 32, todos los *plug-ins* en Maven se definen en el archivo POM, en la sección `<plugins>`. De manera que el *bundle plug-in* para Maven se vería tal como se muestra en la Ilustración 33.

[41] http://felix.apache.org/site/apache-felix-maven-bundle-plugin-bnd.html
[42] http://www.aqute.biz/Code/Bnd

```
<plugins>
    <plugin>
        <groupId>org.apache.felix</groupId>
        <artifactId>maven-bundle-plugin</artifactId>
        <extensions>true</extensions>
        <configuration>
            <instructions>
                <Export-Package>utb.osgi.maven.api</Export-Package>
                <Private-Package>utb.osgi.maven.impl</Private-Package>
                <Bundle-Activator>utb.osgi.maven.impl.Activator</Bundle-Activator>
            </instructions>
        </configuration>
    </plugin>
</plugins>
```

Ilustración 33: Ejemplo de *bundle plug-in* para Maven

Las instrucciones `<Export-Package>` y `<Private-Package>` se encargan de definir el contenido que tendrá el *bundle*. Comparando con el contenido de los archivos MANIFEST.MF que se han visto hasta ahora, se puede deducir correctamente que la instrucción `<Export-Package>` le indica al *plug-in* que paquetes se deben exportar, es decir que son públicas y pueden ser vistas y usadas por otros *bundles*. De manera opuesta, la instrucción `<Private-Package>` le dice que paquetes no deben ser públicos a otros *bundles*. Vale la pena aclarar que ambas instrucciones le dicen al *plug-in* que paquetes harán parte del *bundle* JAR. En caso de que un paquete esté declarado en ambas instrucciones, la instrucción `<Export-Package>` toma precedencia. Normalmente estas instrucciones toman más de un paquete, los cuales deben estar separados por comas.

Se puede notar en la Ilustración 33, que no se definen otras instrucciones que se han visto anteriormente en los archivos MANIFEST.MF, tales como `Bundle-SymbolicName`, el cual es una instrucción requerida por OSGi. Como se verá más adelante, el *bundle plug-in* para Maven provee valores por defecto para este tipo de instrucciones. Vale la pena subrayar, que en el contenido de la instrucción `<Import-Package>` es generado automáticamente por el *plug-in* teniendo en cuenta el contenido o dependencias del *bundle*, lo cual significa que generalmente no se tiene porque definirlo en el archivo POM.

Este *plug-in* permite usar ciertos comodines que no son permitidos en el archivo MANIFEST.MF. Un comodín es el asterisco '*', el cual se usa para indicarle al *plug-in* que tome todos los paquetes del proyecto en cuenta, para la instrucción dada. Por ejemplo si se usa `<Export-Package>*</Export-Package>` entonces todos los paquetes del *bundle* serán exportados. Otro ejemplo sería `<Private-Package>utb.osgi.maven.impl.*</Private-`

Package> y de esta manera se le indicaría al *plug-in* que todo el contenido del paquete utb.osgi.maven.impl (incluyendo sub-paquetes) deben ser considerados privados y por lo tanto no deben ser exportados.

El otro comodín que se puede usar es el signo de exclamación '!', el cual le indica al *plug-in* excluir de la instrucción los paquetes que comiencen con este comodín. Por ejemplo, la instrucción <Export-Package>!utb.osgi.maven.impl</Export-Package> significa que este paquete debe ser excluido del conjunto de paquetes a ser exportados.

Los paquetes que se declaren dentro de las instrucciones, poseen cierto orden si se hace uso de estos comodines. Por ejemplo si se usa la siguiente instrucción <Export-Package>utb.osgi.maven.*, !utb.osgi.maven.impl</Export-Package>, la segunda parte de la instrucción queda sin efecto debido a que la primera parte indica que todo el contenido dentro del paquete utb.osgi.maven debe ser incluido en esta instrucción. Entonces, para obtener el efecto deseado según la anterior instrucción, las partes se deben invertir, quedando de la siguiente manera <Export-Package>!utb.osgi.maven.impl, utb.osgi.maven.*</Export-Package>.

Como se indicó anteriormente, la mayoría de instrucciones poseen valores por defecto, por lo que a veces, solo basta con definir los paquetes a exportar, los paquetes privados y un *Activator* de haberlo, y el resto de instrucciones tomarían valores por defecto. A continuación se lista algunas de las instrucciones que tienen valores por defecto.

- <Bundle-SymbolicName>: Para generar el valor por defecto para esta instrucción, el *plug-in* toma el valor de las etiquetas <groupId> y <artifactId>, definidas en el archivo POM, y las usa de la siguiente forma: <groupId> + "." + <artifactId>. Si regresamos a la Ilustración 32, el valor por defecto de esta instrucción para ese proyecto sería utb.osgi.hola.
- <Export-Package>: Incluye todos los paquetes en el código fuente, excepto aquellos que tengan la palabra 'impl' o 'internal'.

- `<Private-Package>`: Incluye todos los paquetes en el código fuente.
- `<Import-Package>`: Contiene las referencias o dependencias del *bundle* que no se encuentran en el *bundle*. Todos los paquetes que son exportados, también son importados con el propósito de evitar inconsistencias en el espacio de clases.
- `<Include-Resource>`: Incluye los recursos que existan en el folder fuente `src/main/resources`.
- `<Bundle-Version>`: Se toma de la propiedad '`${pom.version}`', es decir de la etiqueta `<version>` en el archivo POM. Sin embargo, este valor es transformado para que sea compatible con el formato de versión de OSGi, el cual es '*Major.Minor.Micro.Qualifier*'. Por ejemplo, la versión 1.0, sería 1.0.0; y la versión 1.3-SNAPSHOT sería 1.3.0.SNAPSHOT.
- `<Bundle-Name>`: Se toma de la propiedad '`${pom.name}`', es decir de la etiqueta `<name>` en el archivo POM.
- `<Bundle-Description>`: Se toma de la propiedad '`${pom.description}`'.
- `<Bundle-License>`: Se toma de la propiedad '`${pom.licenses}`'.
- `<Bundle-Vendor>`: Se toma de la propiedad '`${pom.organization.name}`'.
- `<Bundle-DocURL>`: Se toma de la propiedad '`${pom.organization.url}`'.

El valor para la cabecera OSGi `Bundle-ManifestVersion` es fijo para el *plug-in*, y tiene el valor de '2', que se refiere a la última versión a la fecha de la especificación OSGi, la R4. Las instrucciones descritas anteriormente no son las únicas que acepta el *plug-in*, pero si son las más usadas.

A continuación se realizará un ejemplo para ver en acción el *bundle plug-in* para Maven. Se tomará como base los ejemplos 'holaServicio' y 'holaConsumidor' que se desarrollaron anteriormente para crear dos nuevos ejemplos que se llamarán 'holaServicioMaven' y 'holaConsumidorMaven', los cuales implementarán el *plug-in*. Lo primero que se hará será modificar el archivo POM de 'holaServicio' y agregar el *plug-in*. La información sobre los paquetes privados y a exportar se tomará del archivo MANIFEST.MF que existe en el proyecto 'holaServicio'. Al final, el archivo POM de 'holaServicioMaven' luce como se

muestra en la Ilustración 34.

Ilustración 34: Archivo POM del proyecto 'holaServicioMaven'

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>utb.osgi</groupId>
    <artifactId>holaServicioMaven</artifactId>
    <version>1.0</version>
    <packaging>bundle</packaging>

    <name>holaServicioMaven</name>
    <developers>
        <developer>
            <id>T00014525</id>
            <name>Fayder Alfonso Florez Herrera</name>
            <email>fayder.florez@gmail.com</email>
        </developer>
    </developers>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    </properties>

    <dependencies>
        <!-- For BundleActivator -->
        <dependency>
            <groupId>org.osgi</groupId>
            <artifactId>org.osgi</artifactId>
            <version>3.0.0</version>
            <scope>provided</scope>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <!--plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-jar-plugin</artifactId>
                <version>2.3.1</version>
                <configuration>
                    <archive>
                        <manifestFile>
                            src/main/resources/META-INF/MANIFEST.MF
                        </manifestFile>
                    </archive>
                </configuration>

            </plugin-->
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <configuration>
                    <source>1.6</source>
                    <target>1.6</target>
                </configuration>
            </plugin>
            <plugin>
                <groupId>org.apache.felix</groupId>
                <artifactId>maven-bundle-plugin</artifactId>
                <extensions>true</extensions>
                <configuration>
                    <instructions>
                        <Export-Package>utb.osgi.holaServicioMaven</Export-Package>
                        <Private-Package>utb.osgi.holaServicioMaven.impl.*</Private-Package>
                        <Bundle-Activator>utb.osgi.holaServicioMaven.impl.Activator</Bundle-Activator>
                        <Bundle-Name>Ejemplo Hola Mundo OSGi Servicio con Maven</Bundle-Name>
                        <Bundle-Description>Ejemplo Hola Mundo OSGi como Servicio con Maven</Bundle-Description>
                    </instructions>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

Lo primero que se puede notar es que el tipo de empaquetamiento del proyecto ya no es `jar` como en los proyectos anteriores, sino que ahora es `bundle`; este tipo de empaquetamiento lo provee el *bundle plug-in* para Maven. Sin embargo esta diferencia no cambia la manera en que se ha venido empaquetando los *bundles*, todavía se seguirá usando el comando `mvn clean package`.

El otro aspecto a destacar es que el *plug-in* `maven-jar-plugin`, y su configuración en donde se indica la ubicación del archivo MANIFEST.MF, se encuentra comentado. Esto se debe a que este *plug-in* ya no es necesario, ya que el contenido del archivo MANIFEST.MF ahora será reemplazado por las instrucciones del *bundle plug-in* para Maven.

Por último, se tiene el *bundle plug-in* para Maven como tal, y en su configuración se puede ver las instrucciones que servirán para crear el archivo MANIFEST.MF de este *bundle*. La estructura del proyecto 'holaServicioMaven' prácticamente es la misma que la del anterior proyecto 'holaServicio', y las clases son las mismas. Siendo así, solo queda empaquetar e instalar el *bundle* en el *Framework* Apache Felix para ver si el *bundle* no tiene problemas. Después de instalar el *bundle* 'holaServicioMaven' en el *Framework* se debe apreciar ver algo similar a la Ilustración 35, la cual muestra la información del *bundle*.

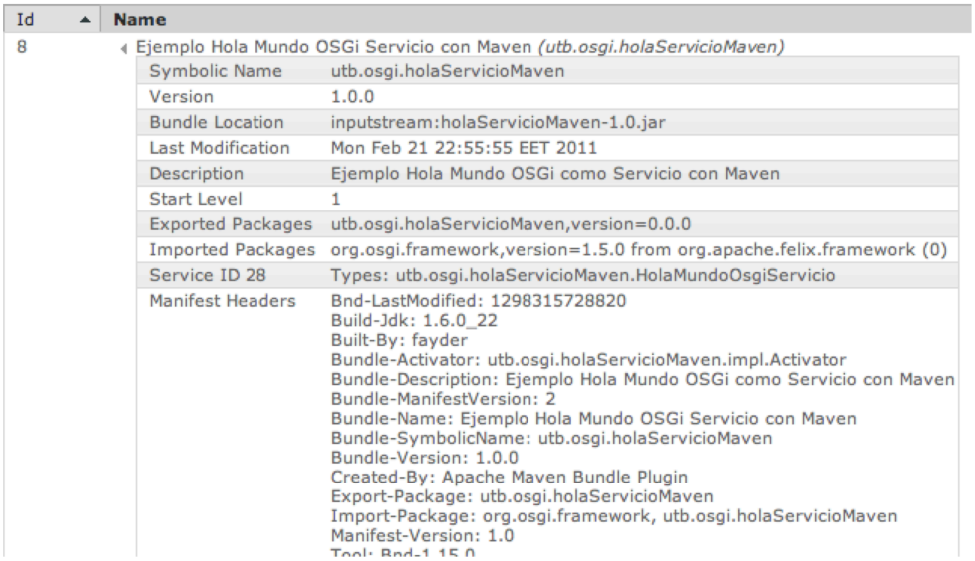


Ilustración 35: *Bundle* 'holaServicioMaven' en *Apache Felix Web Console*

En la Ilustración 35 se puede observar que el servicio 'HolaMundoOsgiServicio' ha sido registrado en el Registro de Servicios del *Framework* Apache Felix, y además el estado del *bundle* es *Active*, lo cual indica que el *bundle* está funcionando como era esperado. También se puede observar que las cabeceras del MANIFEST concuerdan con las instrucciones que definimos en el archivo POM, y vale la pena subrayar que el paquete que se definió para ser exportado, `utb.osgi.holaServicioMaven`, también está siendo importado; como se explicó anteriormente, el *bundle plug-in* para Maven realiza esto para evitar inconsistencias en el espacio de clases.

La segunda parte de este ejercicio para demostrar la funcionalidad básica del *bundle plug-in* para Maven consiste en desarrollar el proyecto 'holaConsumidorMaven', el cual estará basado en el proyecto 'holaConsumidor', con la diferencia de que 'holaConsumidorMaven' tendrá como dependencia al proyecto 'holaServicioMaven' para así poder consumir el servicio prestado por este *bundle*. El *bundle* 'holaConsumidorMaven' solamente tiene clases en el paquete `utb.osgi.holaConsumidorMaven`, por lo tanto tiene sentido que se defina este paquete como privado, ya que este *bundle* no tiene nada para proveer a otros *bundles*. Como se puede ver en la Ilustración 36, para la instrucción `<Export-Package>` hemos usado los comodines para indicarle al *plug-in* que ningún paquete debe ser exportado.

```
<plugin>
    <groupId>org.apache.felix</groupId>
    <artifactId>maven-bundle-plugin</artifactId>
    <extensions>true</extensions>
    <configuration>
        <instructions>
            <Export-Package>!*</Export-Package>
            <Private-Package>utb.osgi.holaConsumidorMaven.*</Private-Package>
            <Bundle-Activator>utb.osgi.holaConsumidorMaven.Activator</Bundle-Activator>
            <Bundle-Name>Ejemplo OSGi Consumidor de Servicio con Maven</Bundle-Name>
            <Bundle-Description>Ejemplo OSGi Consumidor de Servicio con Maven</Bundle-Description>
        </instructions>
    </configuration>
</plugin>
```

Ilustración 36: *Bundle plug-in* para Maven de 'holaConsumidorMaven'

El servlet que tiene el *bundle* 'holaConsumidorMaven' lo se ha registrado con la URL: http://localhost:8080/holamundo_osgi_servicio_maven (hay que recordar que el *bundle* 'holaConsumidor' tiene su servlet registrado en la URL http:localhost:8080/holamundo_osgi_servicio). Una vez empaquetado e instalado el *bundle*

'holaConsumidorMaven' en el *Framework* Apache Felix, si se dirige el navegador hacía la URL previamente mencionada se debe observar ver la frase 'Hola Mundo OSGi Servicio con Maven!'. La Ilustración 37 muestra la información del *bundle* 'holaConsumidor' en *Apache Felix Web Console*.

Id	Name	
9	Ejemplo OSGi Consumidor de Servicio con Maven *(utb.osgi.holaConsumidorMaven)*	
	Symbolic Name	utb.osgi.holaConsumidorMaven
	Version	1.0.0
	Bundle Location	inputstream:holaConsumidorMaven-1.0.jar
	Last Modification	Wed Feb 23 20:31:29 EET 2011
	Description	Ejemplo OSGi Consumidor de Servicio con Maven
	Start Level	1
	Exported Packages	---
	Imported Packages	javax.servlet,version=2.5.0 from org.apache.felix.http.bundle (5) javax.servlet.http,version=2.5.0 from org.apache.felix.http.bundle (5) org.osgi.framework,version=1.5.0 from org.apache.felix.framework (0) org.osgi.service.http,version=1.2.0 from org.apache.felix.http.bundle (5) org.osgi.util.tracker,version=1.4.0 from org.apache.felix.framework (0) utb.osgi.holaServicioMaven,version=0.0.0 from utb.osgi.holaServicioMaven (8)
	Manifest Headers	Bnd-LastModified: 1298473657563 Build-Jdk: 1.6.0_22 Built-By: fayder Bundle-Activator: utb.osgi.holaConsumidorMaven.Activator Bundle-Description: Ejemplo OSGi Consumidor de Servicio con Maven Bundle-ManifestVersion: 2 Bundle-Name: Ejemplo OSGi Consumidor de Servicio con Maven Bundle-SymbolicName: utb.osgi.holaConsumidorMaven Bundle-Version: 1.0.0 Created-By: Apache Maven Bundle Plugin Import-Package: javax.servlet, javax.servlet.http, org.osgi.framework, org.osgi.service.http, org.osgi.util.tracker, utb.osgi.holaServicioMaven Manifest-Version: 1.0 Tool: Bnd-1.15.0

Ilustración 37: *Bundle* 'holaConsumidorMaven' en *Apache Felix Web Console*

El *Bundle Plug-in* para Maven facilita mucho el empaquetamiento de *bundles* y el manejo de sus dependencias, así como también la definición de las cabeceras o instrucciones MANIFEST. Por lo tanto, OSGi no es solamente compatible con Maven, sino que es mucho más fácil de manejar. Vale la pena aclarar que lo que se ha visto en este Trabajo de Grado sobre la integración de Maven y OSGi es apenas un vistazo a todo lo que es posible hacer con estas dos tecnologías juntas.

2.6 OSGi + SPRING: SPRING-DM

Código Fuente: Proyectos 'spring-person-model' y 'spring-person-view'
Folder con ejemplos instalados: felix-spring

Este Caso de Estudio demuestra la integración y compatiblidad de OSGi con el *framework* Spring-DM. A través del proyecto SPRING-DM, el *framework* Spring provee muchas de las capacidades y servicios disponibles para aplicaciones no OSGi. La característica principal de Spring, que se demuestra en este Caso de Estudio es la Inversión de Control.

Spring[43] es un *Framework* para Java que surgió en respuesta a una serie de desventajas que presentaba la tecnología J2EE (*Java 2 Enterprise Edition*), conocida hoy en día solo como Java EE, y básicamente presentó una alternativa a *Enterprise Java Beans* (EJB). Dos de sus características más importantes son la Inversión de Control y la Inyección de Dependencias.

La Inversión de Control, conocido como IoC por sus siglas en inglés, es un patrón de diseño en el cual el flujo de un sistema es invertido con respecto al sistema convencional de procedimientos; este flujo es delegado a una entidad externa, la cual ejecuta una serie de llamadas basadas en unas instrucciones pre-definidas. En otras palabras, es una implementación del principio de POO llamado 'Principio de Hollywood', en donde los servicios de una clase o sistema no invocan, sino que son invocados. En el caso de Spring, resulta ser ella misma aquella entidad externa que ejecuta la Inversión de Control[44].

La Inyección de Dependencias es el tipo de Inversión de Control que implementa Spring, el cual consiste en suministrar objetos a una clase dada, en lugar de ser la propia clase quien se responsabilice por la creación de tales objetos. Por lo tanto, los objetos son inicializados o inyectados por Spring solamente cuando sea necesario. La Inversión de Control le permite a Spring separar la Lógica del Negocio de los servicios y recursos que este necesite, favoreciendo la simplicidad, el mantenimiento la realización o ejecución de pruebas y la re-usabilidad.

[43] http://www.springsource.org/
[44] *Cf.* RUBIO, Daniel. Pro Spring Dynamic Modules for OSGi Service Platforms. United States of America: Apress, 2009, p. 45.

Otra característica importante a resaltar sobre Spring, es el uso de los llamados *Plain Old Java Objects* o POJOs. Un POJO es un objeto instanciado desde una clase y el cual consiste en usa serie de propiedades con sus respectivos *getters* y *setters*. Algunas veces un POJO incluye una cierta lógica en donde se manipulan sus propias propiedades. Los POJO son una parte importante para garantizar la simplicidad en el *Framework Spring*.

Se puede decir que la integración de OSGi y Spring, hace que el *Framework* OSGi se presente como una alternativa viable para el desarrollo de aplicaciones empresariales. Pero también es un hecho que OSGi beneficia al *Framework* Spring ya que la técnica de Inversión de Control puede ser realizada de una manera más inteligente debido a las capacidades de carga dinámica que tiene OSGi. Incluso, los *beans* de Spring pueden ser registrados como servicios en OSGi, haciéndolos disponibles a otros *bundles* en OSGi.

Como se ha visto hasta ahora, la arquitectura de OSGi consiste principalmente en el manejo de servicios, los cuales son registrados o consumidos a través del Registro de Servicios de OSGi, y esta arquitectura la que le permite a OSGi tener un comportamiento dinámico. La manera en la que estos servicios son manejados, es a través de una serie de procedimientos proveídos por la API de OSGi (por ejemplo a través de las clases `Activator` y `HttpServiceTracker`). En este caso, Spring puede proveer su técnica de Inyección de Dependencias para no tener que usar tanto código para el manejo de servicios en OSGi, haciendo de este proceso algo más fácil de realizar.

La mayoría de *beans* en una aplicación que utiliza el *Framework* Spring, son declarados en el tradicional *Classpath* de Java, y este proceso puede conllevar a dependencias no resueltas y conflictos entre clases. Además, si se tienen *beans* en diferentes archivos JAR, estos estarían completamente desconectados e inconscientes los unos de los otros. En este caso OSGi provee su Registro de Servicios para resolver estas limitaciones del *Framework* Spring.

Sin embargo, para que OSGi y Spring puedan actuar juntos, es necesario que existan ciertos

cambios en la estructura de Spring, dándole un sentido a sus *beans* hacía los servicios OSGi, de manera de que los *beans* de Spring soporten las capacidades brindadas por OSGi. Esta nueva 'dirección' o enfoque de Spring se ha llamado *Spring Dynamic Modules for OSGi*[45] o sencillamente Spring-DM. Para que Spring-DM pueda proveer las capacidades del *Framework* Spring en el ambiente OSGi, la gente de Spring ha 'OSGi-ficado' varias de sus librerías[46], es decir que les ha añadido un archivo MANIFEST, para convertirlas en *bundles* de OSGi. En resumidas cuentas, Spring-DM nos libra de usar la API de OSGi específicamente para manejar servicios OSGi y además brinda todas las capacidades de Spring a los *bundles* OSGi. Spring ofrece una guía completa sobre Spring-DM versión 1.2.1 en la siguiente dirección: http://static.springsource.org/osgi/docs/1.2.1/reference/html/.

La mejor manera de obtener una idea de cómo OSGi y Spring trabajan juntos es a través de un ejemplo, y por eso a continuación se desarrollará un ejemplo que permita ver de mejor forma la interacción de estas dos tecnologías. Lo primero es descargar Spring-DM, el cual viene en un archivo .zip y se puede descargar de: http://www.springsource.com/download/community; el nombre exacto del archivo a descargar es `spring-osgi-2.0.0.M1-with-dependencies.zip`. Este zip contiene una serie de *bundles* que proveen ciertos servicios y dependencias necesarios para que el *Framework* Spring, y *bundles* desarrollados con Spring, puedan funcionar en el *Framework* OSGi.

Una vez se ha descomprimido el zip de Spring-DM, se verá una carpeta llamada 'lib', la cual contiene el primer conjunto de *bundles* que se van a necesitar para este ejemplo. La lista de *bundles* que vamos a necesitar es la siguiente:

- `com.springsource.org.aopalliance-1.0.0`
- `com.springsource.org.apache.log4j-1.2.15`
- `com.springsource.slf4j.api-1.5.6`
- `com.springsource.slf4j.log4j-1.5.6`

[45] http://www.springsource.org/osgi
[46] http://ebr.springsource.com/repository/app/

- com.springsource.slf4j.org.apache.commons.logging-1.5.6
- org.springframework.aop-3.0.0.RC1
- org.springframework.asm-3.0.0.RC1
- org.springframework.beans-3.0.0.RC1
- org.springframework.context-3.0.0.RC1
- org.springframework.core-3.0.0.RC1
- org.springframework.expression-3.0.0.RC1

El segundo conjunto de *bundles* de Spring-DM que necesitaremos se encuentra en la carpeta 'extensions', y son los siguientes:

- spring-osgi-core-2.0.0.M1
- spring-osgi-extender-2.0.0.M1
- spring-osgi-io-2.0.0.M1

Una vez estos *bundles* han sido instalados en el *Framework* Apache Felix, se debe ver en el *Apache Felix Web Console*, una lista de *bundles* parecida a la mostrada en la Ilustración 38. Aunque la Ilustración 38 no lo muestra, el *bundle* 'SLF4J Log4J Binding', es en realidad un 'Fragmento'. Un Fragmento o *Fragment*[47] es un *bundle* que puede ser adjuntado a uno o más *bundles* huéspedes, es decir que un *bundle* que sea huésped de un Fragmento será tratado por el *Framework* OSGi como un solo *bundle*. Un ejemplo de un Fragmento puede ser un *bundle* que provea archivos o recursos para la traducción de una aplicación a diferentes idiomas. De esta manera se tendría un *bundle* para las traducciones, separado de la aplicación que lo utilice, pero que durante su ejecución ambos *bundles*, huésped y Fragmento, serían tratados como uno solo. En el caso del Fragmento 'SLF4J Log4J Binding', el *bundle* 'SLF4J API' actúa como su huésped, en el conjunto de *bundles* mostrados en la Ilustración 38.

[47] OSGI Alliance, The. OSGi Service Platform Core Specification. Release 4, Version 4.2, 2009, p. 74

Id	Name
0	System Bundle *(org.apache.felix.framework)*
1	Apache Felix Bundle Repository *(org.apache.felix.bundlerepository)*
2	Apache Felix Gogo Command *(org.apache.felix.gogo.command)*
3	Apache Felix Gogo Runtime *(org.apache.felix.gogo.runtime)*
4	Apache Felix Gogo Shell *(org.apache.felix.gogo.shell)*
5	Apache Felix Http Bundle *(org.apache.felix.http.bundle)*
6	Apache Felix Http Jetty *(org.apache.felix.http.jetty)*
7	Apache Felix Web Management Console *(org.apache.felix.webconsole)*
8	AOP Alliance API *(com.springsource.org.aopalliance)*
9	Apache Log4J *(com.springsource.org.apache.log4j)*
10	SLF4J API *(com.springsource.slf4j.api)*
11	SLF4J Jakarta Commons Logging Over SLF4J Binding *(com.springsource.slf4j.org.apache.commons.logging)*
12	SLF4J Log4J Binding *(com.springsource.slf4j.log4j)*
13	Spring Beans *(org.springframework.beans)*
14	Spring AOP *(org.springframework.aop)*
15	Spring Core *(org.springframework.core)*
16	Spring Context *(org.springframework.context)*
17	spring-osgi-core *(org.springframework.osgi.core)*
18	spring-osgi-extender *(org.springframework.osgi.extender)*
19	spring-osgi-io *(org.springframework.osgi.io)*

Ilustración 38: Spring-DM *bundles* en Apache Felix Web Console

Es una desventaja que los *bundles* de Spring-DM no estén disponibles *online* en algún repositorio, y que haya que instalarlos en el repositorio local de Maven de ser necesario.

Para seguir con el Caso de Estudio, el cual se irá desvelando poco a poco, el siguiente paso es desarrollar una clase que hace parte del modelo. La clase `Person` que se muestra en la Ilustración 40 es un simple POJO, ya que solo posee atributos, con sus respectivos *getters* y *setters*, y además una sobre carga del método equals(), para indicar que 2 personas son iguales cuando su `id` es el mismo. Este tipo de clases también son llamadas clases 'Entidad', ya que se asemejan o son análogas a las entidades en una base de datos relacional.

La capa de acceso de este ejemplo, permitirá a otros *bundles* manejar información sobre la clase `Person`, es decir: seleccionar una o todas, insertar, eliminar y actualizar. Para esto, se proveerá una interfaz con los servicios necesarios para realizar estas acciones. La interfaz `PersonDAO`, que se muestra en la Ilustración 41, brindará acceso a estos servicios, y de la

misma forma la Ilustración 42 muestra la implementación de esta interfaz, llamada `PersonDAOImpl`. Es claro que el *bundle* de ejemplo exportará la interfaz, y mantendrá como privada la implementación. En la Ilustración 43 se puede apreciar que la clase `PersonDAOImpl` posee una referencia a otra clase llamada `PersonDataSource`, a la cual delega todas sus funciones. Esta clase `PersonDataSource` funcionará como la falsa conexión a una base de datos, y digo falsa porque como se ve en la Ilustración 42, la clase `PersonDataSource` en realidad no se conecta a una base de datos sino que provee datos en memoria. Esto lo se hace con el propósito de mantener simplicidad en el ejercicio de demostrar la integración entre OSGi y Spring. Vale la pena destacar que Spring ofrece una API para manejo de base de datos muy poderosa y, por supuesto, esta API se encuentra disponible para OSGi también.

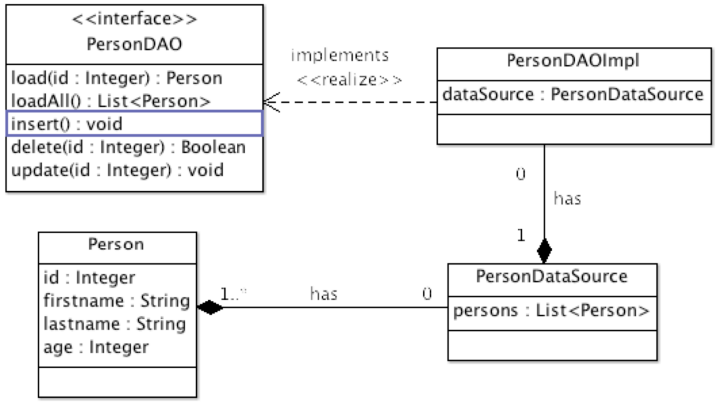


Ilustración 39: Diagrama de clases Caso de Estudio OSGi + Spring-DM

Eso es todo con respecto al código Java que se tendrá en este Caso de Estudio, ahora se proseguirá con la parte donde Spring entra en acción. Se creará un archivo llamado `applicationContext.xml`, en la ruta `src/main/resources/META-INF/spring/`. En este archivo se definen las clases `PersonDAOImpl` y `PersonDataSource` como Spring *beans*, y además se inyecta el *bean* de `PersonDataSource`, en el *bean* de `PersonDAOImpl`. La clase `PersonDAOImpl` posee una referencia a `PersonDataSource` la cual es inicializada en el constructor de la clase `PersonDAOImpl`; pues bien, Spring se encarga de inyectar esta dependencia en la clase `PersonDAOImpl`, en el momento en que el *bean* de `PersonDAOImpl` sea inyectado a otro *bean*. Esta es la técnica de Inyección de Dependencias de Spring en acción, la cual puede parecer algo confusa al principio, pero con este ejemplo se logrará

aclarar esta técnica. La Ilustración 39 muestra un Diagrama de Clases UML para ayudar a entender las relaciones entre estas clases y, por su parte, la Ilustración 44 muestra el contenido del archivo `applicationContext.xml`.

```java
package utb.osgi.spring.person.model;

public class Person {

    private Integer id;
    private String firstname;
    private String lastname;
    private Integer age;

    public Person(Integer id, String firstname, String lastname, Integer age) {
        this.id = id;
        this.firstname = firstname;
        this.lastname = lastname;
        this.age = age;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    public Integer getId() {
        return id;
    }

    public String getFirstname() {
        return firstname;
    }

    public void setFirstname(String firstname) {
        this.firstname = firstname;
    }

    public String getLastname() {
        return lastname;
    }
    public void setLastname(String lastname) {
        this.lastname = lastname;
    }

    public Integer getAge() {
        return age;
    }

    public void setAge(Integer age) {
        this.age = age;
    }

    @Override
    public boolean equals(Object obj) {
        if (this.getId().equals(((Person) obj).getId())) {
            return true;
        }

        return false;
    }

}
```

Ilustración 40: Clase Person de 'spring-person-model'

```java
package utb.osgi.spring.person.data;

import java.util.List;

public interface PersonDAO {

    Person load(Integer id);

    List<Person> loadAll();

    void insert(Person person);

    Boolean delete(Person person);

    void update(Person person);

}
```

Ilustración 41: Clase PersonDAO de 'spring-person-model'

```java
package utb.osgi.spring.person.data.impl;

import java.util.List;

public class PersonDAOImpl implements PersonDAO{

    private PersonDataSource dataSource;

    public PersonDAOImpl(PersonDataSource dataSource) {
        this.dataSource = dataSource;
    }

    public Person load(Integer id) {
        return dataSource.load(id);
    }

    public List<Person> loadAll() {
        return dataSource.loadAll();
    }

    public void insert(Person person) {
        dataSource.insert(person);
    }

    public Boolean delete(Person person) {
        return dataSource.delete(person);
    }

    public void update(Person person) {
        dataSource.update(person);
    }

}
```

Ilustración 42: Clase PersonDAOImpl de 'spring-person-model'

```java
package utb.osgi.spring.person.data.impl;

import java.util.ArrayList;
import java.util.List;

import utb.osgi.spring.person.model.Person;

public class PersonDataSource {

    private List<Person> persons;

    public PersonDataSource() {

        Person personOne = new Person(1, "John", "Rambo", 34);
        Person personTwo = new Person(2, "Rocky", "Balboa", 32);
        Person personThree = new Person(3, "Joseph", "Dredd", 40);

        persons = new ArrayList<Person>();
        persons.add(personOne);
        persons.add(personTwo);
        persons.add(personThree);
    }

    public Person load(Integer id) {

        for (Person person : persons) {
            if (person.getId().equals(id)) {
                return person;
            }
        }

        return null;
    }

    public List<Person> loadAll() {
        return persons;
    }

    public void insert(Person person) {
        if (person != null && load(person.getId()) == null) {
            persons.add(person);
        }
    }

    public Boolean delete(Person person) {
        return persons.remove(person);
    }

    public void update(Person person) {
        if (delete(person)) {
            persons.add(person);
        }
    }

}
```

Ilustración 43: Clase PersonDataSource de 'spring-person-model'

applicationContext.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

    <bean id="dataSource" class="utb.osgi.spring.person.data.impl.PersonDataSource" />

    <bean id="personDAO" class="utb.osgi.spring.person.data.impl.PersonDAOImpl">
        <constructor-arg ref="dataSource" />
    </bean>

</beans>
```

Ilustración 44: Archivo applicationContext.xml de 'spring-person-model'

El anterior ejemplo llamado 'holaServicio', posee una clase `Activator` la cual se encarga de registrar el servicio brindado por este *bundle* en el Registro de Servicios de Apache Felix. En esta ocasión el *Framework* Spring toma esta responsabilidad, con lo cual se reemplaza el código Java por XML. El archivo XML que publicará el servicio estará en la ruta `src/main/resources/META-INF/spring/` y tendrá por nombre `osgiContext.xml`. Como se muestra en la Ilustración 45, este archivo posee un solo elemento `<osgi:service>`, el cual es usado para exportar o publicar Spring *beans* como Servicios OSGi. Este elemento debe tener dos atributos como mínimo: `id`, como todo Spring *bean* para ser identificado de manera única, e `interface`, el cual contiene la interfaz del servicio que queremos exportar.

osgiContext.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:osgi="http://www.springframework.org/schema/osgi"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
        http://www.springframework.org/schema/osgi http://www.springframework.org/schema/osgi/spring-osgi.xsd">

    <osgi:service id="personDAOService" ref="personDAO"
        interface="utb.osgi.spring.person.data.PersonDAO">
    </osgi:service>

</beans>
```

Ilustración 45: Archivo osgiContext.xml de 'spring-person-model'

Uno de los *bundles* que hace parte de Spring-DM es `spring-osgi-extender`, también llamado el '*Spring bundle extender*', tal cual lo se vió anteriormente en la lista de *bundles* que se instalaron en el *Framework* OSGi. Lo que hace este *bundle*, en pocas palabras, es buscar en el *Framework* OSGi aquellos *bundles* que estén desarrollados con el *Framework*

Spring. El *bundle extender* de Spring considera que un *bundle* está desarrollado con tecnología OSGi si este posee la cabecera `Spring-Context` en su archivo MANIFEST o si en la ruta `META-INF/spring` existen archivos XML, el cual es el caso para el presente Caso de Estudio. Una vez el *bundle extender* de Spring encuentra un Spring *bundle*, este carga los archivos de configuración de Spring que tenga el *bundle* para crear lo que se llama 'El Contexto de la Aplicación' o *Application Context*. El *bundle extender* también revisa si el Spring *bundle* exporta o importa algún servicio OSGi, de ser así registra el servicio en el Registro de Servicios del *Framework* en el caso de exportar, y encuentra el servicio que el Spring *bundle* necesite y lo agrega a su *Application Context* en el caso de importar[48].

La razón por la cual se tienen dos archivos XML en la ruta `META-INF/spring`, es simplemente una buena práctica, en donde los Spring *beans* que son solamente de la aplicación se definen en un archivo (i.e. `applicationContext.xml`) y los Spring *beans* OSGi se definen en otro archivo XML diferente (i.e. `osgiContext.xml`).

```
<plugin>
    <groupId>org.apache.felix</groupId>
    <artifactId>maven-bundle-plugin</artifactId>
    <extensions>true</extensions>
    <configuration>
        <instructions>
            <Export-Package>
                utb.osgi.spring.person.data,
                utb.osgi.spring.person.model
            </Export-Package>
            <Private-Package>
                utb.osgi.spring.person.data.impl.*
            </Private-Package>
            <Bundle-Name>Ejemplo OSGi + Spring (model)</Bundle-Name>
        </instructions>
    </configuration>
</plugin>
```

Ilustración 46: Bundle Plug-in para Maven de 'spring-person-model'

Por supuesto, el ejemplo 'spring-person-model' debe tener un archivo MANIFEST para que pueda ser un *bundle* OSGi, y este será generado usando el *Bundle Plug-in* para Maven que se dio a conocer anteriormente. En el archivo POM se denifen las cabeceras del archivo MANIFEST, tal como se muestra en la Ilustración 46.

Como se puede ver en la Ilustración 46, nuestro *bundle* 'spring-person-model' exporta

[48] *Cf.* WALLS, Craig. Modular Java – Creating Flexible Applications with OSGi and Spring. United States of America, 2009, p. 113.

solamente dos paquetes: `utb.osgi.spring.person.data` y `utb.osgi.spring.person.model`, los cuales tienen la interfaz del servicio que se quieren exportar: `PersonDAO` y su entidad o POJO: `Person`. El paquete que guarda la implementación, o lógica del negocio, se mantiene como privado.

La segunda parte del Caso de Estudio es, obviamente, desarrollar el *bundle* que consuma o haga uso de los servicios que provee el *bundle* 'spring-person-model'. A este nuevo *bundle* se le denominará 'spring-person-view'. Como ya se ha hecho en casos anteriores, es buena idea instalar el *bundle* 'spring-person-model' en el repositorio Maven local, ya que se usará su API en el *bundle* 'spring-person-view'. Esto se hace ejecutando el comando `mvn clean install`, estando en la carpeta raíz del *bundle* 'spring-person-model'.

En el *bundle* 'spring-person-model' se crea un archivo XML llamado `osgiContext.xml`, el cual se encarga de publicar el Spring *bean* `PersonDAO` como un Servicio OSGi. En el caso del *bundle* 'spring-person-view' se tendrá el mismo archivo pero en esta ocasión será para obtener el servicio `PersonDAO` del Registro de Servicios de OSGi, tal como se ve en la Ilustración 47.

osgiContext.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:osgi="http://www.springframework.org/schema/osgi"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
    http://www.springframework.org/schema/osgi http://www.springframework.org/schema/osgi/spring-osgi.xsd">

    <osgi:reference id="personDAOService" interface="utb.osgi.spring.person.data.PersonDAO"/>

</beans>
```

Ilustración 47: Archivo osgiContext.xml de 'spring-person-view'

El elemento `<osgi:reference>` posee dos atributos, el atributo `id` es usado por Spring-DM para convertir y agregar al *bundle* un Servicio OSGi como Spring *bean*. El atributo `interface` es usado por Spring-DM para encontrar un servicio que esté relacionado con esa interfaz, que en este caso es `PersonDAO`.

```java
package utb.osgi.spring.person.view;

import java.util.List;

import utb.osgi.spring.person.data.PersonDAO;
import utb.osgi.spring.person.model.Person;

/**
 * Obtiene una referencia al servicio OSGi PersonDAO a través del Spring IoC
 * @author Fayder Florez
 */
public class PersonConsole {

    private PersonDAO personDAO;

    public PersonConsole(){}

    public void init(){}

    // Called by Spring
    public void setPersonDAO(PersonDAO personDAO) {
        System.out.println("Spring Injecting PersonDAO");
        this.personDAO = personDAO;

        loadAllPersons();
    }

    private void loadAllPersons() {
        System.out.println(":: All Persons ::");
        List<Person> persons = personDAO.loadAll();
        for (Person person : persons) {
            System.out.println("ID: " + person.getId());
            System.out.println("Firstname: " + person.getFirstname());
            System.out.println("Lastname: " + person.getLastname());
            System.out.println("Age: " + person.getAge());
            System.out.println();
        }
    }
}
```

Ilustración 48: Clase PersonConsole de 'spring-person-view'

En la Ilustración 48 se puede apreciar la clase `PersonConsole` la cual tiene la principal función de imprimir en consola la lista de todos los objetos `Person` disponibles a través de la interfaz `PersonDAO`. El *Framework* Spring es el responsable de inyectar una instancia de una implementación de la interfaz `PersonDAO` en la clase `PersonConsole`, y Spring lo hace a través del método `setPerson()`. Lo que en realidad ocurre es que la clase `PersonConsole` es inicializada en la clase `Activator`, como se verá más adelante, y luego de ser inicializada Spring-DM se da cuenta de que existe un Spring *bean* que hace referencia a la clase `PersonConsole`, y que esta clase espera una inyección de la interfaz `PersonDAO`. La razón por la cual Spring-DM considera a la clase `PersonConsole` como un Spring *bean*, y que además se le debe inyectar una instancia de `PersonDAO` se puede apreciar en la Ilustración 49.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

    <bean id="personConsole" class="utb.osgi.spring.person.view.PersonConsole">
        <property name="personDAO" ref="personDAOService"></property>
    </bean>

</beans>
```

Ilustración 49: Archivo applicationContext.xml de 'spring-person-view'

Como se explicó anteriormente, la clase `Activator` se encarga de inicializar la clase `PersonConsole` cuando el *bundle* es iniciado, a través del método `start()`, tal como se muestra en la Ilustración 50.

```java
package utb.osgi.spring.person.view;

import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;

public class Activator implements BundleActivator{

    private PersonConsole personConsole = new PersonConsole();

    public void start(BundleContext context) throws Exception {
        System.out.println("Bundle 'spring-person-view' Started");
        personConsole.init();
    }

    public void stop(BundleContext context) throws Exception {
        // Do nothing
    }
}
```

Ilustración 50: Clase Activator de 'spring-person-view'

Para brindar mayor claridad sobre el flujo de ejecución que este *bundle* tiene, la siguiente es la secuencia que interesa entender:

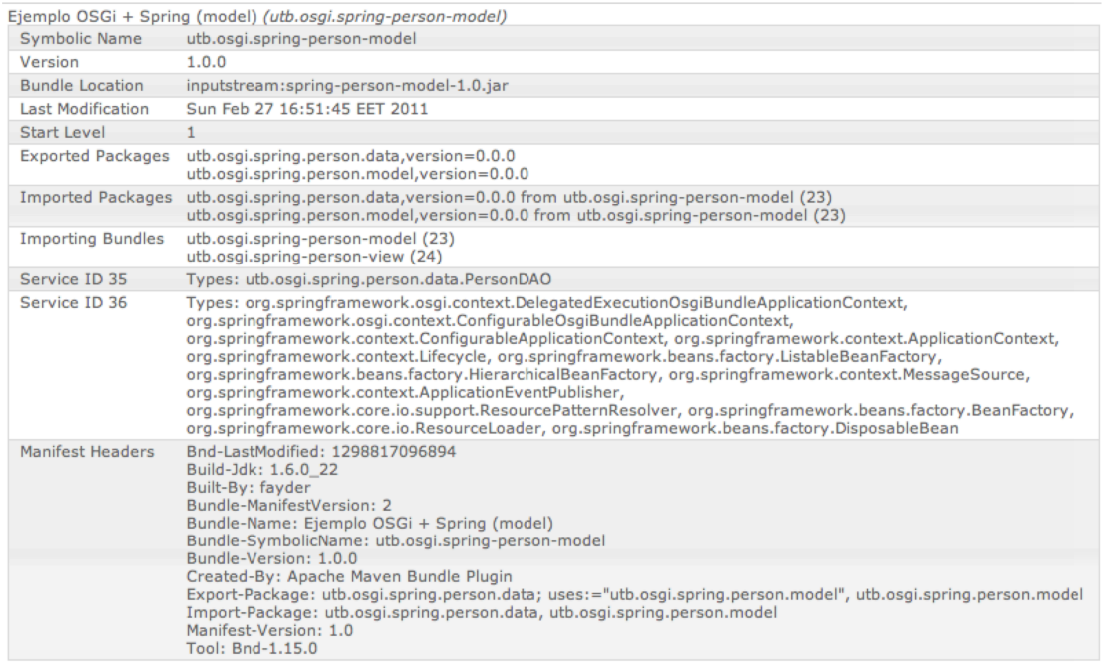
1. El *bundle* es iniciado en el *Framework* Apache Felix, y este encuentra su clase `Activator` e inicializa la referencia `personConsole`.
2. El Framework Apache Felix invoca el método `start()` de clase `Activator`.
3. El método `start()` invoca el método `init()` de la referencia `personConsole`.
4. Spring-DM se da cuenta de que la referencia `personConsole`, es un Spring *bean* el cual está esperando que se le inyecte una referencia a `PersonDAO`, por lo tanto Spring-DM inyecta esta dependencia por medio del método `setPersonDAO()` de la clase `PersonConsole`.
5. El método `setPersonDAO()` invoca al método `loadAllPersons()`, la cual imprime en consola la lista de objetos `Person` disponibles.

Luego, definimos las cabeceras del archivo MANIFEST a través del *Bundle Plug-in* para Maven, en el archivo POM del *bundle* 'spring-person-view'. Este *bundle* no necesita exportar ningún paquete, y además se necesita definir la clase `Activator`, tal como se muestra en la Ilustración 51.

```
<plugin>
    <groupId>org.apache.felix</groupId>
    <artifactId>maven-bundle-plugin</artifactId>
    <extensions>true</extensions>
    <configuration>
        <instructions>
            <Export-Package>!*</Export-Package>
            <Bundle-Activator>utb.osgi.spring.person.view.Activator</Bundle-Activator>
            <Bundle-Name>Ejemplo OSGi + Spring (view)</Bundle-Name>
        </instructions>
    </configuration>
</plugin>
```

Ilustración 51: Cabeceras MANIFEST de 'spring-person-view'

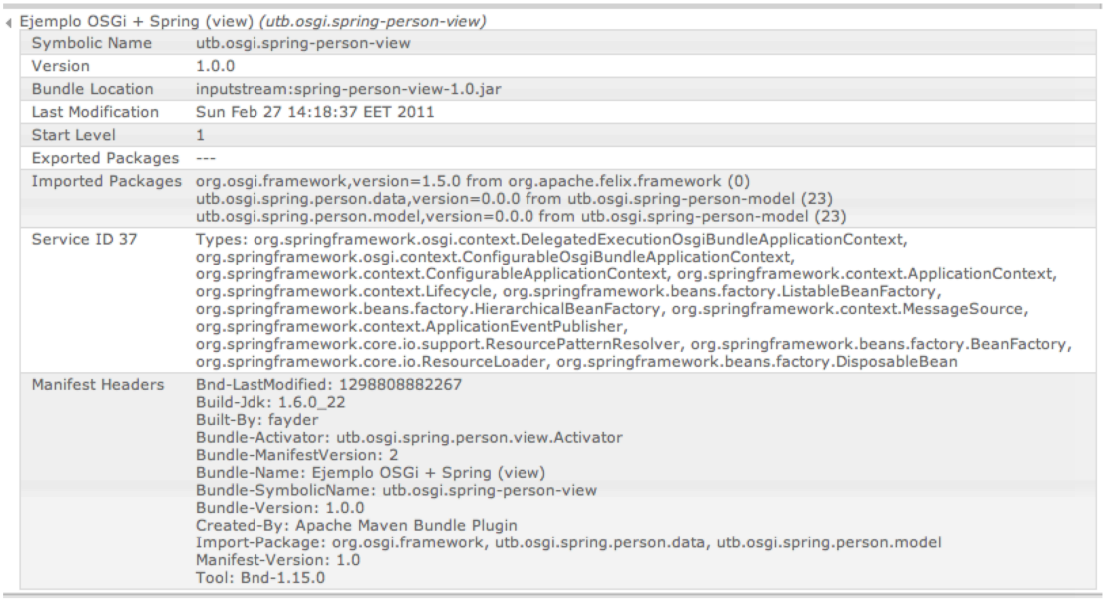
Por último solamente queda instalar los dos *bundles* que se han desarrollado para demostrar la integración entre OSGi y Spring en Apache Felix. Primero se instala el *bundle* 'spring-person-model' y en el *Apache Felix Web Console* la información sobre este *bundle* debe ser similar a la mostrada en la Ilustración 52. Como se puede apreciar, el servicio `PersonDAO` es exitosamente registrado en el Registro de Servicios de OSGi, junto a otros servicios relacionados con Spring-DM para el manejo de Spring *beans*, servicios y el *Application Context* de los *bundles* basados en Spring. Por su parte, la Ilustración 53 muestra la información sobre el *bundle* 'spring-person-view' y como este *bundle* importa los paquetes `utb.osgi.spring.person.data` y `utb.osgi.spring.person.model`.



Ejemplo OSGi + Spring (model) *(utb.osgi.spring-person-model)*

Symbolic Name	utb.osgi.spring-person-model
Version	1.0.0
Bundle Location	inputstream:spring-person-model-1.0.jar
Last Modification	Sun Feb 27 16:51:45 EET 2011
Start Level	1
Exported Packages	utb.osgi.spring.person.data,version=0.0.0 utb.osgi.spring.person.model,version=0.0.0
Imported Packages	utb.osgi.spring.person.data,version=0.0.0 from utb.osgi.spring-person-model (23) utb.osgi.spring.person.model,version=0.0.0 from utb.osgi.spring-person-model (23)
Importing Bundles	utb.osgi.spring-person-model (23) utb.osgi.spring-person-view (24)
Service ID 35	Types: utb.osgi.spring.person.data.PersonDAO
Service ID 36	Types: org.springframework.osgi.context.DelegatedExecutionOsgiBundleApplicationContext, org.springframework.osgi.context.ConfigurableOsgiBundleApplicationContext, org.springframework.context.ConfigurableApplicationContext, org.springframework.context.ApplicationContext, org.springframework.context.Lifecycle, org.springframework.beans.factory.ListableBeanFactory, org.springframework.beans.factory.HierarchicalBeanFactory, org.springframework.context.MessageSource, org.springframework.context.ApplicationEventPublisher, org.springframework.core.io.support.ResourcePatternResolver, org.springframework.beans.factory.BeanFactory, org.springframework.core.io.ResourceLoader, org.springframework.beans.factory.DisposableBean
Manifest Headers	Bnd-LastModified: 1298817096894 Build-Jdk: 1.6.0_22 Built-By: fayder Bundle-ManifestVersion: 2 Bundle-Name: Ejemplo OSGi + Spring (model) Bundle-SymbolicName: utb.osgi.spring-person-model Bundle-Version: 1.0.0 Created-By: Apache Maven Bundle Plugin Export-Package: utb.osgi.spring.person.data; uses:="utb.osgi.spring.person.model", utb.osgi.spring.person.model Import-Package: utb.osgi.spring.person.data, utb.osgi.spring.person.model Manifest-Version: 1.0 Tool: Bnd-1.15.0

Ilustración 52: Bundle 'spring-person-model' en Apache Felix Web Console



Ejemplo OSGi + Spring (view) *(utb.osgi.spring-person-view)*

Symbolic Name	utb.osgi.spring-person-view
Version	1.0.0
Bundle Location	inputstream:spring-person-view-1.0.jar
Last Modification	Sun Feb 27 14:18:37 EET 2011
Start Level	1
Exported Packages	---
Imported Packages	org.osgi.framework,version=1.5.0 from org.apache.felix.framework (0) utb.osgi.spring.person.data,version=0.0.0 from utb.osgi.spring-person-model (23) utb.osgi.spring.person.model,version=0.0.0 from utb.osgi.spring-person-model (23)
Service ID 37	Types: org.springframework.osgi.context.DelegatedExecutionOsgiBundleApplicationContext, org.springframework.osgi.context.ConfigurableOsgiBundleApplicationContext, org.springframework.context.ConfigurableApplicationContext, org.springframework.context.ApplicationContext, org.springframework.context.Lifecycle, org.springframework.beans.factory.ListableBeanFactory, org.springframework.beans.factory.HierarchicalBeanFactory, org.springframework.context.MessageSource, org.springframework.context.ApplicationEventPublisher, org.springframework.core.io.support.ResourcePatternResolver, org.springframework.beans.factory.BeanFactory, org.springframework.core.io.ResourceLoader, org.springframework.beans.factory.DisposableBean
Manifest Headers	Bnd-LastModified: 1298808882267 Build-Jdk: 1.6.0_22 Built-By: fayder Bundle-Activator: utb.osgi.spring.person.view.Activator Bundle-ManifestVersion: 2 Bundle-Name: Ejemplo OSGi + Spring (view) Bundle-SymbolicName: utb.osgi.spring-person-view Bundle-Version: 1.0.0 Created-By: Apache Maven Bundle Plugin Import-Package: org.osgi.framework, utb.osgi.spring.person.data, utb.osgi.spring.person.model Manifest-Version: 1.0 Tool: Bnd-1.15.0

Ilustración 53: Bundle 'spring-person-view' en Apache Felix Web Console

Una vez se ha iniciado el *bundle* 'spring-person-view' en Apache Felix, se debe ver la información sobre los objetos tipo Person en la línea de comando donde se está ejecutando el *Framework*, tal como se ve en la Ilustración 54.

```
-
Bundle 'spring-person-view' Started
g! Spring Injecting PersonDAO
:: All Persons ::
ID: 1
Firstname: John
Lastname: Rambo
Age: 34

ID: 2
Firstname: Rocky
Lastname: Balboa
Age: 32

ID: 3
Firstname: Joseph
Lastname: Dredd
Age: 40


g!
```

Ilustración 54: Bundle 'spring-person-view' en Apache Felix

Este Caso de Estudio, muestra de una manera clara la separación de responsabilidades o separación de capas de una manera muy clara. Por un lado se tiene el *bundle* 'spring-person-model' el cual contiene lo que se puede llamar la parte del Modelo dentro de un diseño MVC, y por otro lado, en un *bundle* diferente, se tiene a 'spring-person-view' que actúa como la Vista. La parte del Controlador es ejercida principalmente por Spring y OSGi.

Lo que se vió con estos ejemplos es simplemente la parte de Inyección de Dependencias de Spring y el manejo de Spring *beans* como Servicios OSGi a través de Spring-DM. Sin embargo, Spring-DM ofrece muchas más utilidades para el *Framework* OSGi, pero esas otras utilidades se salen del contexto de este Trabajo de Grado y por eso no se entra en mayor detalle sobre estas.

OSGi + Spring en Aplicaciones Web

Para desarrollar aplicaciones Web usando OSGi y Spring al mismo tiempo, se hace necesario hacer uso de una parte del *Framework* Spring llamada Spring MVC[49], el cual hace parte a su vez del proyecto Spring Web Flow. Y no solo eso, sino que los *bundles* deben tener el formato WAR y no JAR, como se ha visto que tienen los *bundles* OSGi, aunque con Spring-DM es posible convertir esos WAR en *bundles* OSGi. Sin embargo, esto

[49] http://static.springsource.org/spring/docs/2.5.x/reference/mvc.html

no deja de ser una limitante a la hora de desarrollar aplicaciones Web utilizando la tecnología OSGi y Spring en conjunto.

Alternativas de Servidores de Aplicaciones OSGi

Existen varias alternativas en las cuales se puede hacer uso la tecnología OSGi[50]:

- OSGi embebido en archivos WAR: Esta es la forma más conveniente de utilizar la especificación OSGi en el sentido de que no demanda un cambio del contenedor de aplicaciones, pero el lado negativo de este enfoque es que no se logra obtener una completa disponibilidad de toda la tecnología OSGi, porque con esta alternativa el contenedor de aplicaciones crea un *Classpath* diferente para cada WAR, y por lo tanto cada WAR se encuentra aislado, tal como lo pide la especificación de Java EE.
- Ambiente OSGi: Esta es la alternativa que se ha estado utilizando hasta ahora, con el *Framework* Apache Felix, en donde existe solamente un *Classpath* el cual es administrado por OSGi, lo cual potencializa las capacidades de la implementación OSGi. La desventaja de esta alternativa es que la gran mayoría de aplicaciones Web con Java están empaquetadas con el formato WAR, y los ambientes puramente OSGi no manejan este formato, a menos que esos WAR sean OSGi-ficados.
- Servidores de Aplicaciones Híbridos: Este tipo de servidores de aplicaciones son capaces de manejar tanto archivos WAR como *bundles* OSGi, por ejemplo Eclipse Virgo.

[50] *Cf.* RUBIO, Daniel. Pro Spring Dynamic Modules for OSGi Service Platforms. United States of America: Apress, 2009, p. 113.

2.7 OSGi + Tapestry: El Plug-in Tapestry OSGi

Código Fuente: Proyectos 'tapestry-person-model' y 'tapestry-person-view'
Folder con ejemplos instalados: felix-tapestry

Se podría decir que uno de los problemas o limitantes que tiene el desarrollo Web es el hecho de que resulta ser de muy bajo nivel, comparado con el desarrollo para aplicaciones de escritorio. Un desarrollador de aplicaciones Web debe tener en cuenta una larga cadena de procesos de bajo nivel (métodos GET y POST) que se ejecutan antes y después de que este pueda hacer algo útil en respuesta a una acción del usuario, desde la Interfaz de Usuario o UI (por sus siglas en inglés).

Sin embargo, hoy en día existen muchas soluciones a este problema. En el mundo de Java tenemos tecnologías como Spring[51], Apache Struts[52], JavaServer Faces[53], Apache Wicket[54] y Apache Tapestry[55], entre muchos otros.

Tapestry ofrece un paradigma de desarrollo similar al Desarrollo Rápido de Aplicaciones o RAD por sus siglas en inglés, en el sentido de que por ejemplo en Tapestry un botón en una página tiene un manejador de eventos o *event handler* asociado en una forma declarativa (muy parecido al paradigma de desarrollo de aplicaciones de escritorio). El desarrollador Web con Tapestry no necesita pensar en los detalles de bajo nivel que asocian al botón con su *event handler*.

Todas las tecnologías de desarrollo Web tienen que manejar código HTML de una manera u otra; al final lo que se le envía al usuario es código HTML. Para desarrollar páginas dinámicas, muchas tecnologías combinan HTML con alguna lógica de negocio, rompiendo con una clara separación de responsabilidades, por lo menos en un diseño MVC o parecido, además de conllevar otra serie de desventajas como complejidad en el mantenimiento, etc.

[51] http://www.springsource.org/
[52] http://struts.apache.org/
[53] http://www.javaserverfaces.org/
[54] http://wicket.apache.org/
[55] http://tapestry.apache.org/

En Tapestry5 existe una separación de responsabilidades más clara, es decir que la cantidad de lógica existente en la vista (plantillas HTML) es poca o nula. Tapestry5 es un *Framework* basado en componentes, así como JavaServer Faces y ASP.NET, pero la facilidad de crear componentes propios varía de un *Framework* a otro, y Tapestry5 definitivamente es un *Framework* que brinda una facilidad para crear componentes personalizados muy significativa. Por ejemplo para crear componentes personalizados en JavaServer Faces se debe tener un conocimiento amplio y considerable sobre la estructura interna del *Framework*, mientras que en Tapestry5 incluso un principiante puede crear ciertos componentes personalizados.

Tapestry divide una aplicación Web en un conjunto de páginas, las cuales contienen componentes. Esto provee una estructura consistente de la aplicación Web, lo que le permite a Tapestry hacerse cargo de tareas como la construcción de URL, validación de entrada, internacionalización y manejo de excepciones, entre otras. En Tapestry, cada plantilla HTML posee una clase Java asociada, la cual maneja sus eventos. El slogan del *Framework* Tapestry es '*Code less, deliver more*', con lo cual quieren decir que el principal objetivo de Tapestry es hacer que el programador necesite escribir menos código para terminar aplicaciones Web de manera eficiente.

Tapestry5 tiene su propia técnica de Inversión de Control e Inyección de Dependencias, las cuales se vió funcionar de manera básica anteriormente con el *Framework* Spring, por lo que Tapestry5 no necesita de Spring en este aspecto, aunque las dos tecnologías son completamente compatibles y de hecho es muy ventajoso trabajar con ambas a la vez.

Con respecto a la integración de OSGi y Tapestry5, que en esencia es de lo que se trata este Caso de Estudio, el desarrollador llamado Donf Yang elaboró un *plug-in* llamado Tapestry-OSGi[56], el cual aún se encuentra en su versión beta. Este *plug-in* tiene como objetivo brindar las capacidades de Tapestry5 en la construcción de aplicaciones Web en un ambiente OSGi. Sin embargo, la última versión beta de este *plug-in* no brinda todas las capacidades que tiene el *Framework* Tapestry5, por ejemplo no es posible integrar el

[56] http://groups.google.com/group/tapestry-osgi

Framework Spring-DM con el *plug-in* Tapestry-OSGi, y por esta razón el *plug-in* hace uso de su propia técnica de Inversión de Control.

Para comenzar nuestro ejemplo, lo primero que hay que hacer es descargar el *Plug-in Tapestry OSGi* de la siguiente dirección: http://extwind.googlecode.com/svn/release/tapestry-osgi/tapestry-osgi-v1.0-beta.zip. Así como se hizo con Spring-DM, este .zip contiene una serie de *bundles* que se deben instalar en Apache Felix para aplicaciones Web con Tapestry en un ambiente OSGi, los cuales están listados a continuación:

- `org.antlr.runtime_3.1.1`
- `org.apache.commons.codec_1.3.0`
- `org.eclipse.equinox.http.servlet_1.0.100.v20080427-0830`
- `org.eclipse.osgi.services_3.2.0.200907062331`
- `org.extwind.osgi.bundles.commons.logging_1.0.4.v20080605-1930`
- `org.extwind.osgi.bundles.javassist_3.10.0`
- `org.extwind.osgi.bundles.mortbay.jetty6_6.1.17`
- `org.extwind.osgi.bundles.tapestry_5.1.0.5`
- `org.extwind.osgi.equinox.http.jetty6_1.0.0`
- `org.extwind.osgi.http.filter_1.0.0`
- `org.extwind.osgi.tapestry_5.1.0.5`
- `org.extwind.osgi.tapestry.components_1.0.0`
- `stax_api_1.0.1`
- `stax2-api_3.0.1`
- `woodstox-core-asl_4.0.3`

En la lista anterior se encuentran varios *bundles* relacionados con Jetty, lo cual significa que en esta ocasión no se usará el *bundle* 'HTTP Service Jetty' que provee el *Framework* Apache Felix, sino el que viene con el *Plug-in Tapestry OSGi*.

El ejemplo que se va a desarrollar a continuación es muy parecido a aquel desarrollado para demostrar la integración de OSGi y Spring. Constará de 2 *bundles*, uno de ellos contendrá el Modelo de la aplicación, y el otro las Vistas y los Controladores, teniendo en cuenta el diseño MVC. El *bundle* que contiene la parte del Modelo se llamará 'tapesty-person-

model'. Así como el *bundle* 'spring-person-model', este *bundle* contiene la interfaz `PersonDAO`, su implementación `PersonDAOImpl` y una fuente de datos artificial llamada `PersonDataSource`, pero además también contiene una clase llamada `PersonDAOFactory` la cual provee una instancia de `PersonDAO`. A diferencia del ejemplo 'spring-person-model', en esta ocasión se neceista una forma de obtener una instancia de `PersonDAO`, debido a que Tapestry no es compatible con Spring, y además la clase `PersonDAOImpl` no será visible en Apache Felix, al estar en un paquete Privado. En la Ilustración 55 se puede ver el contenido de la case `PersonDAOFactory`.

PersonDAOFactory.java

```
package utb.osgi.tapestry.person.data;

import utb.osgi.tapestry.person.data.impl.PersonDAOImpl;

public class PersonDAOFactory {

    public static PersonDAO createPersonDAO() {
        return new PersonDAOImpl();
    }

}
```

Ilustración 55: Clase PersonDAOFactory de 'tapestry-person-model'

```
package utb.osgi.tapestry.person.data.impl;

import java.util.List;

public class PersonDAOImpl implements PersonDAO{

    private PersonDataSource dataSource;

    public PersonDAOImpl() {
        this.dataSource = new PersonDataSource();
    }

    public Person load(Integer id) {
        return dataSource.load(id);
    }

    public List<Person> loadAll() {
        return dataSource.loadAll();
    }

    public void insert(Person person) {
        dataSource.insert(person);
    }

    public Boolean delete(Person person) {
        return dataSource.delete(person);
    }

    public void update(Person person) {
        dataSource.update(person);
    }

}
```

Ilustración 56: Clase PersonDAOImpl de 'tapestry-person-model'

En esta ocasión la clase `PersonDAOImpl` no toma un argumento de tipo `PersonDataSource` en su constructor, como en el ejemplo 'spring-person-model', sino que esta vez simplemente es inicializado, como se muestra en la Ilustración 56.

El *bundle* 'tapestry-person-model' exportará los paquetes `utb.osgi.tapestry.model` y `utb.osgi.tapestry.data`, y mantendrá el paquete `utb.osgi.tapestry.data.impl` como Privado, de manera similar al *bundle* 'spring-person-model' desarrollado anteriormente.

Con respecto al *bundle* 'tapestry-person-view', en la Ilustración 57 se puede ver el código HTML que por ahora tiene el archivo `Index.tml`, el cual será nuestra página de Inicio. Lo primero que se puede notar es el Espacio de Nombre `xmlns:t="http://tapestry.apache.org/schema/tapestry_5_1_0.xsd"` definido en el elemento `<html>`. Este Espacio de Nombres permite utilizar los componentes básicos que provee Tapestry5, como por ejemplo el componente `<t:pagelink>` el cual se interpreta como un link a otra página; más exactamente, el archivo `Index.tml` posee una instancia del componente de Tapestry `<t:pagelink>` El nombre de la página a donde este link apunta es definido en el parámetro `page`, que en este caso es `ShowAll`. Más adelante se verán los detalles de la página `ShowAll`.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html xmlns:t="http://tapestry.apache.org/schema/tapestry_5_1_0.xsd">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
<title>Ejemplo OSGi + Tapestry5</title>
</head>
<body>
    <h1>Ejemplo OSGi + Tapestry5</h1>

    <t:pagelink page="ShowAll">Ver Todas las Personas</t:pagelink>
</body>
</html>
```

Ilustración 57: Versión 1 de archivo Index.tml de 'tapestry-person-view'

Las páginas en Tapestry son una combinación de un POJO y una plantilla tml, ambos con exactamente el mismo nombre, excepto por la extensión, ya que uno es una clase .java y el otro es un archivo .tml. El archivo .tml contienen el código HTML de la página, lo cual sería la parte de la Vista en un diseño MVC, y la clase Java actúa como Controlador.

Ambos archivos se encuentran en el mismo paquete, solamente que en archivos fuente diferentes. Por ejemplo, el archivo `Index.tml` se encuentra exactamente en `src/main/resources/utb.osgi.tapestry.pages` y el archivo `Index.java`, el cual se puede apreciar en la Ilustración 58, se encuentra en `src/main/java/utb.osgi.tapestry.pages`.

```
package utb.osgi.tapestry.pages;

public class Index {

}
```

Ilustración 58: Versión 1 de clase Index de 'tapestry-person-view'

La clase Index por ahora es una clase vacía, ya que no se neceista ninguna lógica para que la pareja `Index.java` e `Index.tml` funcionen debidamente. Si el usuario hace clic en el link 'Ver Todas las Personas', este evento es manejado por el componente de Tapestry `<t:pagelink>`. Para que este link pueda funcionar debidamente y muestre todas las personas, se necesita de otra pareja de archivos que, de acuerdo a la Ilustración 59, se llaman `ShowAll.tml` y `ShowAll.java`.

```
<?xml version="1.0" encoding="utf-8" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:t="http://tapestry.apache.org/schema/tapestry_5_0_0.xsd"
    xmlns:p="tapestry:parameter">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
<title>All Persons</title>
</head>
<body>
    <h2>All Persons</h2>

    <t:grid source="allPersons" row="person" />
</body>
</html>
```

Ilustración 59: Plantilla ShowAll.tml de 'tapestry-person-view'

La plantilla HTML `ShowAll.tml`, la cual se muestra en la Ilustración 59, posee el componente Tapestry llamado `<t:grid>` el cual tiene dos atributos: `source` y `row`. El atributo `source` toma como argumento un arreglo de objetos o una Lista, para luego presentar esta información de forma tabular. Por su parte, el atributo `row` toma como argumento el tipo de objeto que estará representado en cada fila de la tabla o grilla. Estos

argumentos le son proveídos a la plantilla `ShowAll.tml` a través de su clase asociada `ShowAll.java`. La clase `ShowAll` posee dos atributos: la propiedad `personDAO` de tipo `PersonDAO`, y la propiedad `person` de tipo `Person`. Estas dos propiedades tienen una anotación asociada cada una. La propiedad personDAO posee la anotación `DynamicInject`, la cual es proveída por el *plug-in Tapestry OSGi* y su función es la de inyectar una instancia de este objeto en la clase `ShowAll`. Es decir que esta anotación tiene la tarea de realizar la Inyección de Dependencias que hasta ahora no se tenía debido a que no se está utilizando el *Framework* Spring. Pero el *plug-in Tapestry OSGi* debe obtener una instancia de la interfaz `PersonDAO` de alguna parte, esto se verá más adelante.

La clase `ShowAll` además también posee el método `getAllPersons()` el cual regresa una Lista de tipo `Person`, así como se muestra en la Ilustración 60.

```
package utb.osgi.tapestry.pages;

import java.util.List;

import org.apache.tapestry5.annotations.Property;
import org.extwind.osgi.tapestry.annotations.DynamicInject;

import utb.osgi.tapestry.person.data.PersonDAO;
import utb.osgi.tapestry.person.model.Person;

public class ShowAll {

    @DynamicInject
    private PersonDAO personDAO;

    @Property
    private Person person;

    public List<Person> getAllPersons() {
        return personDAO == null ? null : personDAO.loadAll();
    }
}
```

Ilustración 60: Clase ShowAll.java de 'tapestry-person-model'

La clase `ComponentModule`, que se muestra en la Ilustración 61, es la que se encarga de proveer una instancia concreta de la interfaz `PersonDAO` a cualquier clase que intente obtenerla por la vía de Inyección de Dependencias. Cuando el *plug-in Tapestry OSGi* encuentra una anotación `DynamicInject`, busca en la clase ComponentModule por algún método con el nombre '`built+<nombre de la interfaz>`', que en este caso es `buildPersonDAO()`.

```java
package utb.osgi.tapestry.internal;

import utb.osgi.tapestry.person.data.PersonDAO;
import utb.osgi.tapestry.person.data.PersonDAOFactory;

public class ComponentModule {

    public static PersonDAO buildPersonDAO() {
        return PersonDAOFactory.createPersonDAO();
    }
}
```

Ilustración 61: Clase ComponentModule de 'tapestry-person-view'

Para que esete ejemplo funcione correctamente, se debe crear un archivo llamado `component.xml` en la ruta `META-INF/extwind`. En este caso, este archivo contiene un elemento `<package>` y un elemento `<module>`. El elemento `<package>` define el paquete raíz en donde se encuentran las páginas y componentes de Tapestry, que en nuestro caso es el paquete `utb.osgi.tapestry`. El elemento `<module>` por su parte define la clase que proveerá servicios dinámicamente (Inyección de Dependencias), y en este caso esa clase es `utb.osgi.tapestry.internal.ComponentModule`. El contenido del archivo `component.xml` se puede ver en la Ilustración 62.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<component xmlns="http://www.extwind.org/schema/component"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.extwind.org/schema/component
    http://www.extwind.org/schema/component.xsd">

    <package>utb.osgi.tapestry</package>
    <module>utb.osgi.tapestry.internal.ComponentModule</module>

</component>
```

Ilustración 62: Archivo component.xml de 'tapestry-person-view'

El *bundle* 'tapestry-person-view' tiene solamente un paquete que es privado: utb.osgi.tapestry.internal, el resto debe ser exportado para que la aplicación funcione correctamente. Es decir todas las páginas y componentes de Tapestry de la aplicación deben ser exportados.

Una vez los *bundles* 'tapestry-person-model' y 'tapestry-person-view' son empaquetados e instalados en Apache Felix, en la dirección http://localhost:8080/Index se debe observar algo parecido a la Ilustración 63. Luego si se hace clic en el link 'Ver Todas las Personas', se debe ver algo parecido a la Ilustración 64. Una de las capacidades básicas del componente `<t:grid>` de Tapestry, es que se puede re-ordenar la tablar haciendo clic sobre

sus cabeceras, se puede intentar hacer esto también.

Ilustración 63: Página Index de Ejemplo OSGi + Tapestry

Ilustración 64: Página ShowAll de Ejemplo OSGi + Tapestry

Así concluye entonces este Caso de Estudio de compatibilidad OSGi y Tapestry5. Como se ha comprobado, el *plug-in Tapestry OSGi* permite usar las capacidades de Tapestry para crear aplicaciones Web en ambientes OSGi, y a la vez le permite a Tapestry usar las capacidades que tiene OSGi de modularidad, actualización en tiempo de ejecución y manejo de versiones. Sin embargo, Tapestry por si solo es capaz de manejar la modularidad de una forma similar a OSGi.

Modularidad de Tapestry sin OSGi[57]

Con Tapestry también es posible hacer uso del archivo MANIFEST para desarrollar

[57] *Cf.* DOBRIAZKO, Igor. Tapestry IoC: Modularization of Web applications without OSGi. Tapestry5[Web log] Artículo del 19 de Enero del 2010. Disponible en Internet ne la dirección: http://blog.tapestry5.de/index.php/2010/01/19/tapestry-ioc-modularization-of-web-applications-without-osgi/

aplicaciones de manera modular, muy similar a como funciona la modularidad en OSGi. Los modulos de Tapestry también se empaquetan en archivos JAR, y el *Framework* Tapestry es capaz de buscar y encontrar módulos que se encuentren el *Classpath* de la aplicación. Tapestry logra hacer esto a través de una cabecera en el archivo MANIFEST llamada `Tapestry-Module-Classes`, el cual toma como valor una lista de clases, con su nombre completo, separados por comas. En OSGi es necesario registrar los servicios y recursos que se quieran hacer públicos a través de una clase que implemente la interfaz `BundleActivator`, mientras que en Tapestry, la técnica de Inversión de Control o Inyección de Dependencias propias del *Framework* realizan este trabajo automáticamente. De igual forma, mientras que en OSGi es necesario buscar en el Registro de Servicios algún servicio o recurso que se requiera, en Tapestry la Inyección de Dependencias se encarga de este proceso.

La demostración de la modularidad de Tapestry se sale del contexto de este Trabajo de Grado, por eso no se desarrollará un ejemplo para ver de mejor manera esta característica de Tapestry. Sin embargo, se puede afirmar que si se quiere utilizar una modularidad mucho mejor de la que provee la plataforma Java en sí, y si no es tan crucial actualizar aplicaciones en tiempo de ejecución, entonces Tapestry provee las capacidades necesarias para lograr este objetivo en una aplicación Web.

2.7 Ventajas de la Especificación OSGi

La especificación OSGi provee una gama de ventajas y soluciones a problemas, algunos de los cuales son de hecho considerados intrínsicos o naturales, que existen en el mundo de desarrollo de aplicaciones Java. A continuación se describirá brevemente las características más representativas de la especificación OSGi, muchas de las cuales ya se han visto en los Casos de Estudio desarrollados anteriormente, pero que tal vez no fueron señalados o detallados en su momento.

2.7.1 Modularidad

Esta característica de OSGi es la más fácil de observar, y la se ha visto en casi todos los ejemplos que se han desarrollado en este Trabajo de Grado. Parte del concepto de modularidad consiste en poder dividir una aplicación en diferentes archivos JAR, cada uno representando una parte lógica de la aplicación. Este comportamiento se pudo ver por ejemplo en los *bundles* 'tapestry-person-model' y 'tapestry-person-view'. Si la guía es una arquitectura MVC, en el *bundle* 'tapestry-person-model' se tiene la parte del Modelo, es decir la Lógica del Negocio, mientras que en el *bundle* 'tapestry-person-view' se tiene la parte de la Vista y Controlador de la aplicación, logrando así una clara separación de responsabilidades.

Sin ninguna duda OSGi brinda un nivel de modularidad superior al que existe en la mayor parte del mundo de desarrollo Java, el cual implementa una limitada modularidad a través de paquetes, que residen en un mismo archivo JAR o WAR.

Pero una buena modularidad no está completa sin un Control de Versiones eficaz, algo que también está prácticamente ausente en la plataforma Java.

Apache Tomcat

Es posible lograr una separación similar y hacerla correr en Apache Tomcat. Esta

separación consistiría por ejemplo en colocar la parte del modelo en un JAR y la parte de la vista y controlador en un WAR. Sin embargo existe una diferencia importante: el archivo JAR debe ser parte del archivo WAR para que pueda funcionar en el servidor de aplicaciones Apache Tomcat. Este decir que deben ser desplegados en un mismo contenedor (el archivo WAR), por lo tanto la modularidad se deteriora con respecto a como funciona en un servidor de aplicaciones OSGi. Además, como el archivo JAR hace parte del archivo WAR, en muchas ocasiones esto significa que el WAR puede "ver", no solamente las interfaces o clases que requiere, sino todas las demás que existan en el archivo JAR.

2.7.2 Control de Versiones y Actualización Dinámica

Folder con ejemplos instalados: felix-versioning

En los Casos de Estudio que se han desarrollado durante este Trabajo de Grado, no se ha estado utilizando esta característica de OSGi, pero vale la pena aclarar que es una característica muy importante a la hora de desarrollar *bundles* de OSGi, porque facilita el re-uso, la compatibilidad y el manejo de *bundles* en un ambiente OSGi.

Lo que permite el Control de Versiones es básicamente la capacidad de tener paquetes con el mismo nombre o firma dentro del *Classpath* de OSGi pero con versiones diferentes. Esto no es posible hacerlo en servidores Web tradicionales.

Para ilustrar esta característica de OSGi, se puede tomar uno de los ejemplos anteriormente desarrollados, e instalar en Apache Felix dos versiones diferentes de este *bundle*. Por ejemplo, si se toma el *bundle* 'tapestry-person-model' y en las instrucciones del *Bundle Plugin* para Maven, se define la versión '1.0.0' a los paquetes que se están exportando, tal como se muestra en la ilustración 65.

```
<instructions>
    <Export-Package>
        !utb.osgi.tapestry.person.data.impl.*,
        utb.osgi.tapestry.person.data;version=1.0.0,
        utb.osgi.tapestry.person.model;version=1.0.0
    </Export-Package>
    <Private-Package>
        utb.osgi.tapestry.person.data.impl.*
    </Private-Package>
    <Bundle-Name>Ejemplo OSGi + Tapestry (model)</Bundle-Name>
</instructions>
```

Ilustración 65: Paquetes exportados de 'tapestry-person-model' versión 1.0.0

Ejemplo OSGi + Tapestry (model) *(utb.osgi.tapestry-person-model)*

Symbolic Name	utb.osgi.tapestry-person-model
Version	1.0.0
Bundle Location	inputstream:tapestry-person-model-1.0.jar
Last Modification	Sun Mar 13 09:54:10 EET 2011
Start Level	1
Exported Packages	utb.osgi.tapestry.person.data,version=1.0.0 utb.osgi.tapestry.person.model,version=1.0.0

Ilustración 66: Bundle 'tapestry-person-model' con versiones 1.0.0 en Apache Felix Web Console

Luego de instalar el *bundle* 'tapestry-person-model' en Apache Felix con las versiones de los paquetes exportados definidas, se debe ver algo parecido a la Ilustración 66.

Si ahora se instala el *bundle* 'tapestry-person-view' sin ninguna modificación, este *bundle*, aunque no se ha definido la versión de los paquetes que se necesitan del *bundle* 'tapestry-person-model', importa automáticamente las versiones 1.0.0 del *bundle* 'tapestry-person-model', ya que son los únicos disponibles. Así se podemos ver en la Ilustración 67.

Ejemplo OSGi + Tapestry5 (view) *(utb.osgi.tapestry-person-view)*

Symbolic Name	utb.osgi.tapestry-person-view
Version	1.0.0
Bundle Location	inputstream:tapestry-person-view-1.0.jar
Last Modification	Sun Mar 13 10:05:07 EET 2011
Start Level	1
Exported Packages	layout,version=0.0.0 layout.images,version=0.0.0 utb.osgi.tapestry.pages,version=0.0.0
Imported Packages	org.apache.tapestry5,version=0.0.0 from org.extwind.osgi.bundles.tapestry (18) org.apache.tapestry5.annotations,version=0.0.0 from org.extwind.osgi.bundles.tapestry (18) org.apache.tapestry5.ioc.annotations,version=0.0.0 from org.extwind.osgi.bundles.tapestry (18) org.extwind.osgi.tapestry.annotations,version=0.0.0 from org.extwind.osgi.tapestry (15) utb.osgi.tapestry.person.data,version=1.0.0 from utb.osgi.tapestry-person-model (23) utb.osgi.tapestry.person.model,version=1.0.0 from utb.osgi.tapestry-person-model (23)

Ilustración 67: Bundle 'tapestry-person-view' importa versiones 1.0.0 de los paquetes exportados por el bundle 'tapestry-person-model'

Con estos dos *bundles* instalados en Apache Felix, si se revisa la dirección http://localhost:8080/Index, se puede ver que el ejemplo sigue funcionando tal cual como lo

hizo originalmente.

Ahora lo que se hará será instalar una nueva versión del *bundle* 'tapestry-person-model' en Apache Felix con algunas modificaciones para constatar luego de que de hecho funciona como se espera. La modificación que consiste en agregar dos personas más a la lista de personas en el constructor de la clase `PersonDataSource` del *bundle* 'tapestry-person-model', así como se puede ver en la Ilustración 68.

Se debe cambiar la versión del *bundle*, en la instrucción `<version>` en el archivo POM del *bundle* 'tapestry-person-model', así como también la versión de los paquetes exportados, que ahora es '2.0.0' tal como se muestra en la Ilustración 69.

```
public class PersonDataSource {

    private List<Person> persons;

    public PersonDataSource() {

        Person personOne = new Person(1, "John", "Rambo", 34);
        Person personTwo = new Person(2, "Rocky", "Balboa", 32);
        Person personThree = new Person(3, "Joseph", "Dredd", 40);
        Person personFour = new Person(4, "Kid", "Pambele", 50);
        Person personFive = new Person(5, "El Pibe", "Valderrama", 55);

        persons = new ArrayList<Person>();
        persons.add(personOne);
        persons.add(personTwo);
        persons.add(personThree);
        persons.add(personFour);
        persons.add(personFive);
    }
```

Ilustración 68: Constructor de la Clase PersonDataSource del *bundle* 'tapestry-person-model' versión 2.0

```
<instructions>
    <Export-Package>
        !utb.osgi.tapestry.person.data.impl.*,
        utb.osgi.tapestry.person.data;version=2.0.0,
        utb.osgi.tapestry.person.model;version=2.0.0
    </Export-Package>
    <Private-Package>
        utb.osgi.tapestry.person.data.impl.*
    </Private-Package>
    <Bundle-Name>Ejemplo OSGi + Tapestry (model) Version 2</Bundle-Name>
</instructions>
```

Ilustración 69: Paquetes exportados de 'tapestry-person-model' versión 2.0.0

Para asegurarse de que el *bundle* 'tapestry-person-view' utilice los paquetes del *bundle* 'tapestry-person-view', solamente se debe cambiar el valor del elemento `<version>` de la

dependencia `<tapestry-person-model>` en el archivo POM del *bundle* 'tapestry-person-view', así como se muestra en la Ilustración 79. Vale la pena aclarar que también se debe instalar el *bundle* 'tapestry-person-model' versión 2.0 en el repositorio local de maven, utilizando el comando `mvn clean install`.

```
<!-- API de PersonDAO -->
    <dependency>
        <groupId>utb.osgi</groupId>
        <artifactId>tapestry-person-model</artifactId>
        <version>2.0</version>
        <scope>provided</scope>
    </dependency>
```

Ilustración 70: Versión de dependencia 'tapestry-person-model' del bundle 'tapestry-person-view' actualizada a 2.0

Luego de instalar los nuevos *bundles* en Apache Felix, 'tapestry-person-model' versión 2.0 y 'tapestry-person-view', se puede ver en Apache Felix que el *bundle* 'tapestry-person-view' ahora importa los paquetes 2.0.0 del *bundle* 'tapestry-person-model' versión 2.0, tal como se quiere y es mostrado en la Ilustración 71.

Ejemplo OSGi + Tapestry5 (view) *(utb.osgi.tapestry-person-view)*

Symbolic Name	utb.osgi.tapestry-person-view
Version	1.0.0
Bundle Location	inputstream:tapestry-person-view-1.0.jar
Last Modification	Sun Mar 13 11:03:06 EET 2011
Start Level	1
Exported Packages	layout,version=0.0.0 layout.images,version=0.0.0 utb.osgi.tapestry.pages,version=0.0.0
Imported Packages	org.apache.tapestry5,version=0.0.0 from org.extwind.osgi.bundles.tapestry (18) org.apache.tapestry5.annotations,version=0.0.0 from org.extwind.osgi.bundles.tapestry (18) org.apache.tapestry5.ioc.annotations,version=0.0.0 from org.extwind.osgi.bundles.tapestry (18) org.extwind.osgi.tapestry.annotations,version=0.0.0 from org.extwind.osgi.tapestry (15) utb.osgi.tapestry.person.data,version=2.0.0 from utb.osgi.tapestry-person-model (23) utb.osgi.tapestry.person.model,version=2.0.0 from utb.osgi.tapestry-person-model (23)

Ilustración 71: Bundle 'tapestry-person-view' importa los paquetes exportados por 'tapestry-person-model' versión 2.0

Y si ahora se revisa la URL http://localhost:8080/ShowAll se verá que la lista de personas ya no es de tres, sino de cinco, así como se muestra en la Ilustración 72.

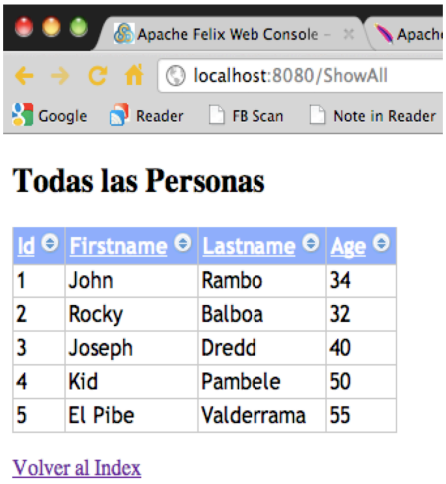


Ilustración 72: Página ShowAll de 'tapestry-person-view' actualizada con 'tapestry-person-model' versión 2.0

Las versiones de los paquetes en la especificación OSGi también se pueden dar en rangos, es decir que un *bundle* puede definir que puede importar paquetes en sus versiones de la 1.0.0 a la 3.0.0. Para esto se usan corchetes y paréntesis. El uso de corchetes es la versión inclusiva del rango, y los paréntesis la versión exclusiva. Por ejemplo, se puede definir que un *bundle* importa un paquete con la siguiente versión "(1.0.0, 3.4.0]". Esto significa que este *bundle* importa las versiones superiores a 1.0.0, sin incluirlo (es decir que no importa la versión 1.0.0 pero si importaría la versión 1.0.1), hasta la versión 3.4.0, inclusivamente. Para más información sobre el manejo de versiones en la especificación OSGi, se puede revisar la página 32 de la especificación OSGi llamada '*OSGi Service Platform Core Specification Release 4, Version 4.2*' de la Alianza OSGi.

Es muy importante anotar que la instalación de los *bundle* 'tapestry-person-model' versión 2.0 y 'tapestry-person-view' actualizado, se hicieron sin necesidad de reiniciar Apache Felix, por lo tanto las actualizaciones y los cambios ocurrieron en tiempo de ejecución, con lo cual también se ha demostrado la característica de **Actualización Dinámica de OSGi.**

Con aras de observar de forma más clara la característica de Actualización Dinámica de OSGi, se puede realizar otro pequeño ejemplo. Se modifica el archivo `Index.tml` y se introducen los cambios que se pueden ver en la Ilustración 73: se agrega la versión del *bundle* al título, y además ahora se muestra la fecha actual.

```
<?xml version="1.0" encoding="utf-8" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"  "http://w
<html xmlns:t="http://tapestry.apache.org/schema/tapestry_5_1_0.xsd
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8"
<title>Ejemplo OSGi + Tapestry5</title>
</head>
<body>
    <h1>Ejemplo OSGi + Tapestry5 Versión 1.0.0</h1>

    <p>La fecha actual es ${currentTime}</p>

    <t:pagelink page="ShowAll">Ver Todas las Personas</t:pagelink>
</body>
</html>
```

Ilustración 73: Archivo Index.tml de 'tapestry-person-view' con cambios para actualización dinámica

También se debe hacer un pequeño cambio a la clase `Index` del *bundle* 'tapestry-person-view' para que la propiedad `${currentTime}` tenga un valor adecuado. Este cambio se puede ver en la Ilustración 74.

```
package utb.osgi.tapestry.pages;

import java.util.Date;

public class Index {

    public Date getCurrentTime() {
        return new Date();
    }

}
```

Ilustración 74: Clase Index de 'tapestry-person-view' con cambios para actualización dinámica

Por último solo hace falta instalar el *bundle* en Apache Felix, y sin reiniciar el Servidor Web, en la URL http://localhost:8080/Index se debe ver algo parecido a la Ilustración 75.

Ilustración 75: Página Index de 'tapestry-person-view' después de actualización dinámica

Apache Tomcat

En Apache Tomcat, tal control de versiones y actualización dinámica es simplemente imposible de lograr. En Apache Tomcat, cada archivo WAR tiene dedicado su propio espacio de clases, al contrario de lo que sucede en un ambiente OSGi en donde solamente existe un solo espacio de clases. En Apache Tomcat, si un archivo WAR necesita de alguna clase, este utilizará la primera ocurrencia que encuentre, sin importar su versión, por lo tanto si existen 2 clases de diferente versión, y la que el archivo WAR obtiene no es la clase con la versión que este necesita, entonces habrá un problema y la aplicación no se ejecutará apropiadamente. Una herramienta como Apache Maven puede ayudar al control de versiones de aplicaciones Java en servidores de aplicaciones no OSGi, como Apache Tomcat, pero no resuelve el problema en su totalidad.

En cuanto a actualización dinámica, el servidor de aplicaciones Apache Tomcat posee una herramienta llamada *Manager App*, el cual resulta muy útil en un ambiente de producción ya que brinda la capacidad de desplegar una aplicación, o remover una existente, sin necesidad de detener y re-iniciar todo el contenedor, así como también re-cargar una aplicación existente, para reflejar cambios que se hayan hecho en los contenidos de las rutas `/WEB-INF/classes` y `/META-INF/lib`, por lo que si existe un cambio por fuera de estas rutas, Apache Tomcat no lo tomará en cuenta.

2.7.3 Extensibilidad y Adaptación

Folder con ejemplos instalados: felix-extensibility

Extensibilidad se refiere a poder extender una aplicación a través de otro archivo que provea ciertos servicios o recursos. Es decir, que si se tiene una aplicación Java, se pueda agregar servicios, páginas o recursos simplemente colocando otro archivo JAR en el *Classpath* de esta aplicación.

Para demostrar esta característica de la especificación OSGi, se puede utilizar el Apache Felix Web Console como ejemplo. Este *bundle* de Apache viene preparado para aceptar múltiples extensiones, de manera de que el número de servicios proveídos por Apache Felix Web Console puede ser aumentado durante tiempo de ejecución. Por ejemplo, una de las extensiones que acepta Apache Felix Web Console es el servicio *Log Service*, el cual

actúa como un sistema de *logging* para Apache Felix Web Console, registrando todo lo que suceda en el servidor Web Apache Felix.

Si se hace clic en 'Log Service' la primera vez que se instala Apache Felix Web Console, el sistema dirá que ese servicio no ha sido instalado o no se está ejecutando, tal como se muestra en la Ilustración 76.

Ilustración 76: Servicio Log Service de Apache Felix Web Console

Para instalar este *bundle* y extender las capacidades de Apache Felix Web Console, se puede hacer clic en 'OSGi Repository', buscar 'Apache Felix Log Service', y hacer clic sobre la versión que se quiere instalar. Se debe ver entonces algo parecido a la Ilustración 77.

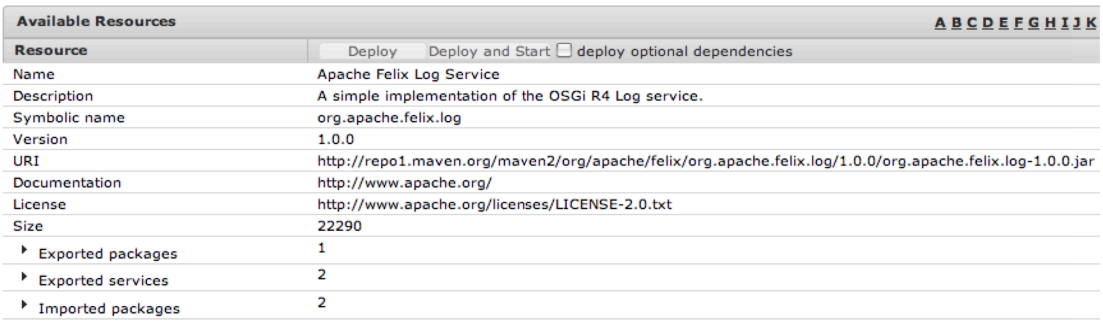


Ilustración 77: Instalando Apache Felix Log Service para Apache Felix Web Console

Luego solamente basta con hacer clic en 'Deploy and Start' para instalar este *bundle* y por ende extender la funcionalidad de Apache Felix Web Console. Una vez instalado, si se vuelve a hacer clic en 'Log Service', el servicio de *logging* comenzará a funcionar inmediatamente, y se debe ver algo parecido a la Ilustración 78.

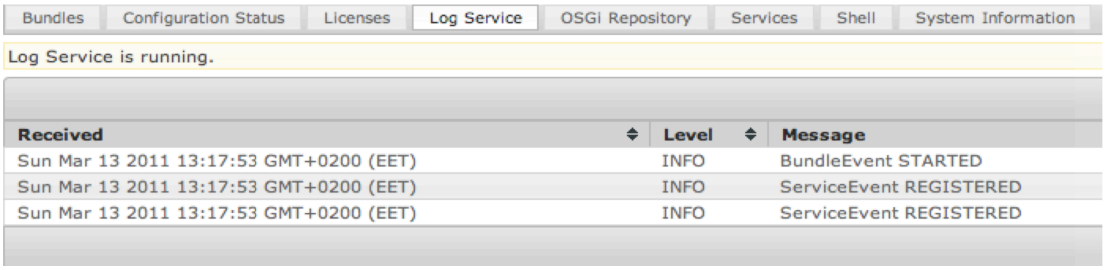


Ilustración 78: Log Service instalado para Apache Felix Web Console

Y de esta manera se ha extendido el Apache Felix Web Console, para proveer un servicio de *logging* que monitoree lo que suceda en el servidor Web Apache Felix.

Para demostrar la característica de Adaptación de la especificación OSGi, también se puede usar al *bundle* Apache Felix Web Console. Como lo se dijo anteriormente, Apache Felix Web Console es capaz de funcionar normalmente, con ciertos servicios básicos, y sin la necesidad de que todas sus dependencias estén disponibles en Apache Felix. Por ejemplo, Apache Felix Web Console funciona normalmente sin el *bundle* Log Service que acabamos de instalar, pero hay otro *bundle* que si bien tampoco es necesario para que Apache Felix funcione normalmente, es algo incómodo porque genera una error `java.lang.NoClassDefFoundError` en Apache Felix. Como ya se dijo en la Introducción, esto no significa que algo está mal con Apache Felix, sino que de acuerdo con la especificación de la Alianza OSGi, cuando una dependencia opcional de un *bundle* no se encuentra disponible, el *bundle* puede lanzar una excepción.

Para deshacerse de esta molesta excepción, solamente basta con instalar dos *bundles*, uno es 'Apache Felix Configuration Admin Service' y el otro es 'Apache Felix Declarative Services', los cuales se instalan de la misma forma en que instalamos 'Apache Felix Log Service'. Una vez instalados estos dos *bundles* ya no se ve el error `java.lang.NoClassDefFoundError` al iniciar Apache Felix Web Console.

Apache Tomcat

De manera muy similar a la característica de actualización dinámica de OSGi, en Apache Tomcat no es posible lograr extensibilidad y adaptación de manera tal como se ha

demostrado que es posible con OSGi.

Normalmente si se quiere realizar una extensión a una aplicación que resida en Apache Tomcat, este debe ser detenido, el WAR debe ser modificado para incluir tal extensión, y luego ser re-desplegado en Apache Tomcat. Por ejemplo, se puede tener varias ediciones de una misma aplicación: “Simple” y “Avanzada”, en donde la edición “Avanzada” contiene funcionalidades que no están disponibles en la edición “Simple”; y para realizar un *upgrade* solo basta con colocar un archivo JAR dentro del WAR de la aplicación, pero de nuevo nos encontramos con la limitación de que el JAR debe ser parte del archivo WAR, con lo que la definición de “extensibilidad” en Apache Tomcat queda un poco en entredicho.

Otra forma de lograr extensibilidad, la cual no es muy recomendable hacer, es colocar un JAR con los recursos necesarios en el folder CATALINA_HOME/lib (CATALINA_HOME es la ubicación la instalación de Apache Tomcat), que es en donde Apache Tomcat contiene los recursos necesarios (JAR) para el funcionamiento del servidor de aplicaciones como tal. Aún si se toma esta alternativa, Apache Tomcat debe ser detenido y re-iniciado para que cierta aplicación WAR pueda ver y tener acceso a los nuevos recursos que existan en el recién desplegado JAR.

En teoría, es posible desarrollar una aplicación en formato WAR y lograr cierta adaptabilidad, en el sentido de que la aplicación se pueda ejecutar exitosamente dentro de un servidor de aplicaciones como Apache Tomcat, sin necesidad de tener todas sus dependencias satisfechas. Sin embargo, a la hora de extender la aplicación para que dichas dependencias sean satisfechas, y siguiendo buenas prácticas de programación, el archivo WAR debe ser modificado y re-desplegado necesariamente, por lo que esta definición de

"adaptabilidad" también queda en entre dicho, o por lo menos es inferior a las posibilidades de adaptabilidad del ambiente OSGi.

2.7.4 Transparencia y *Laziness*

Folder con ejemplos instalados: felix-transparency

El Apache Felix Web Console permite ver los detalles y por menores de cada *bundle* así como también el estado actual del servidor Web Apache Felix a través de por ejemplo, el servicio de Log Service, aunque Apache Felix también provee una interfaz de línea de comando para inspeccionar al detalle los *bundles* instalados en el Apache Felix.

Por ejemplo, si se observa el *bundle* 'tapestry-person-view', la Ilustración 79 muestra toda la información a la que se tiene acceso. Se puede ver información básica como la versión del *bundle*, y la fecha de la última modificación (instalación o actualización) que se le hizo al *bundle*. Más detalladamente se puede ver cuales son los paquetes que exporta este *bundle* y sus respectivas versiones, y también se puede ver los paquetes que son importados, sus versiones y de que otros *bundles* los obtiene. Por último se puede observar el contenido completo del archivo MANIFEST. En el caso de que este *bundle* registrara servicios en el Registro de Servicios de Apache Felix, también se podría ver en esta vista.

Id	Name	
33	Ejemplo OSGi + Tapestry5 (view) *(utb.osgi.tapestry-person-view)*	
	Symbolic Name	utb.osgi.tapestry-person-view
	Version	1.0.0
	Bundle Location	inputstream:tapestry-person-view-1.0.jar
	Last Modification	Sun Mar 13 09:36:02 EET 2011
	Start Level	1
	Exported Packages	layout,version=0.0.0 layout.images,version=0.0.0 utb.osgi.tapestry.pages,version=0.0.0
	Imported Packages	org.apache.tapestry5,version=0.0.0 from org.extwind.osgi.bundles.tapestry (13) org.apache.tapestry5.annotations,version=0.0.0 from org.extwind.osgi.bundles.tapestry (13) org.apache.tapestry5.ioc.annotations,version=0.0.0 from org.extwind.osgi.bundles.tapestry (13) org.extwind.osgi.tapestry.annotations,version=0.0.0 from org.extwind.osgi.tapestry (18) utb.osgi.tapestry.person.data,version=0.0.0 from utb.osgi.tapestry-person-model (32) utb.osgi.tapestry.person.model,version=0.0.0 from utb.osgi.tapestry-person-model (32)
	Manifest Headers	Bnd-LastModified: 1300001736892 Build-Jdk: 1.6.0_24 Built-By: fayder Bundle-ManifestVersion: 2 Bundle-Name: Ejemplo OSGi + Tapestry5 (view) Bundle-SymbolicName: utb.osgi.tapestry-person-view Bundle-Version: 1.0.0 Created-By: Apache Maven Bundle Plugin Export-Package: utb.osgi.tapestry.pages; uses:="org.apache.tapestry5.annotations, org.extwind.osgi.tapestry.annotations, utb.osgi.tapestry.person.model, utb.osgi.tapestry.person.data", layout, layout.images Import-Package: org.apache.tapestry5, org.apache.tapestry5.annotations, org.apache.tapestry5.ioc.annotations, org.extwind.osgi.tapestry.annotations, utb.osgi.tapestry.person.data, utb.osgi.tapestry.person.model Manifest-Version: 1.0 Tool: Bnd-1.15.0

Ilustración 79: Detalle del bundle 'tapestry-person-view' en Apache Felix Web Console

La característica de Transparencia en la especificación OSGi se refiere a que sea claro como interactúan todos los *bundles* entre sí dentro del contenedor de *bundles*.

Con respecto a la característica de *Laziness*, se puede tomar como ejemplo los *bundles* 'holaServicio' y 'holaConsumidor' que se desarrollaron durante este Trabajo de Grado. Cuando el *bundle* 'holaServicio' es instalado e iniciado en Apache Felix, el servicio que este *bundle* ofrece solamente es registrado en el Registro de Servicios de Apache Felix, pero no se crea ninguna instancia del mismo. Es solamente cuando el *bundle* holaConsumidor busca este servicio, en su clase `Activator` que el servicio como tal es creado para su uso. Esta características, en ambientes OSGi con un tamaño considerable, provee un beneficio en cuanto a consumo de recursos muy significativo.

Apache Tomcat

Como se dijo anteriormente, el servidor de aplicaciones Apache Tomcat posee una herramienta llamada *Manager App*. Esta herramienta también permite obtener cierta información sobre las aplicaciones que han sido desplegadas en el servidor de aplicaciones, e información del propio servidor como tal. Esta herramienta provee cierta transparencia dada la información que provee sobre lo que sucede en contenedor, sin embargo, esta información no es tan granular como la información disponible en un ambiente OSGi, en donde se pude observar información muy detallada de cada *bundle* en el *Framework*.

Con respecto a la característica de *laziness*, tampoco se puede comparar con las posibilidades que brinda el *Framework* OSGi. Es posible lograr algo de *laziness* en aplicaciones que son desplegadas en Apache Tomcat, por ejemplo a través del uso de otros *Frameworks* como Spring. El sistema de Inversión de Control de Spring permite configurar que los *beans* declarados en el Contexto de la Aplicación (*Application Context*) sean instanciados cuando la aplicación necesite de ellos, y no al inicio de la aplicación. Por defecto, Spring crea instancias de las interfaces y clases definidos en el Contexto de la Aplicación una vez la aplicación ha sido iniciada, pero esta configuración pude ser cambiada muy fácilmente, proveyendo así cierto *laziness* para aplicaciones que son desplegadas en Apache Tomcat.

Esta funcionalidad de Spring, también es aplicable a *bundles* OSGi, por lo que la característica de *laziness* en OSGi es mucho más completa, ya que también incluye los beneficios que se demostraron anteriormente con respecto al consumo de servicios en el *Framework* OSGi.

CONCLUSIONES

Con OSGi se puede dividir nuestra aplicación en diferentes *bundles*, los cuales pueden ser instalados, actualizados o eliminados durante tiempo de ejecución sin necesidad de reiniciar el servidor Web; además estos y otros *bundles* de los cuales dependan, tienen, o deberían tener, la capacidad de adaptarse a estos cambios también durante el tiempo de ejecución.

De manera general, la forma de que un archivo JAR normal se convierta en un *bundle* de OSGi a través de un archivo de texto llamado MANIFEST, y obviamente un correcto contenido con las cabeceras necesarias, es una forma para nada invasiva de crear *bundles* OSGi.

Durante los últimos años la tecnología OSGi se ha venido consolidando y su uso ha venido aumentando de manera significativa, especialmente debido a las capacidades que tiene OSGi sobre la modularización, muy ausente en el mundo de Java, y sobre el control de versiones.

Las capacidades de modularidad de OSGi, y las ventajas que tiene sobre la modularidad de la plataforma Java, han quedado claras durante casi todos los ejemplos desarrollados durante este Trabajo de Grado. En múltiples ocasiones se desarrollaron dos o más *bundles*, para crear una aplicación completa, cada *bundle* con una función bastante clara, en especial la capacidad para dividir *bundles* según su responsabilidad dentro de una arquitectura MVC, escondiendo los paquetes que contienen código de implementación y solamente publicando interfaces. Este tipo de modularidad hace que ya no sea algo limitado a aplicaciones en sistemas distribuidos, sino que jueguen un papel muy importante en un diseño orientado a servicios local.

El Control de Versiones es otra gran característica de los *bundles* OSGi, permitiendo el manejo de visibilidad y restricciones a nivel de *bundles* o a nivel de paquetes, algo prácticamente ausente en el mundo de Java. Al esconder efectivamente la implementación

de una aplicación, y hacer que otros *bundles* usen los servicios de otros *bundles* solamente a través de interfaces, se mejora y se asegura la capacidad de cambiar la implementación sin ningún riesgo de que se produzcan problemas de compatibilidad. El Control de Versiones también permite que existan *bundles* y paquetes que provean servicios similares pero con versiones diferentes, y la consecuencia de esto es que en un ambiente OSGi pueden existir aplicaciones que usen el mismo servicio, pero en versiones diferentes, lo cual no es posible en un ambiente no-OSGi. Vale la pena destacar que resulta muy importante tener buenas prácticas con respecto a las versiones de *bundles* y sus paquetes

La extensibilidad y adaptación que provee OSGi son también características muy atractivas, ya que permiten agregar funcionalidades, recursos o servicios a aplicaciones que se encuentren ejecutándose en el contenedor de *bundles*. Además de que se puede hacer en tiempo de ejecución, resulta muy fácil extender aplicaciones con la especificación OSGi.

La capacidad de tener *bundles* en un ambiente OSGi que exporten los mismos paquetes pero en versiones diferentes, es también una ventaja que tiene OSGi sobre los contenedores tradicionales, ya que permite, entre otras cosas, la convivencia de diferentes aplicaciones que de otra forma sería imposible tener en el mismo ambiente.

Uno de los problemas de OSGi es que existe una alta posibilidad de gastar mucho tiempo haciendo *debug* por problemas que se puedan presentar en el *Class Loader* de los *bundles* que se desarrollen.

Otra desventaja de la especificación OSGi, muy señalada por otros autores, es el hecho de que la gran mayoría de librerías de Java no son compatibles con OSGi, es decir no tienen una alternativa *bundle*. Algunas iniciativas como la de SpringSource, proveen muchas librerías en forma de *bundles* de OSGi, pero normalmente estas librerías se encuetran actualizadas. Aunque no es muy difícil transformar una librería que no sea compatible con OSGi en un *bundle*, ya que solamente hay que agregarle un archivo MANIFEST con las cabeceras necesarias, existen muchos *bundles* cuyas cabeceras en el archivo MANIFEST no siguen las mejores prácticas de la especificación OSGi. Por ejemplo, muchos *bundles* no

declaran la versión del *bundle* o peor aún las versiones de los paquetes que hacen públicos, derrotando definitivamente la capacidad de Control de Versiones de OSGi.

En cuanto al desarrollo de aplicaciones Web en OSGi, se concluye que se encuentra todavía en un estado de madurez inferior a las capacidades y opciones disponibles en relación con los ambientes tradicionales no-OSGi. En OSGi el principal protagonista para aplicaciones Web es el *HttpService*, el cual se encarga básicamente de registrar *Servlets* y otros recursos en el Registro de Servicios de OSGi. Considerando solamente esta característica, se puede decir que el desarrollo de aplicaciones Web en OSGi está limitado a las capacidades de *HttpService*. Aunque existen tecnologías que ofrecen otras capacidades y alternativas, tales como Pax Web, Spring-DM y Tapestry OSGi, que se encargan de mejorar la experiencia de desarrollar aplicaciones Web en OSGi, todavía falta mucho para que esta experiencia sea más agradable y suficientemente fácil de usar.

En el caso de la integración de OSGi y Tapestry, si bien la una se beneficia de la otra, una de las consecuencias negativas es que se hace un poco más difícil poder utilizar la característica de Tapestry que consiste en actualizar páginas y recursos en tiempo de ejecución y ver los cambios inmediatamente. Por ejemplo si se está desarrollando una aplicación Web con Tapestry, y se utiliza Jetty[58] para correr la aplicación durante el proceso de desarrollo, es posible hacer modificaciones en las plantillas HTML, o en otros recursos (imágenes, CSS, javascript, etc.) y ver los resultados de esta modificación sin necesidad de reiniciar Jetty. Es posible, pero solamente que hay que realizar unos pasos de más para que pueda ser compatible con el ambiente OSGi también.

El hecho de que exista una especificación formal sobre OSGi, hace que cualquier *bundle* funcione prácticamente de la misma forma en cualquier especificación OSGi. Esto quiere decir que no se está atado a una implementación en específico cuando desarrollamos *bundles* OSGi, ya sea Apache Felix, Equinox o Virgo.

Vale la pena aclarar que durante este Trabajo de Grado no se mostraron todas las

[58] http://jetty.codehaus.org/jetty/

características, posibilidades y aspectos de la especificación OSGi, sino más bien aquellos más representativos. La especificación OSGi tiene mucho más que ofrecer por ejemplo en el ámbito de la Seguridad de aplicaciones, Pruebas o *Testing*, *Debugging,* Manejo de Excepciones y Acceso Remoto, los cuales no fueron abarcados durante este Trabajo de Grado.

RECOMENDACIONES

Durante este Trabajo de Grado no se mostraron todas las características, posibilidades y aspectos de la especificación OSGi, sino más bien aquellos más representativos Se considera interesante investigar sobre otros aspectos relacionados con la tecnología OSGi y se propone:

- Seguridad: La capa de seguridad de la especificación OSGi no fue ilustrado durante este Trabajo de Grado, excepto por una breve introducción. La capa de Seguridad es una capa opcional y se basa en la arquitectura de seguridad de Java 2. Esta capa provee la infraestructura para desplegar y administrar aplicaciones que deben ejecutarse en ambientes con un control bastante granular. La capa de Seguridad de OSGi abarca aspectos tales como:
  - Autenticación de código
  - Archivos JAR digitalmente firmados
  - Permisos
- *Testing*: Una manera de asegurarse de que, durante el desarrollo, una aplicación no se convierta en lo que se denomina "código spaghetti" es a través de Pruebas o *Tests*. Los *tests* pueden ayudar a confirmar que el código que se está desarrollando de forma modular cumple con los requerimientos definidos para la aplicación en cuestión. Los *test* también pueden verificar que el código continuará funcionando después de realizar cambios o re-factorización. Para proyectos ya existentes, que se quieran migrar a OSGi, es necesario o por lo menos recomendable no solamente migrar el código fuente, sino también los *tests*. En el caso de OSGi, el *Framework* Spring-DM, la librería *Apache Commons Pool*, y *OPS4J Pax-Exam* brindan herramientas para desarrollar *tests* en un ambiente OSGi.
- *Debugging*: Una actividad bastante común durante el desarrollo de software es el *debugging*, y para el caso del desarrollo de aplicaciones OSGi, también puede ser my útil. Las aplicaciones OSGi pueden tener múltiples versiones de la misma

clase ejecutándose al mismo tiempo, lo cual requiere un mejor manejo de versiones. Una aplicación OSGi es más difícil de hacerle *debug* utilizando alguna IDE como Eclipse, tal como se hace con muchas aplicaciones hoy en día. Normalmente el *debug* de aplicaciones OSGi ocurre dentro del ambiente en el cual dichas aplicaciones han sido desplegadas.

- Persistencia: Las aplicaciones normalmente necesitan algún tipo de fuente de datos externa, ya sea una Base de Datos o un archivo de texto. Es posible usar la *Java Persistance API (JPA)* para manejar la persistencia en aplicaciones OSGi, y *frameworks* como Spring también proveen herramientas para lograr este objetivo. Incluso se puede usar una herramienta como Hibernate[59] para el manejo de la persistencia en aplicaciones OSGi. La ilustración de la compatabilidad entre OSGi y Hibernate seguramente resulta muy interesante y útil.
- OSGi + Spring MVC: El *Framework* Spring provee muchas herramientas que facilitan el desarrollo de aplicaciones Java, y en el caso de aplicaciones OSGi no es la excepción. Además de Spring-DM, el proyecto Spring MVN resulta muy útil a la hora de desarrollar aplicaciones Web con tecnología OSGi.

[59] http://www.hibernate.org/

FUENTES

APACHE Software Foundation, The. *Maven Getting Started Guide*. [En Línea] Marzo del 2011. Disponible en Internet en la dirección: http://maven.apache.org/guides/getting-started/index.html.

APACHE Software Foundation, The. *Tapestry5 Documentation*. [En Línea] Disponible en Internet en la dirección: http://tapestry.apache.org/documentation.html.

COYLER, Adrian M. et ál. *Spring Dynamic Modules Reference Guide 1.2.1*. SpringSource. [En Línea] Disponible en Internet en la dirección: http://static.springsource.org/osgi/docs/1.2.1/reference/html/.

DOBRIAZKO, Igor. *Tapestry IoC: Modularization of web applications without OSGi*. Tapestry5 [Web log] Artículo del 19 de Enero del 2010. Disponible en Internet en la dirección: http://blog.tapestry5.de/index.php/2010/01/19/tapestry-ioc-modularization-of-web-applications-without-osgi/.

HALL, Richard S. et ál. *OSGi in Action – Creating Modular Applications in Java*. Manning, 2010

OSGi Alliance, The. *OSGi Service Platform Core Specification. Release 4*, Version 4.2. 2009.

OSGi Alliance, The. *OSGi Technology*. [En Línea] Disponible en Internet en la dirección: http://www.osgi.org/About/Technology.

O'BRIEN, Tim, John Casey, Brian Fox, Bruce Snyder, Jason Van Zyl, Eric Redmond. *Maven Definitive Guide*. United States of America: Sonatype, 2008.

PATIL, Sunil. *Hello, OSGi, Part 2: Introduction to Spring Dynamic Modules*. JavaWorld. [En línea] Abril del 2008. Disponible en Internet en la dirección: http://www.javaworld.com/javaworld/jw-04-2008/jw-04-osgi2.html.

RUBIO, Daniel. *Pro Spring Dynamic Modules for OSGi Service Platforms*. United States of America: Apress, 2009.

WALLS, Craig. *Modular Java - Creating Flexible Applications with OSGi and Spring*. United States of America, 2009.

YANG, Donf. *Tapestry-OSGi Screenshots of Key Features*. [En Línea]: Mayo del 2009. Disponible en Internet en la dirección: http://groups.google.com/group/tapestry-osgi/web/tapestry-osgi-screenshots.

GLOSARIO

API (acrónimo de *Application Programmer Interface*): Conjunto de interfaces, funciones y métodos especificadas y proveídas por un subsistema o librería.

Bean: Ver *Java Bean*

Bug (*Computer Bug*): Es el resultado de un fallo o deficiencia durante el proceso de desarrollo o ejecución de un *software*.

Bundle: Este es el nombre que se le da a los componentes OSGI. Un *bundle* es un componente *jar* normal el cual tiene un detallado archivo MANIFEST.MF con una configuración dada por la especificación OSGI. Un ejemplo de un archivo MANIFEST.FM puede ser el siguiente:

[60]
```
Bundle-Name: Hello World Bundle-Symbolic
Name: org.wikipedia.helloworld
Bundle-Description: A Hello World bundle
Bundle-ManifestVersion: 2
Bundle-Version: 1.0.0
Bundle-Activator: org.wikipedia.Activator
Export-Package: org.wikipedia.helloworld;version="1.0.0"
Import-Package: org.osgi.framework;version="1.3.0"
```

Classloader: Es una parte de la MV que dinámicamente carga clases.

Classpath: Se entiende como una opción admitida en la línea de comando, o consola, o mediante variable de entorno que indica a la Máquina Virtual de Java en donde buscar paquetes y clases definidas por el usuario, a la hora de ejecutar programas.

Debugging: Técnica de detección de falta o *bugs* que intenta encontrar una falta producida por una falla no planeada o inesperada, analizando paso a paso diferentes estados del programa.

EJB: API que foma parte del estándar de construcción de aplicaciones empresariales JEE.

Entorno de Desarrollo Integrado: Framework: Conjunto de clases que proveen una solución general que puede ser usada para generar aplicaciones o subsistemas.

IDE (acrónimo de *Integrated Development Environment*): Ver *Entorno de Desarrollo Integrado.*

Java Bean: Modelo de componentes creado por Sun Microsystems para la construcción de aplicaciones Java que encapsulan varios objetos en un único objeto, para hacer uso de un solo objeto en lugar de varios más simples.

---

[60] http://en.wikipedia.org/wiki/OSGi#Bundles

JEE (acrónimo de *Java Enterprise Edition*): Plataforma de programación, que hace parte de la Plataforma Java, y se usa para desarrollar y ejecutar software de aplicaciones Java empresariales.

JVM (acrónimo de Java Virtual Machine): Ver *Máquina Virtual Java.*

Listener: Clase o programa que detecta peticiones o *requests* y que normalmente ejecuta alguna acción dependiendo de esa petición.

Laziness: En programación, se refiere a la técnica de retrasar un comando hasta que sea necesario. También conocido como evaluación perezosa.

Logging: Se refiere a "grabar" o registrar información sobre el estado de una o más aplicaciones en relación al tiempo.

Máquina Virtual Java: Máquina virtual de proceso nativo, capaz de interpretar y ejecutar instrucciones expresados en código binario especial de Java, el cual es generado por un compilador del lenguaje Java.

MVC (acrónimo de *Model-View-Controller*): Patrón de arquitectura de software que separa la interfaz de usuario de la lógica de negocio a través de un tercer componente llamado Controlador.

Plug-in: Es un conjunto de componentes de software que agregan capacidades específicas a otra aplicación.

POJO (acrónimo de Plain Java Object): Es una sigla usada para enfatizar el uso de clases simples y que no dependen de un framework en especial.

Servlet: Objeto que se ejecuta dentro del contexto de un contenedor de *servlets* o servidor Web.