Proyecto de Aplicación Web

María Isabel Alfonso <eli@ua.es>
Domingo Gallardo <domingo.gallardo@ua.es>
Alejandro Such <alejandro.such@ua.es>
Jose Luis Zamora <joseluis.zamora@ua.es>

Tabla de contenidos

1. Presentación

En esta primera sesión vamos a introducir brevemente la plataforma Java EE, la arquitectura del proyecto de aplicación web y vamos a presentar una *guía de laboratorio* en la que introduciremos y practicaremos algunos de los servicios y herramientas que utilizaremos durante el curso: servidores del curso, ordenadores de la UA, máquina virtual, IntelliJ y Bitbucket.

1.1. La plataforma Java EE

La plataforma Java EE (Java Enterprise Edition) es la plataforma Java estándar propuesta por Oracle para el desarrollo de aplicaciones web y aplicaciones empresariales (*enterprise applications*). Nace en el año 2000 con el nombre de Java 2 EE y en sus 15 años de existencia ha evolucionado en 6 distintas versiones hasta llegar a la versión Java EE 7 existente en la actualidad.

¿Qué es una aplicación empresarial o una aplicación web? ¿Cuál es la diferencia fundamental entre estas aplicaciones y las aplicaciones Java de escritorio (Java SE)? Lo veremos con detalle más adelante, pero vamos a adelantar un concepto muy importante: el *servidor de aplicaciones*. La diferencia fundamental entre estos tipos de aplicaciones y una aplicación Java de escritorio (Java SE) es el soporte de ejecución sobre el que corren. Estas últimas corren sobre la JVM mientras que las primeras se ejecutan *dentro* de un servidor de aplicaciones.

> Un *servidor de aplicaciones* es un programa Java que corre sobre la JVM y que da soporte de ejecución a otros programas Java que se *despliegan* en él.

La Máquina Virtual Java (JVM) proporciona a una aplicación Java SE todo el soporte necesario para su ejecución (bibliotecas, ejecución de *bytecodes*, soporte de multi-hilos, etc.). Sin embargo, una aplicación web o una aplicación empresarial necesita para su ejecución una capa más (que también corre sobre la JVM), un programa Java denominado *servidor de aplicaciones*. Este programa ejecuta aplicaciones web o empresariales (que *se despliegan* en él) y da soporte de ejecución (*runtime*) a todo un conjunto de servicios y recursos, como procesamiento de las peticiones HTTP, acceso a bases de datos, a colas de mensajes o gestión de objetos creados por el propio servidor de aplicaciones. Todos estos recursos están a disposición de la aplicación desplegada y del programador que la desarrolla. Veremos más adelante estos conceptos con más detalle.

Existen dos posibles enfoques para definir la arquitectura de una aplicación Java EE.

- En la versión más tradicional de la plataforma Java EE las aplicaciones web están diseñadas como una aplicación multi-capa, con una capa de *frontend* que genera la presentación HTML, una capa intermedia que proporciona seguridad y transaccionalidad y una capa de *backend* que proporciona conectividad a una base de datos o a un sistema heredado (*legacy*). Todo ello en el servidor.
- En la versión mas moderna de Java EE se promueve una arquitectura alternativa en donde la capa de presentación o *frontend* se lleva al cliente web y se implementa en HTML5 y JavaScript. El servidor implementa un servicio REST con el que se comunica el cliente usando el protocolo HTTP. Este servicio REST, implementado con el API de Java EE 7, proporciona todas las funcionalidades de negocio y de acceso a datos. Al separar físicamente la capa de *frontend* de la capa de *backend* la aplicación se hace mucho más modular y más fácil de diseñar y mantener. Este es el estilo que vamos a seguir en el proyecto de aplicación web y en el Experto Java en general.

Aunque Java EE es una propuesta de Oracle, su diseño e implementación es un proceso en el que participan un gran número de empresas e instituciones, formando una comunidad abierta muy dinámica, con muchas aportaciones, propuestas y oportunidades.

La plataforma Java

La definición de la plataforma Java[1] en la Wikipedia en inglés es muy acertada:

> La plataforma Java es el nombre de un conjunto de programas relacionados que permiten desarrollar y ejecutar programas escritos en el lenguaje de programación Java. La plataforma no está ligada a un procesador o sistema operativo, sino a un motor de ejecución (llamado máquina virtual - *Java Virtual Machine*, JVM) y a un compilador con un conjunto de bibliotecas implementadas para distintos sistemas operativos y sistemas hardware, de forma que los programas Java pueden correr de forma idéntica en todos ellos.

Entre los programas que forman parte de la plataforma los dos más importantes son:

- El compilador `javac` que convierte el código fuente Java en ficheros `.class` formados por código Java intermedio o *bytecodes*, idénticos para todas las plataformas
- El intérprete `java` (JVM, *Java Virtual Machine*) que ejecuta los programas *bytecodes* de forma nativa en cada plataforma

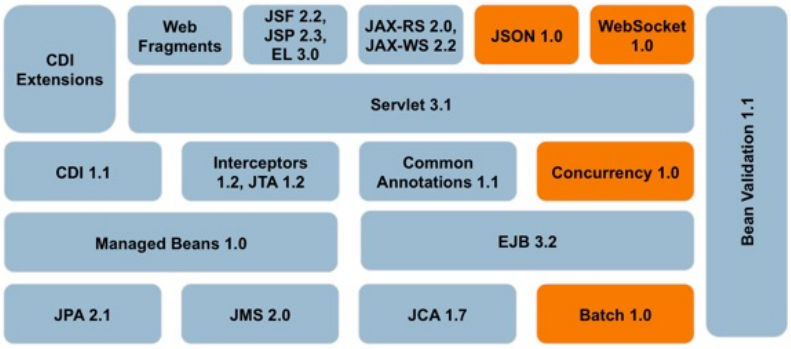
Una característica fundamental de Java es que los ficheros `.class` son, por tanto, **multiplataforma**. Y también lo son los ficheros `.jar` y `.war`, ficheros archivo en el se incluyen múltiples archivos `.class` y que constituyen las bibliotecas de clases Java que se distribuyen e instalan en los distintos sistemas operativos. No es necesario compilar distintas versiones de una determinada biblioteca para los distintos sistemas operativos, sino que la diversidad la trata el soporte de ejecución (la JVM) que interpreta estos ficheros JAR.

Dentro de la JVM hay un denominado *compilador JIT* (*Just In Time*) que optimiza el funcionamiento en tiempo de ejecución del código interpretado. De esta forma se consigue reducir muchísimo la diferencia en tiempo de ejecución entre un programa Java con *bytecodes* interpretado y un programa compilado a código nativo. La introducción de esta técnica en la JVM supuso un gran avance en la adopción de Java y una forma de vencer las críticas sobre la lentitud del código comparado con código nativo. Hoy en día ya casi nadie critica a Java por ese motivo e incluso el enfoque de la máquina virtual que interpreta código intermedio ha sido adoptado también por la plataforma .NET de Microsoft.

Otro elemento fundamental de la plataforma Java es su enorme biblioteca de clases (*class libraries*). Se trata de un conjunto de bibliotecas estándar que proporcionan una gran cantidad de utilidades y funciones para todo tipo de operaciones, como el procesamiento de expresiones regulares, el trabajo con distintos tipos de colecciones, funciones de bajo nivel de entrada-salida o el procesamiento de imágenes. Las librerías se distribuyen como ficheros JAR que se cargan de forma dinámica en el intérprete y a las que nuestras aplicaciones llaman en tiempo de ejecución.

El API de las bibliotecas Java depende de la plataforma de Java (Java SE o Java EE). Toda la biblioteca de funciones de Java SE está disponible en Java EE. La plataforma Java EE define un conjunto adicional de funcionalidades implementadas por más de 20 APIs.

[1] http://en.wikipedia.org/wiki/Java_(software_platform)



Aplicaciones web y servidores de aplicaciones

Las aplicaciones en el servidor necesitan estar funcionando continuamente para responder a todas las peticiones que reciben. Por ejemplo, una aplicación web que recibe peticiones HTTP debe estar continuamente escuchando el puerto (normalmente el 80), recibiendo las peticiones GET o POST de los clientes (navegadores), procesándolas y contestándolas.

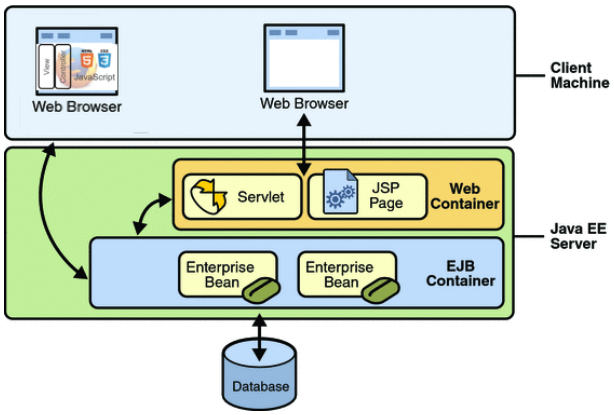
El desarrollo de este tipo de aplicaciones sería complicadísimo si tuviéramos que preocuparnos de todo su ciclo de vida. Por ejemplo, sería un infierno tener que implementar nosotros todo el bucle de procesamiento de las peticiones que llegan al servidor, utilizando hilos concurrentes que gestionan la entrada-salida y que lanzan las llamadas a los métodos que realizan el procesamiento.

En la plataforma Java EE se utiliza la idea de *contenedor* para solucionar este problema. El contenedor es la aplicación Java que se encarga de realizar todo el trabajo de infraestructura (como escuchar los puertos para recibir las peticiones) y que delega el procesamiento final en las aplicaciones (*componentes*) *desplegadas* en él. El desarrollador implementa estos componentes escribiendo las interfaces de las clases que se utilizan para definir los componentes. Un ejemplo concreto es el funcionamiento de los *servlets* que veremos en la primera asignatura del curso.

En palabras de Arun Gupta (en su libro *Java EE 7 Essentials*):

> Los componentes Java se despliegan en contenedores que proporcionan el soporte de su ejecución (*runtime*). Los contenedores proporcionan las APIs Java EE subyacentes a los componentes de aplicación. Las aplicaciones componentes Java EE nunca interaccionan directamente con otras aplicaciones componente. Utilizan protocolos y métodos del contenedor para interactuar entre ellas y con los servicios de la plataforma. La interposición de un contenedor entre las aplicaciones componentes y los servicios Java EE permite al contenedor inyectar transparentemente los servicios requeridos por el componete, como gestión declarativa de las transacciones, comprobaciones de seguridad, *pooling* de recursos y gestión del estado.

Los servidores de aplicaciones ofrecen tambien un conjunto de servicios relacionados con la alta disponibilidad, como son tolerancia a fallos, concurrencia o clustering.



Historia de Java EE

En la actualidad Sun proporciona tres grandes distribuciones (o ediciones):

Java ME (Java Micro Edition)
: Para el desarrollo de aplicaciones Java en pequeños dispositivos (móviles, tarjetas de crédito, bluetooth, televisiones o reproductores blu-ray).

Java SE (Java Standard Edition)
: Para el desarrollo de aplicaciones de escritorio en ordenadores personales.

Java EE (Java Enterprise Edition)
: Para el desarrollo de aplicaciones distribuidas (cliente-servidor o con múltiples capas) como aplicaciones web o servicios web.

La historia de las versiones de estas distribuciones Java es la siguiente:

Enero de 1996
: **JDK 1.0**. Lanzamiento de JDK (Java Development Kit) 1.0, la primera versión del lenguaje.

Febrero de 1997
: **JDK 1.1**. lanzamiento de JDK 1.1, que incluía la primera versión de JDBC y de RMI (llamadas a objetos remotos).

Diciembre de 1998
: **J2SE 1.2**. En el primer cambio de nombre, la plataforma pasa a llamarse J2SE (Java 2 Platform, Standard Edition). La versión inicial de la distribución SE es la 1.2 (para mantener la numeración de versiones consistente con la del JDK). Se introducen importantes cambios en el lenguaje y en la plataforma. Se introduce la API Swing para el desarrollo de interfaces de usuario.

Diciembre de 1999
: **J2EE 1.2**. Aparece la primera versión de Java Enterprise, que incluye: JSP, Servlets, JDBC, EJB, JMS, JTA y JavaMail.

Mayo de 2000
: **J2SE 1.3**. Mejora la eficiencia de Java con la máquina virtual HotSpot.

Septiembre de 2001
: **J2EE 1.3**. Segunda versión de Java Enterprise, en la que se mejora el rendimiento de los EJB (EJB 2.0) y se introducen nuevas versiones de las APIs como JSP 1.2 o servlets 2.3.

Febrero de 2002
: **J2SE 1.4**. Se introducen APIs para tratar XML (JAXP), seguridad y criptografía (JCE, JSSE, JAAS). Se incluye Java Web Start para la distribución remota de aplicaciones Java de escritorio.

Noviembre de 2003

J2EE 1.4. Nuevas versiones de las APIs: EJB 2.1, JSP 1.3, Servlets 2.4, JDBC 3.0. Se introducen por primera vez las librerías para los servicios Web.

Septiembre de 2004

J2SE 1.5. Importantes cambios en el lenguaje: genéricos, anotaciones, enumeraciones o iteración.

Mayo de 2006

Java EE 5. Otro cambio de nomenclatura de la plataforma, junto con un gran cambio en bastantes APIs. Se elimina el 2 después de la palabra Java y se elimina el 1 del número de versión. Se introduce la especificación 3.0 de los EJB con anotaciones, uso de persistencia (JPA) y timers. Nuevas versiones de APIs: JSP 2.1, Servlets 2.5, JDBC 4.0. Se introduce JSF y mejoras en los servicios Web.

Diciembre de 2006

Java SE 6. Se incluye el cambio de nomenclatura que elimina el 2 después de Java. Mejoras en el rendimiento de Swing. Mejoras: Servicios web en Java SE, scripting (soporte para Python y Ruby), Java DB (base de datos basada en Apache Derby).

Julio de 2008

Java SE 7. Lenguajes dinámicos en la JVM. Nueva librería de entrada/salida. Mejoras en el intérprete/compilador HotSpot.

Diciembre de 2009

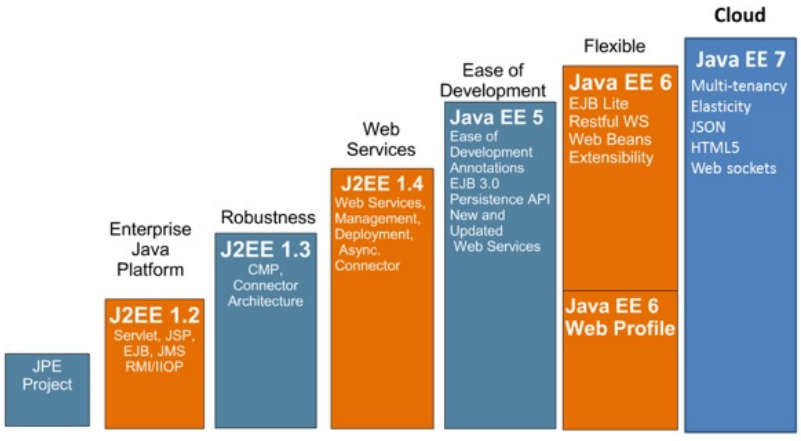
Java EE 6. Se introduce el perfil Web, con la intención de popularizar Java EE no sólo para el desarrollo de aplicaciones enterprise, sino también aplicaciones web sencillas. En esta línea, se define la versión reducida de EJB, EJB Lite, el API de servicios REST y los *Web Beans*.

Marzo de 2014

Java SE 8. Grandes cambios en el lenguaje: expresiones lambda, anotaciones.

Junio de 2014

Java EE 7. Se mejora el perfil Web, con énfasis en servicios REST, JSON, *Web sockets* y conexión con HTML5/JavaScript. En la parte enteprise, se profundiza en el despliegue en la nube y en el concepto de *platform as a service*.



Detalles de la plataforma Java EE 7

La plataforma Java EE 7 se define en la JSR 342[2]. En la especificación se definen dos *perfiles*: el perfil web y el perfil completo. El perfil web está orientado a aplicaciones web básicas que

[2] https://www.jcp.org/aboutJava/communityprocess/final/jsr342/index.html

no necesitan componentes transaccionales distribuidos. Las APIs que debe ofrecer el servidor de aplicaciones que soporta este perfil son las siguientes, agrupadas por la funcionalidad que proporcionan:

- Anotaciones:
  # Common Annotations for the Java Platform (JSR-250) 1.2
- Procesamiento de peticiones HTTP y ciclo de vida de la aplicación web:
  # Servlet 3.1
  # Managed Beans 1.0
  # Interceptors 1.2
  # Contexts and Dependency Injection for the Java EE Platform 1.1
- Generación de páginas HTML:
  # JavaServer Pages (JSP) 2.3
  # Expression Language (EL) 3.0
  # Standard Tag Library for JavaServer Pages (JSTL) 1.2
  # JavaServer Faces (JSF) 2.2
- Servicios REST y APIs avanzadas de comunicación con clientes:
  # Java API for RESTful Web Services (JAX-RS) 2.0
  # Java API for JSON Processing (JSON-P) 1.0
  # Java API for WebSocket (WebSocket) 1.0
- Capa de lógica de negocio y de acceso a bases de datos:
  # Bean Validation 1.1
  # Enterprise JavaBeans (EJB) 3.2 Lite
  # Java Persistence API (JPA) 2.1
  # Java Transaction API (JTA) 1.2

Veremos una gran parte de estas APIs a lo largo de la primera parte del curso.

La plataforma completa Java EE 7 añade a las APIs anteriores otro conjunto de características, orientadas sobre todo al soporte de aplicaciones transaccionales distribuidas:

- Enterprise JavaBeans (EJB) 3.2 completo
- Java Message Service (JMS) 2.0
- JavaMail 1.5
- Connector 1.7
- Web Services 1.4
- Concurrency Utilities 1.0
- Batch 1.0
- Procesamiento XML con JAXB 2.2
- Java EE Management 1.1
- Java Authorization Contract for Containers (JACC) 1.5
- Java Authentication Service Provider Interface for Containers (JASPIC) 1.1
- Web Services Metadata 2.1

Todas estas especificaciones se encuentran detalladas en distintas especficaciones denominadas JSRs (*Java Specification Request*). Cada JSR tiene un número de identificación y puede ser consultada en el sitio web Java Community Process[3]. Todas las especificaciones de APIs sufren un largo proceso desde que comienza su propuesta hasta su aprobación final en una votación en la que participan todos los *partners* interesados en el API.

Por ejemplo, ahora mismo (el 22 de septiembre) se acaba de aprobar (con 24 votos a favor y ninguno en contra) que comience el desarrollo de la JSR 366[4] que definirá la nueva especificación de Java EE 8. Todas las JSRs de APIs de la plataforma Java EE (incluidas las ya retiradas) se encuentran en este enlace[5]. Y, por ejemplo, la especificación de Java EE 7 se define en la JSR 342[6] que tardó dos años en ser completada.

> Si la especificación de Java EE 8 tarda lo mismo en terminarse que la de Java EE 7, la nueva versión de la plataforma llegará a finales de 2016 o comienzos de 2017.

Todas estas especificaciones sirven para que empresas independientes de Oracle (y también Oracle) desarrollen servidores de aplicaciones que compitan en rendimiento y funcionalidades adicionales, pero que sean todos compatibles entre si. El servidor GlassFish de Oracle es un servidor gratuito que sirve de plataforma de prueba de las especificaciones.

En la página de Oracle de compatibilidad de servidores de aplicaciones Java EE[7] se listan los servidores de aplicaciones que han pasado las pruebas necesarias para obtener una compatibilidad con una determinada especificación Java EE. Existen muy pocos servidores de aplicaciones totalmente compatibles con Java EE 7, tanto en el perfil Web como en el perfil completo:

- GlassFish (Oracle, perfiles web y completo)
- Wildfly (anteriormente JBoss, RedHat, perfiles web y completo)

Por su mayor antiguedad, hay más servidores compatibles con Java EE 6:

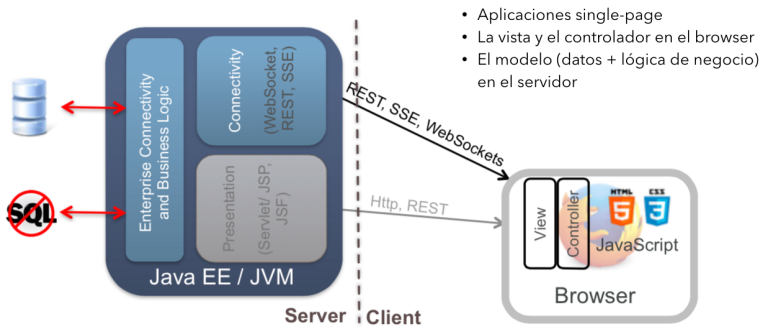
[3] https://www.jcp.org/en/home/index
[4] https://www.jcp.org/en/jsr/detail?id=366
[5] https://www.jcp.org/en/jsr/platform?listBy=3&listByType=platform
[6] https://www.jcp.org/en/jsr/detail?id=342
[7] http://www.oracle.com/technetwork/java/javaee/overview/compatibility-jsp-136984.html

- GlassFish (Oracle, perfiles web y completo)
- WebLogic (Oracle, perfil completo)
- Wildfly (anteriormente JBoss, RedHat, perfiles web y completo)
- WebSphere (IBM, perfil completo)
- Geronimo (Apache, perfil completo)
- TomEE (Apache, perfil web)

El servidor de aplicaciones que vamos a utilizar en el curso es *Wildfly*, uno de los más avanzados, ligeros y competitivos. Está desarrollado por Red Hat y es muy posible que lo conozcas con el nombre que tenía anteriormente: *JBoss*. Es un servidor gratuito que tiene la posibilidad de contratar licencias comerciales para su uso en entornos que requieran un funcionamiento continuo y una disposición inmediata para solución de incidencias.

1.2. Arquitectura del proyecto de aplicación web

En los últimos años está ganando cada vez más popularidad la arquitectura REST como enfoque para construir las aplicaciones web. De hecho, el temario del curso de experto y sus asignaturas está muy influenciado por esta filosofía. Y la aplicación web que vamos a implementar a lo largo del curso va a tener precisamente esta arquitectura.



¿En qué consiste una arquitectura REST? ¿Cuáles son sus ventajas? ¿Cuál es la razón de su popularidad? En el curso vamos a dedicar una asignatura completa a hablar de cómo construir APIs REST utilizando la tecnología Java JAX-RS. Pero vamos a avanzar ahora algunos aspectos básicos de esta arquitectura.

Aplicaciones web basadas en REST

Ideas fundamentales:

- Separación clara de responsabilidades entre del cliente y el servicio: el cliente se encarga de la interfaz de usuario y el servicio de la lógica de negocio. El cliente suele ser un navegador web o una aplicación móvil en el que se ejecuta el programa que interactua con el usuario, recoge sus peticiones y las envía al servicio. El servicio realiza toda las operaciones relacionadas con el procesamiento de la petición. Recoge los datos de la petición, accede a la base de datos, procesa los resultados y los devuelve al cliente.
- Se utiliza el protocolo HTTP como base de la comunicación entre el cliente y el servidor: el cliente realiza peticiones utilizando los métodos GET, POST, PUT o DELETE para indicar el tipo de acción a realizar y las URLs como identificadores de los recursos sobre los que se

está haciendo la petición. El servidor recibe la petición y devuelve un código de respuesta HTTP y el contenido de la respuesta.

- La comunicación entre el cliente y el servidor se realiza usando texto estructurado. Las peticiones llevan los parámetros en las cabeceras HTTP o en las URL del recurso al que se está accediendo. Y los resultados se devuelven en formato texto XML o JSON.

Servicio REST

El servicio REST proporciona el *backend* de la aplicación. Lo implementamos con la tecnología Java EE, que proporciona APIs para definir las distintas capas de la aplicación:

- API JAX-RS para implementar el API REST de la aplicación
- Objetos EJB **lite** para definir la lógica de negocio
- JPA/Hibernate para definir la capa de modelo y la conexión con la base de datos

Aplicación cliente JavaScript

La aplicación cliente se ejecuta en dispositivos separados del servidor (navegadores web o dispositivos móviles), construye la interfaz con la que interactúan los usuarios y se comunica con el servicio REST usando HTTP.

La tendencia que está ganando cada vez más fuerza es construir la aplicación cliente como una aplicación JavaScript que corre en un navegador web usando algún *framework* de alto nivel como AngularJS o BackboneJS. La aplicación cliente lanza peticiones HTTP al servicio y *pinta* la interfaz de usuario y la rellena con los resultados de las peticiones.

1.3. Guía de laboratorio

En esta sesión de ejercicios vamos a describir los aspectos más importantes de las distintas plataformas, utilidades y sitios web necesarios para el desarrollo de las prácticas del Experto.

Máquina VirtualBox Linux Ubuntu
: Las prácticas se desarrollarán en una máquina virtual (MV) VirtualBox basada en Linux Ubuntu. Disco SSD externo de 120 GB. Ponemos a vuestra disposición un disco SSD de 120 GB de alta velocidad, de más de 400 MB/s de escritura y lectura, el Samsung 850 EVO[8]. El disco está instalado en una caja USB 3.0 StarTech[9] con cable integrado. En el disco se guarda una imagen de la máquina VirtualBox con todo el software necesario para desarrollar las prácticas (fichero `Experto_Java_2015.vdi`). El disco se encuentra formateado con el sistema de ficheros exFAT, para que sea posible guardar ficheros de más de 4GB (límite de FAT32). Este sistema de ficheros está soportado también por MacOS a partir de su versión 10.6.5.

Cuentas Bitbucket
: Profesores y estudiantes usaremos cuentas en Bitbucket[10] para guardar los repositorios git de los proyectos que se van desarrollando a lo largo del curso. La cuenta común bitbucket/java_ua[11] guardará los repositorios iniciales (con ejemplos y *plantillas* para los ejercicios) de los distintos módulos del curso. Una vez terminado el módulo y entregados y corregidos los ejercicios se añadirá un repositorio con las soluciones. Las cuentas de los estudiantes serán privadas y contendrán los repositorios desarrollados por cada uno. Servirán como copia de seguridad del trabajo realizado y se utilizará también para realizar las entregas de los ejercicios de cada módulo.

Apuntes y materiales docentes
: Todos los apuntes, transparencias y materiales docentes están disponibles en una zona restringida de la web de apuntes del Experto (http://expertojava.ua.es).

Moodle
: Se utilizará la plataforma Moodle del Campus Virtual de la UA para la interacción on-line. Usaremos principalmente sus funcionalidades de foros y de puntuaciones de las entregas de ejercicios.

A continuación vamos a detallar el uso de estos elementos y algunos otros también necesarios para el desarrollo de las prácticas del Experto.

Materiales docentes

Los apuntes, trasparencias y demás material docente se encuentran en un sitio web restringido a los alumnos del Experto. Puedes acceder a los materiales de cada módulo desde los enlaces disponibles en las páginas públicas de cada uno de las asignaturas del experto, accesibles desde el menú superior de las páginas públicas del Experto:

- Componentes Web[12]
- JPA - Framework de persistencia[13]

[8] http://www.samsung.com/global/business/semiconductor/minisite/SSD/global/html/ssd850evo/overview.html
[9] http://es.startech.com/Almacenamiento-Datos/Cajas/Caja-de-disco-duro-USB-3-con-UASP~S2510BMU33CBS
[10] https://bitbucket.org
[11] https://bitbucket.org/java_ua
[12] http://expertojava.ua.es/experto/publico/web-expertojava.html
[13] http://expertojava.ua.es/experto/publico/jpa-expertojava.html

- Componentes Enterprise[14]
- Servicios REST[15]
- Servidores Web y PAAS[16]
- Lenguaje JavaScript[17]
- Frameworks JS[18]
- AngularJS[19]
- Bases de datos NOSQL[20]
- Framework Grails[21]
- Proyectos de aplicación Web[22]

Cuando intentes acceder a la zona restringida del Experto Java aparecerá una página pidiéndote tu usuario y contraseña. El usuario será tu nombre de usuario facilitado por la Universidad para tu correo `@alu.ua.es`.

Si deseas cambiar tu contraseña (o no la recuerdas), puedes pulsar en el enlace correspondiente. Se enviará un mensaje a tu dirección de correo con un enlace con el que podrás modificar la contraseña.

Moodle

La Universidad de Alicante utiliza dos plataformas on-line para la interacción docente: Campus Virtual y Moodle. En nuestro título vamos a utilizar Moodle por considerarlo bastante más potente, flexible y fácil de utilizar que el Campus Virtual.

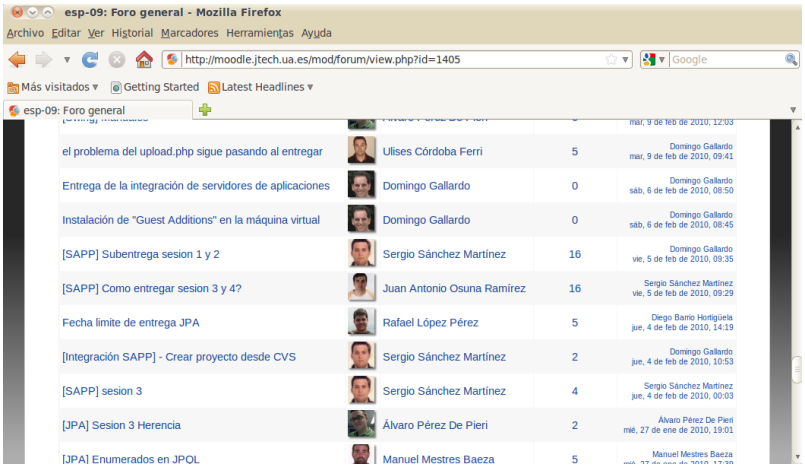
Para acceder a Moodle debes *logearte* en UACloud seleccionar la aplicación Moodle UA:

14 http://expertojava.ua.es/experto/publico/ejb-expertojava.html
15 http://expertojava.ua.es/experto/publico/rest-expertojava.html
16 http://expertojava.ua.es/experto/publico/paas-expertojava.html
17 http://expertojava.ua.es/experto/publico/js-expertojava.html
18 http://expertojava.ua.es/experto/publico/backbone-expertojava.html
19 http://expertojava.ua.es/experto/publico/angularjs-expertojava.html
20 http://expertojava.ua.es/experto/publico/nosql-expertojava.html
21 http://expertojava.ua.es/experto/publico/grails-expertojava.html
22 http://expertojava.ua.es/experto/publico/proyint-expertojava.html

Vamos a utilizar Moodle como plataforma de trabajo colaborativo. La usaremos para gestionar los foros, las calificaciones de los ejercicios y alguna que otra encuesta que iremos presentando.

Uno de los elementos principales que utilizaremos de Moodle es el foro, que utilizaremos para resolver dudas que puedan surgir sobre el desarrollo de los ejercicios de las asignaturas o sobre cualquier tema relacionado con el título. Cualquiera puede publicar una entrada nueva en el foro o contestar a las ya existentes. Cada nueva entrada o contestación genera un correo electrónico que se envía a todos los profesores y estudiantes.

Para poder utilizar correctamente el foro es muy importante que actualices tu foto. Para esto debes pinchar en *Administración* > *Ajustes de mi perfil* > *Editar perfil*) y seleccionar *Imagen del usuario*. Allí puedes colocar tu foto. Cuando todos tenemos la foto es mucho más sencillo contestar e interactuar en el foro. La siguiente imagen muestra un ejemplo del foro general en la edición 2009-2010 del experto:



En la página principal del curso podrás encontrar también:

- Enlaces a las tareas de entregas de ejercicios de cada una de las asignaturas
- Calendario de clases y entregas de ejercicios
- Enlaces a los apuntes y materiales de la asignatura en curso

Ordenadores de la UA

Las clases se impartirán en un laboratorio gestionado por la EPS (los viernes, el laboratorio L17) y en un laboratorio gestionado por el Servicio de Informática (los sábados, en el Aula Mac (Aula INF1/BG) de la Biblioteca General).

En los ordenadores de la EPS seleccionaremos el sistema operativo *Windows*. Los ordenadores del Aula Mac son iMacs con el sistema operativo Mac OS. En ambos sistemas operativos se encuentra el programa VirtualBox con el que se pondrá en marcha la MV en la que se realizarán las prácticas.

Trabajaremos directamente con la imagen de la MV en el disco externo SSD. Incluso en ordenadores con USB 2.0 el rendimiento será aceptable. En los ordenadores con USB 3.0 podrás aprovechar completamente la velocidad del disco SSD.

> ¡Cuidado con tus datos! Debes tener precaución con el disco externo. Si le sucediera algo a la máquina virtual perderías todo lo hecho en el curso. Por ello debes tener cuidado de copiar regularmente la máquina virtual en tu ordenador de casa y de subir a Bitbucket los repositorios con los proyectos Java.

Máquina virtual Virtual Box

Uno de los elementos más importantes de las prácticas del curso es la MV con una distribución de Linux Lubuntu y con las herramientas necesarias para realizar los ejercicios. Su uso te hace sencillo continuar en casa los ejercicios y prácticas realizados en clase y garantiza que todos utilizamos el mismo entorno de trabajo. También nos hace inmunes a posibles cambios en las instalaciones de los ordenadores de la universidad.

El disco imagen de la MV original se encuentra en el disco externo, y también comprimida en la zona restringida de apuntes de la web del Experto y dividida en 3 ficheros ZIP de unos 900 MB cada uno:

- Experto_Java_2015-16.vdi.z01[23]
- Experto_Java_2015-16.vdi.z02[24]
- Experto_Java_2015-16.vdi.zip[25]

[23] http://expertojava.ua.es/j2ee/restringido/software/Experto_Java_2015-16.vdi.z01
[24] http://expertojava.ua.es/j2ee/restringido/software/Experto_Java_2015-16.vdi.z02
[25] http://expertojava.ua.es/j2ee/restringido/software/Experto_Java_2015-16.vdi.zip

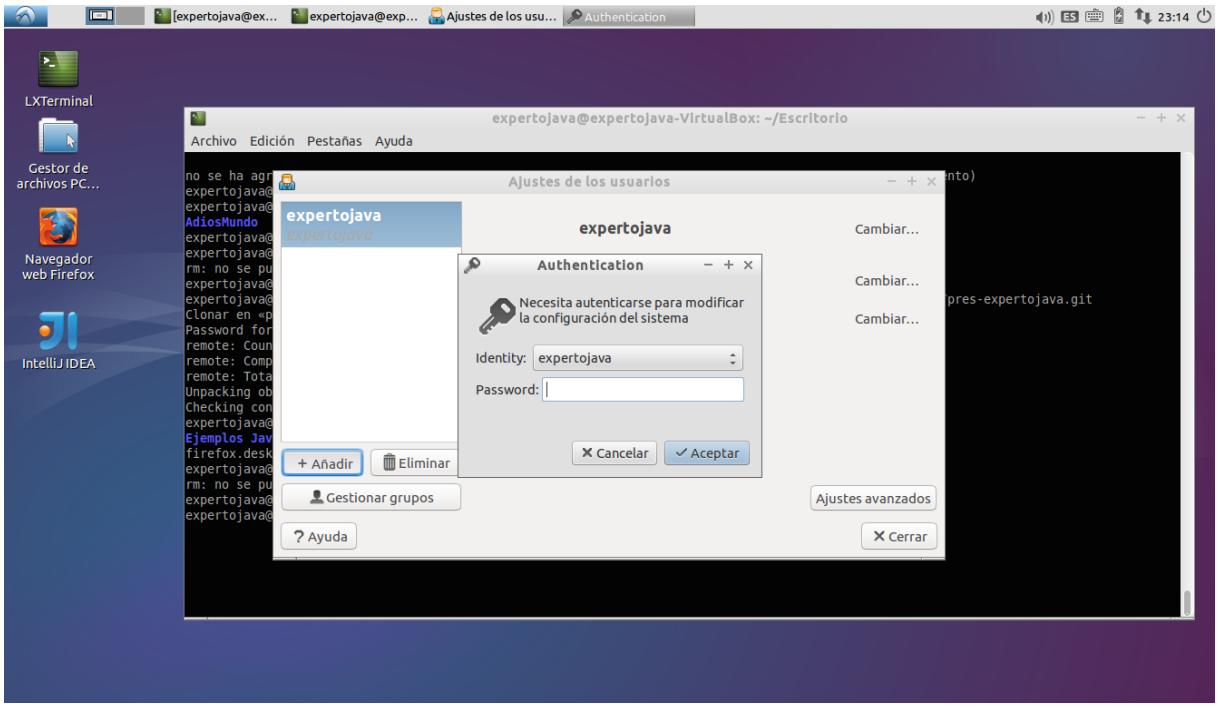
La MV Ubuntu es compatible con las últimas versiones VirtualBox. La versión de las Guest Additions instalada en la MV es la 4.3.30.

VirtualBox es multiplataforma y opensource. Existen versiones para Windows, Mac y Linux y es posible trabajar con la misma máquina virtual en distintos sistemas operativos. Puedes, por ejemplo, trabajar en la EPS en Windows y después continuar el trabajo en el Aula Mac. Puedes bajar la última versión e instalarla en tu ordenador de casa desde la web de VirtualBox[26]. Una vez instalado el programa VirtualBox, debes instalar también el *VirtualBox Extension Pack* que proporciona soporte para hardware adicional en el ordenador anfitrión, como el USB 2.0.

Máquina virtual Lubuntu

En la máquina virtual está instalada la versión 14.04 de Lubuntu[27] de 64 bits, una versión más ligera del sistema operativo Ubuntu que usa una interfaz de usuario mínima y funcional basada en LXDE.

En la MV se ha creado el usuario `expertojava` con la contraseña `expertojava`. Tendrás que utilizar este login para entrar en el sistema, para ejecutar comandos en modo superusuario o cuando se bloquee la pantalla:



El disco de la máquina virtual tiene una capacidad máxima de 60 GB. Las aplicaciones instaladas ocupan inicialmente alrededor de 8 GB.

En la máquina virtual se ha instalado el software que vamos a utilizar a lo largo de todos los módulos del curso:

- **Plataforma Java** 1.8.0_60
- **Entorno de desarrollo** IntelliJ IDEA 14.1.4 con licencia de desarrollo para la Universidad de la Alicante

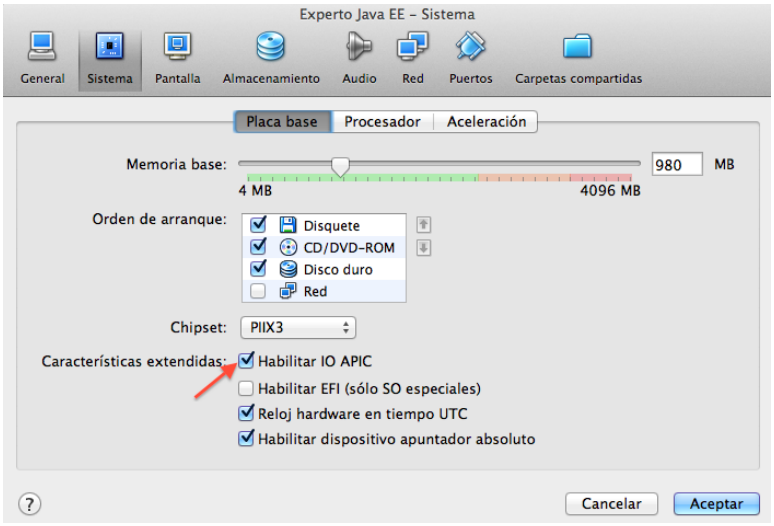
[26] https://www.virtualbox.org/wiki/Downloads
[27] http://lubuntu.net

- **Servidor de aplicaciones** JBoss WildFly 8.2.1 Final
- **Bases de datos**: MySQL 5.5 (contraseña del usuario root: `expertojava`) y MongoDB
- **Navegadores**: Mozilla y Chrome
- Editor de textos avanzado Atom
- Node y npm
- Docker
- Herramientas: Maven, Git, Curl

Paso a paso: creación de la MV Ubuntu en el ordenador anfitrión

Lo primero que tenemos que hacer es crear con VirtualBox la máquina virtual en la que vamos a trabajar. Vamos a crear la MV para que trabaje con la imagen vdi en el disco externo (`Experto_Java_2015.vdi`).

1. Arranca VirtualBox en el ordenador anfitrión y crea una nueva MV de tipo **Linux Ubuntu (64 bits)** con el nombre `Experto Java`.
2. Define el tamaño de la memoria de la MV en un valor suficiente para trabajar cómodamente con el entorno de trabajo y que no comprometa el rendimiento del ordenador anfitrión. Los ordenadores de la EPS tienen 4 GB de memoria y 2 GB está en límite de lo recomendable. Pondremos alrededor de 2 GB.
3. Ahora debemos vincular la imagen vdi del disco externo con la máquina virtual que estamos creando. Para ello, en la pantalla Disco Duro Virtual seleccionamos la opción *Usar un archivo de disco duro virtual existente* y seleccionamos el fichero `Experto_Java_2015.vdi` en el disco duro externo.
4. Terminamos configurando el número de procesadores de la MV. Es muy recomendable trabajar con al menos 2 procesadores, porque el rendimiento aumenta muchísimo. Para ello debemos seleccionar la opción *Configuración* > *Sistema* > *Habilitar IO APIC*:



Y después definir más de 1 procesador en la pestaña de *Procesador*.

La configuración de la máquina virtual creada se guarda en la carpeta `VirtualBox VMs` del directorio de usuario en el ordenador anfitrión. Como en los ordenadores de la universidad

se restauran los discos duros frecuentemente, deberás repetir esto cada vez que empiece la sesión de prácticas. En tu ordenador de casa, bastará que lo hagas una vez.

Instalación de Guest Additions

Es recomendable instalar las *Guest Additions*. Con ellas instaladas es posible pasar del SO invitado (lubuntu) al SO anfitrión sin tener que pulsar ninguna combinación de teclas, sólo moviendo el cursor. También son útiles para copiar y pegar texto entre ambos sistemas operativos, así como para cambiar fácilmente la resolución de la pantalla.

Las *Guest Additions* ya están instaladas en la imagen inicial. Si en algún momento actualizas VirtualBox o lubuntu, deberás también volver a instalar Guest Additions. Para ellos debes seleccionar la opción *Dispositivos* > *Insertar Guest Additions CD Image* del menú de Virtual Box que aparece cuando estamos ejecutando la MV. Esto montará en lubunutu un disco con distintos comandos disponibles. Deberás abrir un terminal y ejecutar:

```
$ cd /media/expertojava/VBOXADDITIONS_<version>
$ sudo ./VBoxLinuxAdditions.run
```

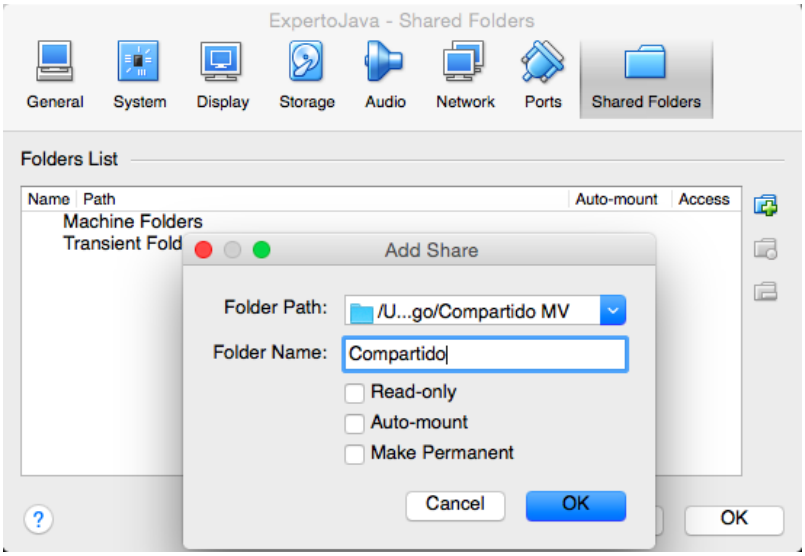
Una vez instaladas, debes desmontar el CD y reiniciar lubuntu.

Compartición de directorios con el anfitrión

Una vez instaladas las *Guest Additions* es posible compartir directorios entre el ordenador invitado (lubuntu) y el anfitrión (Windows, Mac, etc.). Para ello selecciona la opción *Dispositivos* > *Directorios Compartidos* y pulsa en el icono para añadir un nuevo directorio transitorio (no se guardan los datos de un arranque a otro).

Aparecerá una ventana en la que debes indicar la ruta del directorio del ordenador anfitrión que se quiere compartir y un nombre simbólico con el que identificar ese directorio. Para indicar la ruta del directorio en el anfitrión puedes también escoger la opción del desplegable que abre el navegador de archivos para seleccionarlo gráficamente.

Crea el directorio `Compartido MV` en el ordenador anfitrión, escógelo y escribe como nombre simbólico `Compartido`.



De esta forma estamos creando un dispositivo que puede ser montado en el sistema y que tiene como nombre `Compartido` y que estará conectado con el directorio `Compartido MV`_ en el ordenador anfitrión.

Por último debemos crear un directorio en la MV que haga de punto de montaje del dispositivo que acabamos de crear. Lo podemos llamar con cualquier nombre, por ejemplo `Host`. Y después usamos el comando mount como root para montarlo:

```
$ cd Escritorio
$ mkdir Host
$ sudo mount -t vboxsf -o uid=1000,gid=1000 Compartido Host
```

Los parámetros *uid=1000,gid=1000* hacen que el propietario de la carpeta compartida sea el propio usuario, con lo que no es necesario ser root para explorarla.

Para desmontar el directorio llamamos al comando *umount*:

```
$ sudo umount Host
```

Git y Bitbucket

Vamos a desarrollar todos los proyectos del experto utilizando el sistema de control de versiones distribuido Git. Con este sistema de control de versiones trabajaremos sobre un repositorio local y tendremos una réplica en un sitio remoto. Iremos confirmando (*commit*) todos los cambios que vamos realizando sobre el código fuente de nuestros proyectos en el repositorio local y luego *subiremos* (*push*) estos cambios al repositorio remoto. El repositorio remoto servirá de copia de seguridad y para compartir el código con los profesores.

Para la creación de los repositorios remotos utilizaremos el servicio *Bitbucket*[28].

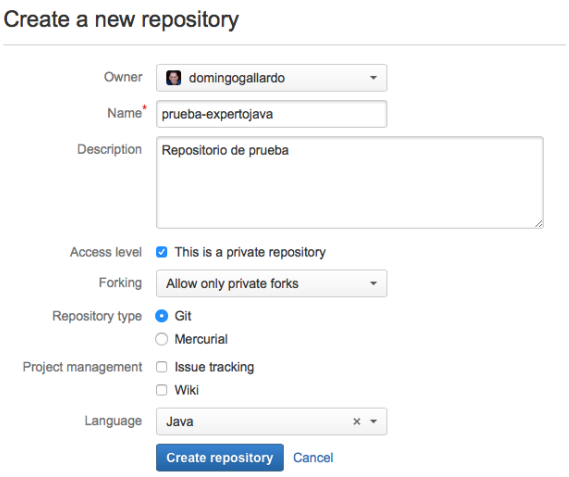
Vamos a ver en cómo crear nuestro primer repositorio git. Podemos hacerlo primero en remoto y después bajarlo a nuestro ordenador o al revés.

Creación de un repositorio remoto en bitbucket

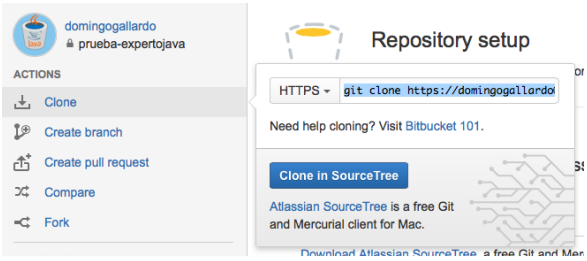
Vamos a ver cómo crear un repositorio privado en bitbucket *Bitbucket*[29] que vincularemos con nuestro repositorio local.

1. En primer lugar, deberemos crearnos una cuenta personal en Bitbucket[30]. Créala con el mismo nombre de usuario que tu login de la UA. Una vez creada la cuenta te añadiremos al grupo `Estudiantes ExpertoJava` del *team* java_ua[31] y podrás acceder a los repositorios creados por los profesores para las distintas asignaturas del Experto.
2. Creamos desde nuestra cuenta de bitbucket un repositorio (*Repositories* > *Create repository*).
3. Deberemos darle un nombre al repositorio, por ejemplo `prueba-expertojava`. Será de tipo Git y como lenguaje especificaremos **Java**.

[28] https://bitbucket.org/
[29] https://bitbucket.org/
[30] https://bitbucket.org
[31] https://bitbucket.org/java_ua/



4. Una vez hecho esto, veremos el repositorio ya creado, en cuya ficha podremos encontrar la ruta que nos dará acceso a él.



Será útil copiar la dirección anterior para vincular con ella nuestro repositorio local al remoto. Veremos como hacer esto en el siguiente apartado.

Creación del repositorio git local

Tenemos dos alternativas para crear un repositorio local vinculado al remoto:

- Clonar el repositorio remoto, lo cual inicializa un repositorio local en el que ya está configurado el vínculo con el remoto.
- Crear un repositorio local independiente, y vincularlo posteriormente con un repositorio remoto.

Para realizar cualquiera de estas dos alternativas hay que utilizar comandos de Git. Es posible hacerlo desde el IDE *IntelliJ*, pero es mucho más útil aprender a trabajar con Git desde línea de comandos. Así podremos utilizar los comandos en cualquier entorno y no dependeremos de tener instalado un entorno gráfico que es mucho más pesado que un sencillo terminal.

Creación a partir del repositorio remoto

La forma más sencilla de crear un repositorio Git local es hacerlo directamente a partir del repositorio remoto. Si ya tenemos un repositorio remoto (vacío o con contenido) podemos clonarlo en nuestra máquina local con:

```
$ git clone https://<usuario>:bitbucket.org/<usuario>/prueba-expertojava
```

Este comando podemos copiarlo directamente desde bitbucket, tal como hemos visto en el último paso del apartado anterior (opción *Clone* de la interfaz del repositorio).

De esta forma se crea en nuestro ordenador el directorio `prueba-expertojava` y se descarga en él el contenido del proyecto, en caso de no estar vacío el repositorio remoto. Además, quedará configurado como repositorio git local y conectado de forma automática con el repositorio git remoto del que lo hemos clonado.

Creación de un repositorio local y vinculación con el remoto

Esta forma es algo más compleja que la anterior, pero será útil si tenemos ya creado un repositorio git local de antemano, o si queremos vincularlo con varios repositorios remotos.

Para la creación de un repositorio git local seguiremos los siguientes pasos.

1. Creamos un directorio local y nos movemos a él:

```
$ mkdir prueba-expertojava
$ cd prueba-expertojava
```

1. Inicializamos el repositorio git. Estando en la raíz del directorio `prueba-expertojava` hacemos:

```
$ git init
```

1. Conectamos el repositirio local con el remoto. En bitbucket veremos la URL que identifica el repositorio, que será del tipo: `https://<usuario>@bitbucket.org/<usuario>/presentacion-expertojava.git`. Desde el directorio raíz del proyecto ejecutamos:

```
$ git remote add origin https://<usuario>@bitbucket.org/<usuario>/prueba-
expertojava.git
```

De esta forma añadimos añadiendo un repositorio remoto llamado `origin` (el nombre por defecto del repositorio remoto en git) conectado al local. Hemos inicializado nuestro directorio como un repositorio local git y lo hemos conectado con el repositorio remoto situado en Bitbucket.

Registrar cambios en el repositorio

Independientemente de cuál de los métodos anteriores hayamos utilizado para inicializar nuestro repositorio git local, lo habremos conectado con el repositorio remoto de Bitbucket.

Vamos a ver ahora cómo trabajar con un repositorio git.

En primer lugar será recomendable añadir un fichero `.gitignore` al directorio del proyecto, que dependerá del tipo de proyecto y que se encargará de excluir del control de versiones todos aquellos tipos de ficheros que sean generados automáticamente (por ejemplo las clases compiladas). Podemos encontrar diferentes modelos de `.gitignore` en: (https://github.com/github/gitignore)

Tras añadir el `.gitignore` correcto para nuestro tipo de proyecto podremos añadir nuevos ficheros, registrarlos en el sistema de control de versiones y confirmar los cambios que realicemos.

Antes de añadir ningún cambio al repositorio debemos inicializar en la máquina linux las variables de git `user.name` y `user.email` para que quede registrado el usuario que hace los commits. El `user.email` debe coincidir con el correo electrónico registrado en Bitbucket.

```
$ git config --global user.name "Pepito Pérez"
$ git config --global user.email pepito.perez@gmail.com
```

A partir de ahora todos los cambios se registrán como realizados por ese usuario. A partir de ahora cada vez que queramos registrar cambios en el repositorio local deberemos:

1. Si hemos añadido nuevos archivos al proyecto, deberemos añadirlos al sistema de control de versiones con `git add`:

```
$ echo "Hola mundo" > hola-mundo.txt
$ git add .
```

1. Podemos confirmar los cambios realizados y añadirlos al control de versiones con el comando `git commit -a -m` (o haciendo *commit* desde el IDE). Es obligatorio añadir un mensaje con una explicación del cambio realizado:

```
$ git commit -a -m "Primer fichero en el repositorio"
```

1. Git almacena los cambios confirmados en el repositorio local. En este caso hemos añadido un fichero llamado `hola-mundo.txt`. Cuando queramos subir un conjunto de cambios al repositorio remoto deberemos hacer un *push* para subir al repositorio *origin* (en Bitbucket):

```
$ git push -u origin master
```

1. Al hacer `- u` indicamos que la rama master local está haciendo *tracking* de la rama master en origin. A partir de ahora sólo será necesario hacer `git push` para subir los cambios.
2. Editamos con algún editor sencillo el fichero `hola-mundo.txt` y añadimos un par de líneas más. Hacemos después un *commit*:

```
$ gedit hola-mundo.txt
$ git status  # Comprobamos los cambios sin confirmar
$ git commit -a -m "Añadidas un par de líneas en hola-mundo.txt"
```

Cada *commit* representa un punto del desarrollo al que podríamos volver con el comando `git checkout <commit-id>` para examinar esa versión o crear nuevas ramas.

Para listar todos los *commits* realizados podemos hacer:

```
$ git log --oneline
```

Se listan el identificador del *commit* y su comentario.

Por último, podemos volver a hacer `git push` para subir los cambios al repositorio:

```
$ git status
$ git push
```

Compartición de repositorio

Bitbucket permite compartir un repositorio con otros usuarios. Vamos a utilizar esta características para compartir las plantillas iniciales de los ejercicios y para realizar las entregas de los mismos:

- Los profesores compartirán con vosotros los repositorios iniciales de cada módulo, a partir de los que se comenzaréis a realizar los ejercicios del módulo. Deberéis hacer una copia de propia haciendo un *fork* en la cuenta personal de Bitbucket.
- Una vez que hayáis terminado de realizar los ejercicios, en la fecha de entrega de la asignatura, deberéis dar permiso de lectura al repositorio al profesor que ha impartido el módulo. Se puede hacer desde *Settings* > *Access management*:

Creación de proyectos con IntelliJ

Vamos a realizar una rápida introducción a la creación de proyectos y la sincronización con git y Bitbucket usando *IntelliJ*, el IDE que utilizaremos a lo largo de todo el curso.

En IntelliJ es importante diferenciar entre *proyecto* y *módulo*. El *proyecto* es el directorio principal de trabajo de IntelliJ, en el que se guarda la configuración de los distintos elementos que vamos creando en el entorno. Puede constituir un proyecto Java único, con sus clases, sus bibliotecas, sus ficheros de configuración, etc. O también puede contener más de un *módulo*, subdirectorios que constituyen subproyectos independientes pero que pueden compartir ciertos elementos situados en el proyecto principal.

Los proyectos y módulos del entorno se corresponden con directorios del sistema operativo y guardan la configuración en ficheros XML. La información de un proyecto se guarda en el directorio oculto `.idea` dentro del directorio con el nombre del proyecto. Por ejemplo, si creamos el proyecto `prueba` se creará un directorio con el mismo nombre que contendrá el directorio `.idea`. Lo podemos comprobar desde el terminal:

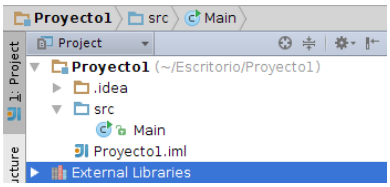
```
$ cd prueba
$ ls -la
```

Los que conocen Eclipse pueden ver un proyecto IntelliJ como un *workspace* de Eclipse, con la diferencia de que en Eclipse no es posible usar un *workspace* como un proyecto con código fuente.

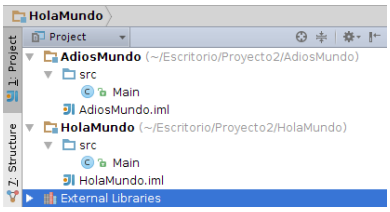
Cada módulo se guarda como un directorio con el nombre del módulo, en el que se crea un fichero `<modulo>.iml` con la configuración del módulo.

En IntelliJ podremos crear tres tipos de proyectos:

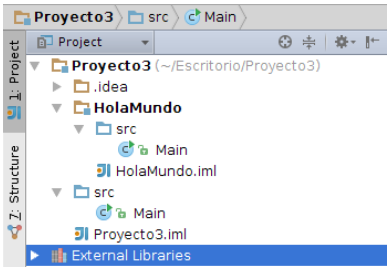
1. Proyectos que contienen únicamente código fuente, sin incluir ningún módulo adicional:



2. Proyectos vacíos que contienen distintos módulos:



3. Proyectos que contienen código fuente y que además incluye módulos adicionales:

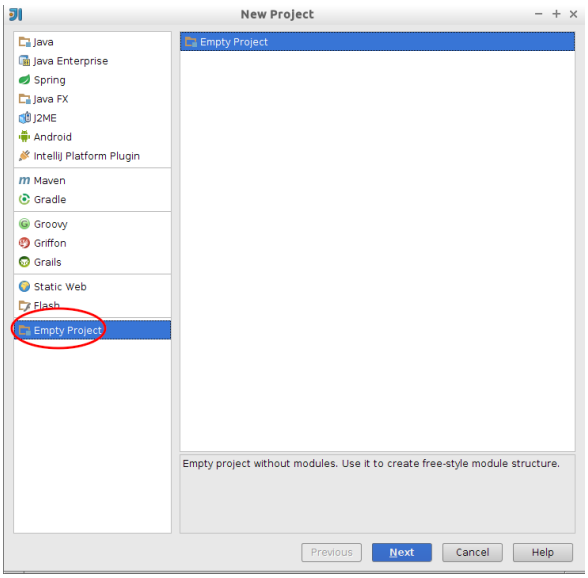


En todos los casos anteriores se puede definir un repositorio git en el proyecto principal, que contenga todos sus elementos (ya sea código fuente y/o otros subproyectos).

Vamos a empezar con un ejemplo del segundo tipo de proyectos: un proyecto vacío que contiene un par de módulos (programas Java) y que sincronizaremos posteriormente con Bitbucket. Lo hacemos paso a paso.

Paso a paso: creción de un proyecto git con varios módulos en IntelliJ

1. Crea un repositorio `pres-expertojava` en tu cuenta de Bitbucket.
2. Abre IntelliJ y crea un proyecto vacío con el mismo nombre `pres-expertojava` en cualquier directorio, por ejemplo el escritorio.



3. Se habrá creado un directorio nuevo con ese nombre. Vamos ahora a un terminal. Es más fácil inicializar git desde línea de comando que desde IntelliJ. Crea el fichero `.gitignore` con un editor, por ejemplo `atom`:

```
$ cd Escritorio/pres-expertojava
$ atom .gitignore
```

4. Copia el siguiente código:

Fichero .gitignore:

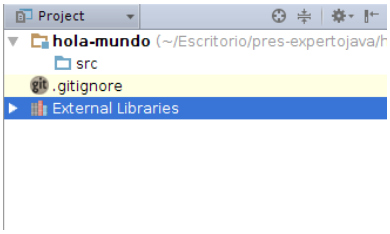
```
# IntelliJ
out/
.idea/workspace.xml

# Maven output
target

#OS X stuff
.DS_Store
```

5. Inicializa git en el directorio, añade los ficheros al repositorio y conéctalo con el repositorio remoto en Bitbucket:

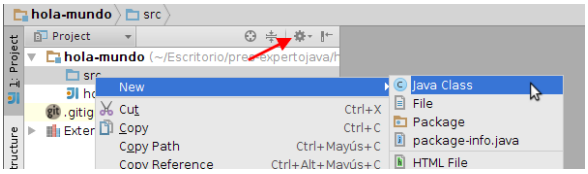
```
$ git init
$ git add .
$ git commit -m "Creado el repositorio"
$ git remote add origin https://<usuario>@bitbucket.org/<usuario>/pres-
expertojava.git
$ git push -u origin master
```

Si ejecutas el comando `ls -la` verás que se ha creado un directorio oculto `.git` en el que se guarda la configuración del repositorio git creado (en el fichero `.git\config`) y todos los *commits* que se realicen en el repositorio local.

6. Ahora ya podemos cambiar a IntelliJ. Una vez creado el repositorio es cómodo realizar los *commits* desde el IDE. Creamos un nuevo módulo Java dentro del proyecto, con la opción *File > New Module…*. Pichamos *Next* y le damos el nombre al nuevo módulo: `hola-mundo`:



7. Creamos una nueva clase `HolaMundo` pinchando con el botón derecho sobre el directorio `src` y seleccionando *New > Java Class*:



Aunque lo habitual es crear las clases Java dentro de *packages*, en este caso no lo hemos hecho para simplificar el ejercicio. En el experto vamos a nombrar los paquetes siempre empezando por `org.expertojava`. Para trabajar con paquetes es recomendable seleccionar la opción *Flatten Packages* que hay en la rueda dentada en la parte superior derecha del panel de proyectos.

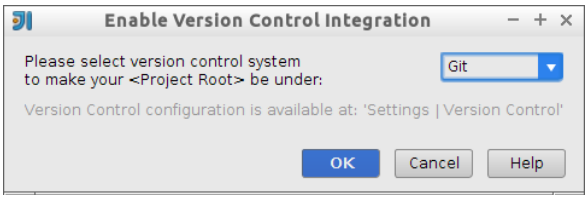
8. Le damos a la clase como nombre `HolaMundo` y escribimos el típico código *Hola mundo*:

```
public class HolaMundo {
    public static void main(String[] args) {
        System.out.println("Hola, mundo\n");
    }
}
```

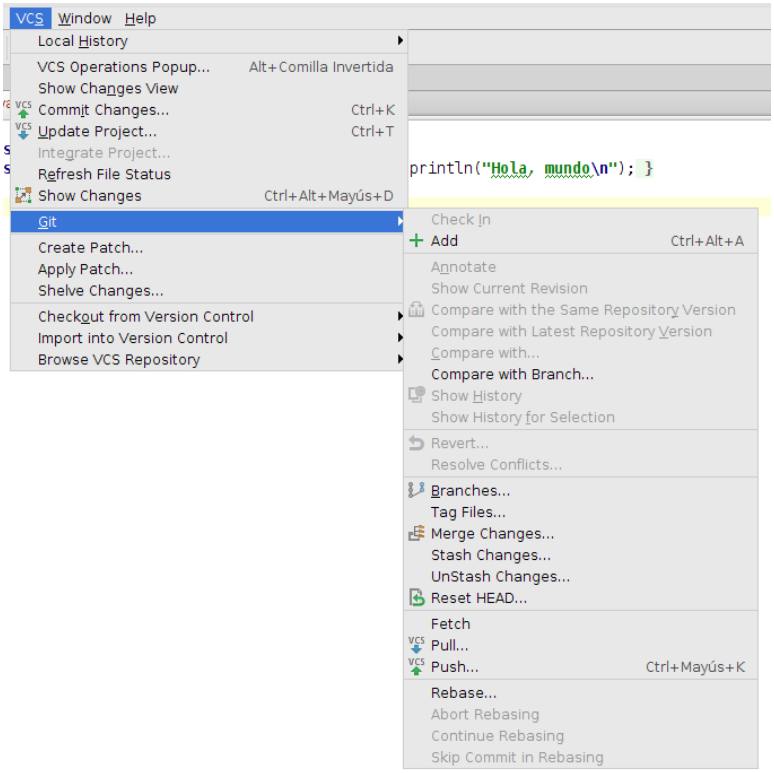
En IntelliJ es posible utilizar **abreviaturas** que se expanden en código. Por ejemplo, escribe `psvm` y pulsa el tabulador. Verás que se expande en la plantilla de la función `main`. Puedes también escribir `sout` y pulsar el tabulador. Verás que se expande en una plantilla para hacer un `System.out.println`.

Estas abreviaturas se denominan *Live Templates*. Puedes consultar, modificar y añadir nuevas plantillas seleccionando *File > Settings > IDE Settings > Live Templates*.

9. IntelliJ todavía no ha detectado que hemos inicializado git en el repositorio. Para ello hay que seleccionar *VCS > Enable Version Control Integration…* y seleccionar `Git`.

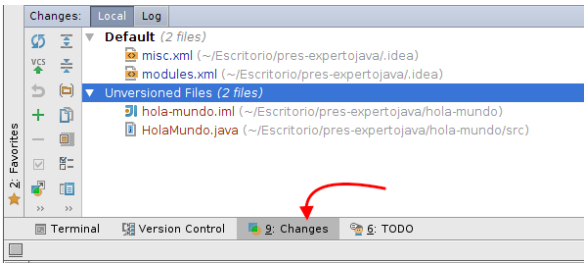


El menú *VCS* (*Version Control System*) es muy importante, ahí se encuentran todas las opciones relacionadas con Git.



Verás que automáticamente en el panel del proyecto cambia el color de los ficheros que no están confirmados y que aparecen nuevas opciones en el entorno.

10 Abriendo el panel inferior *Changes* podemos gestionar los cambios pendientes de confirmar del proyecto

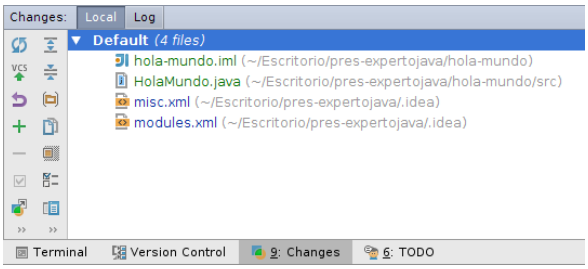


El color del nombre de un fichero indica su estado en el control de versiones:

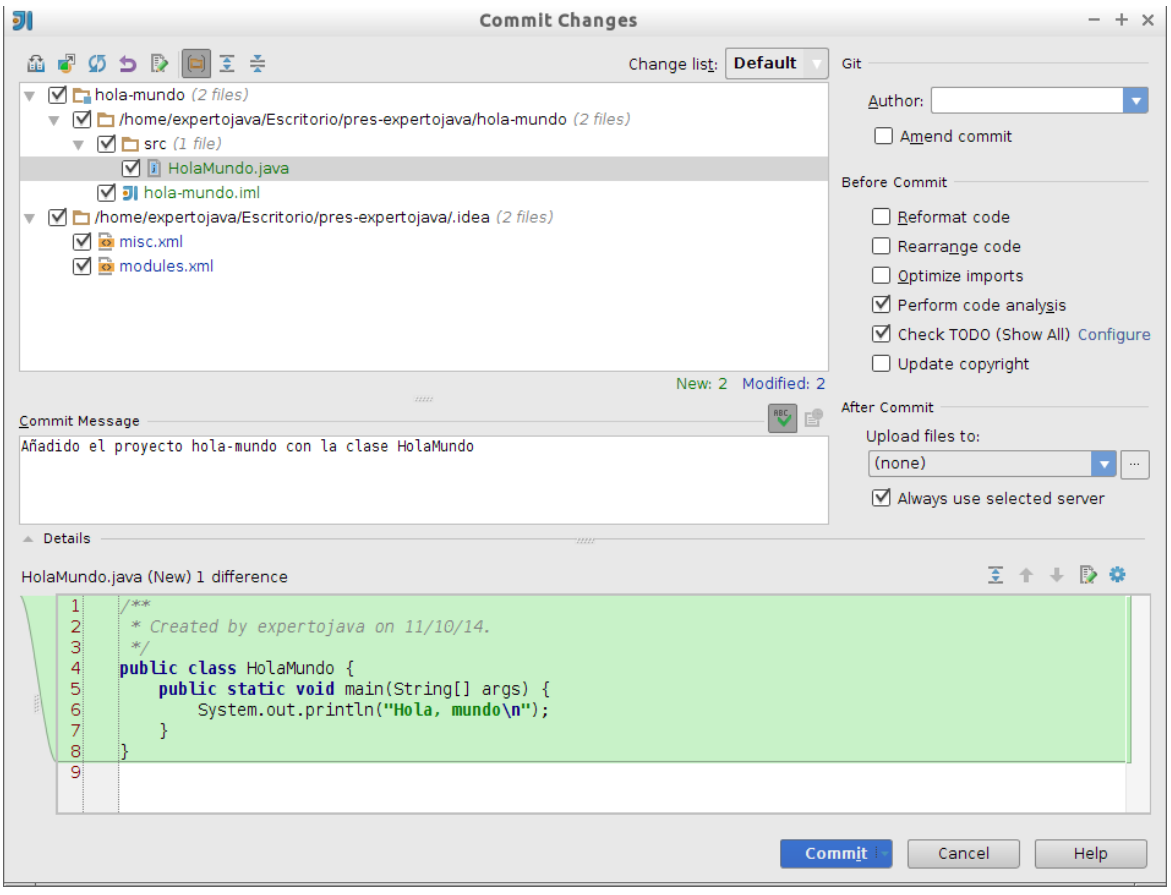
- Rojo: Fichero sin añadir
- Verde: Fichero que será añadido en el siguiente commit

- Azul: Fichero con cambios pendientes de confirmar
- Negro: Fichero sin cambios

11. Pulsamos el botón derecho sobre *Unversioned Files* y seleccionamos la opción *Add to VCS* (o pulsamos *Ctrl+Alt+A*). Los ficheros se añadirán a la lista de cambios (*Changelist*) Default:



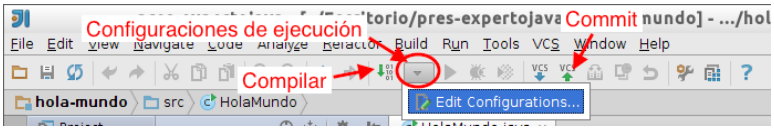
12. Pulsamos con el botón derecho *Commit* y aparecerá una ventana en la que podemos revisar los cambios, añadir un comentario y confirmarlos



También podemos realizar un *commit* de los cambios en un fichero o en un directorio:

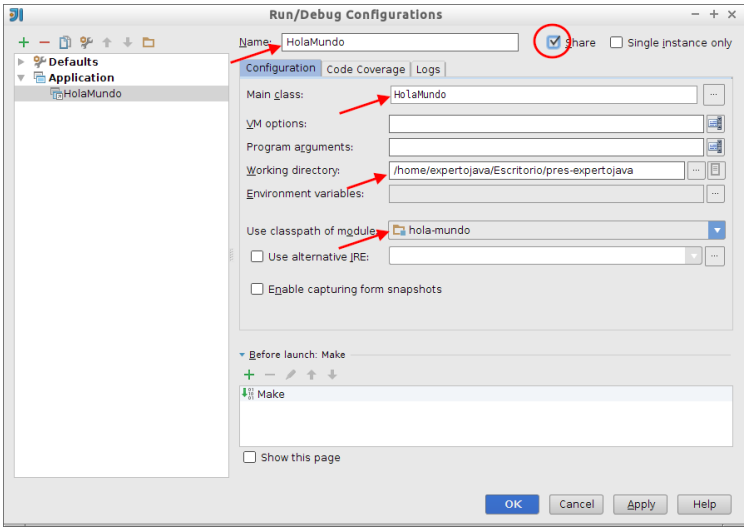
- Desde el menú *VCS*, seleccionando el fichero o directorio en el panel del proyecto y seleccionando la opción *VCS > Git > Commit File…*.
- Pulsando *Ctrl+K*

13. Vamos a terminar creando una configuración de ejecución para ejecutar el programa y subiendo todos los cambios a Bitbucket. Para crear una configuración de ejecución pulsa en el desplegable junto al botón *Play* (ahora inactivo) y selecciona *Edit Configurations…*

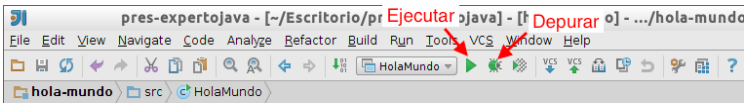


14. Aparece una ventana para gestionar las configuraciones de ejecución. Crea una nueva configuración de ejecución pulsando el símbolo + en la esquina superior izquierda y seleccionando *Application*. Rellena los siguientes datos:

    - *Share*: chequeado (para que la configuración se guarde en el control de versiones)
    - *Name*: HolaMundo
    - *Main class*: HolaMundo (puedes seleccionarla con el botón de la derecha del campo)
    - *Working directory*: /home/expertojava/Escritorio/pres-expertojava (aparece por defecto)
    - *Use classpath of module*: hola-mundo

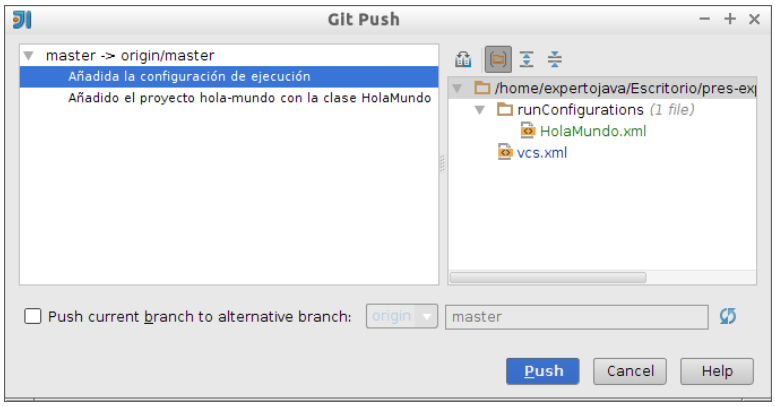


    Aparecerá un aviso preguntando si queremos añadir al control de versiones la configuración de ejecución. Decimos que sí. Ya podemos ejecutar o depurar el programa:

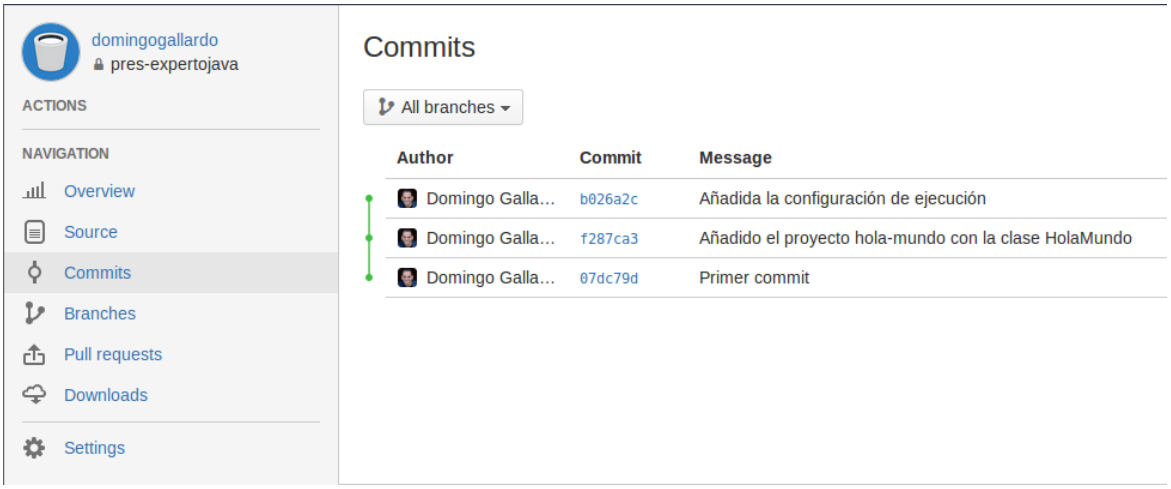


15. Ejecutamos el programa, comprobando que aparece el panel con la salida en la parte inferior de la ventana. Confirmamos los cambios.

16. Por último, subimos (*push*) los cambios al repositorio en Bitbucket. Lo podemos hacer seleccionando *VCS > Git > Push…* o *Ctrl+Mayús+K*. Aparece una ventana que nos permite revisar por última vez los cambios que vamos a subir al repositorio remoto:



17. Comprobamos en el navegador que se han subido los cambios correctamente a Bitbucket:

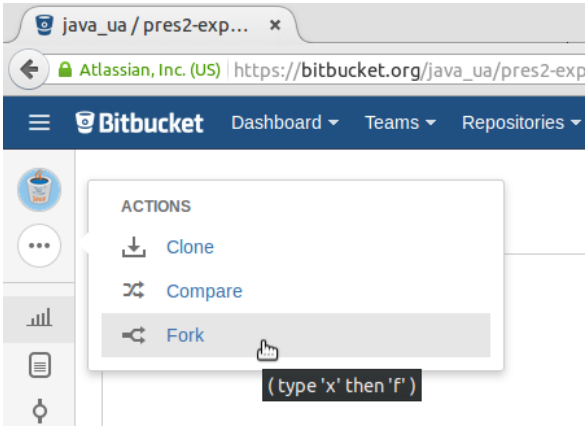


18. Ahora que ya has aprendido a crear proyectos y módulos en IntelliJ y trabajar con git y bitbucket, crea un nuevo módulo denominado `saludo` que imprima por la salida estándar `Hola, soy <mi nombre>`. Crea una configuración de ejecución y sube todo a Bitbucket.

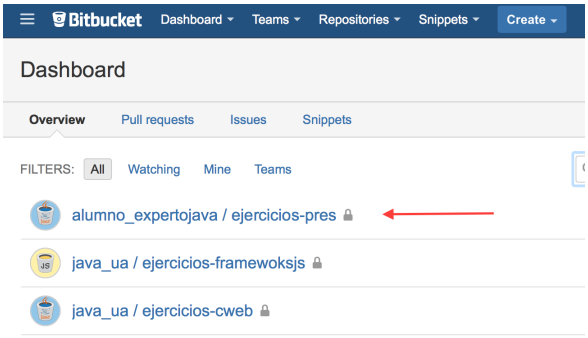
Paso a paso: importar plantillas de bitbucket

Vamos a terminar explicando cómo trabajar con proyectos dejados en Bitbucket por los profesores, que servirán como plantillas de los ejercicios a realizar en la asignatura.

1. En tu cuenta de Bitbucket verás el repositorio `java_ua/ejercicios-pres`. Ese repositorio está en la cuenta `java_ua`, usada por los profesores del experto para dejar repositorios accesibles a los alumnos. Allí dejaremos los repositorios con las plantillas de ejercicios de las asignaturas y con las soluciones, una vez concluida la fecha de entrega. Los repositorios tienen sólo permiso de lectura para los estudiantes.
2. Copia el repositorio en tu cuenta, haciendo un *fork* del mismo. Para ello entra en el repositorio, pincha en los puntos suspensivos que hay en la esquina superior izquierda y escoge la opción *Fork*



El repositorio se copia en tu cuenta:



3. Descárgalo en tu ordenador desde línea de comando. Ve al directorio en el que quieras descargarlo, por ejemplo `Escritorio`, y haz un `git clone`:

```
$ cd Escritorio
$ git clone https://<usuario>@bitbucket.org/<usuario>/ejercicios-
pres.git
```

4. Abre el proyecto en IntelliJ seleccionando la opción *File* > *Open*.

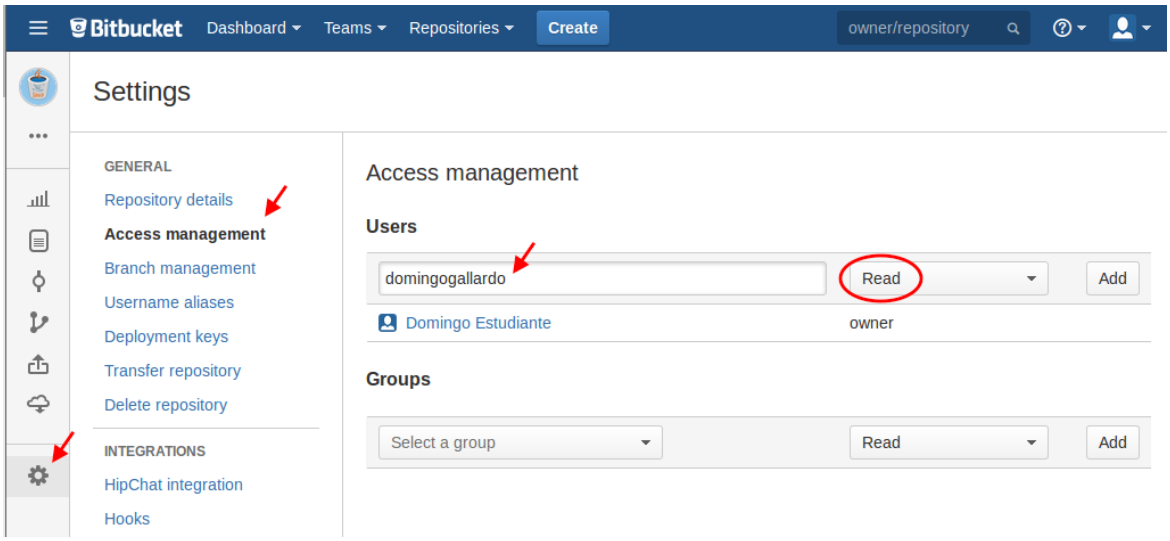
5. El proyecto se llama `agenda` y contiene la siguiente clase `org.expertojava.pres.Tarjeta` que define una tarjeta de contacto de una agenda, con un nombre y un correo electrónico de tipo `String`, y un identificador de tipo `Integer`:

```
package org.expertojava.pres;

public class Tarjeta {
    Integer id;
    String nombre;
    String eMail;

    public Tarjeta(String nombre, String eMail) {
        this.nombre = nombre;
        this.eMail = eMail;
    }

    public void setId(Integer id) {
```

```
        this.id = id;
    }

    public Integer getId() {
        return id;
    }

    public String getNombre() {
        return nombre;
    }

    public String geteMail() {
        return eMail;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;

        Tarjeta tarjeta = (Tarjeta) o;

        if (!eMail.equals(tarjeta.eMail)) return false;
        if (id != null ? !id.equals(tarjeta.id) : tarjeta.id !=
null) return false;
        if (!nombre.equals(tarjeta.nombre)) return false;

        return true;
    }

    @Override
    public int hashCode() {
        int result = nombre.hashCode();
        result = 31 * result + eMail.hashCode();
        return result;
    }

    @Override
    public String toString() {
        return "Tarjeta{" +
                "id=" + id +
                ", nombre='" + nombre + '\'' +
                ", eMail='" + eMail + '\'' +
                '}';
    }
}
```

6. Debes completar la clase `Main` escribiendo un guarde cuatro tarjetas en un `ArrayList` y después lo recorra e imprima las tarjetas en la salida estándar. Cuando funcione correctamente haz un *commit* y sube los cambios a Bitbucket.

7. Para terminar debes compartir los repositorios creados en la sesión con el profesor de la asignatura. En Bitbucket selecciona la configuración del repositorio pulsando el botón *Settings*, la rueda dentada que hay abajo a la izquierda. Pulsa la opción *Access management* y añade al profesor de la asignatura con permiso de lectura:


Bitbucket
Dashboard
Teams
Repositories
Create
owner/repository
Settings
GENERAL
Repository details
Access management
Branch management
Username aliases
Deployment keys
Transfer repository
Delete repository
INTEGRATIONS
HipChat integration
Hooks
Access management
Users
domingogallardo
Read
Add
Domingo Estudiante
owner
Groups
Select a group
Read
Add

2. (1,5 puntos) Caso de estudio

2.1. Introducción a Maven

Maven es una herramienta Java de gestión del proceso de construcción de proyectos software, que simplifica la complejidad de sus distintas partes: compilación, prueba, empaquetamiento y despliegue. Es una herramienta muy popular en proyectos open source que facilita:

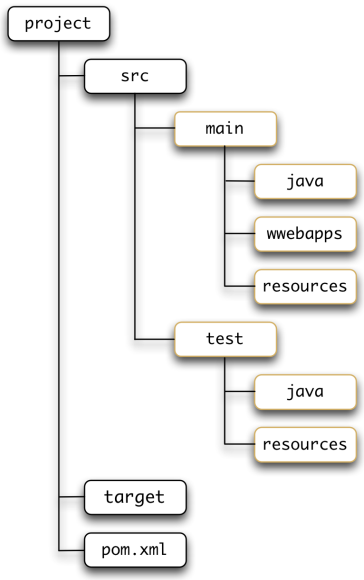
- La descarga de las librerías (ficheros JAR) externas de las que depende un proyecto
- La construcción, prueba y despliegue del proyecto desarrollado, produciendo el fichero JAR o WAR final a partir de su código fuente y del fichero POM de descripción del proyecto

Maven se origina de hecho en la comunidad *open source*, en concreto en la Apache Software Foundation en la que se desarrolló para poder gestionar y minimizar la complejidad de la construcción del proyecto Jakarta Turbine en 2002. El diseñador principal de Maven fue Jason van Zyl, ahora en la empresa Sonatype. En 2003 el proyecto fue aceptado como proyecto de nivel principal de Apache. En octubre de 2005 se lanzó Maven 2. Desde entonces ha sido adoptado como la herramienta de desarrollo de software de muchas empresas y se ha integrado con muchos otros proyectos y entornos. Maven 3.0 se lanzó en octubre de 2010, siendo la mayoría de sus comandos compatibles con Maven 2.

Maven es una herramienta de línea de comando, similar a las herramientas habituales en Java como `javac`, `jar` o a proyectos como Ant. Aunque es posible utilizar Maven en IDEs como Eclipse o Glassfish, es muy útil conocer la utilización de Maven en línea de comandos porque es la base de cualquier adaptación gráfica.

Una de las características principales de Maven es su enfoque declarativo, frente al enfoque orientado a tareas de herramientas tradicionales como Make o Ant. En Maven, el proceso de compilación de un proyecto se basa en una descripción de su estructura y de su contenido. Maven mantiene el concepto de modelo de un proyecto y obliga a definir un identificador único para cada proyecto que desarrollemos, así como declarar sus características (URL, versión, librerías que usa, tipo y nombre del artefacto generado, etc.). Todas estas características deben estar especificadas en el fichero POM (*Project Object Model*, fichero `pom.xml` en el directorio raíz del proyecto).

Maven impone una estructura de directorios en la que guardar los distintos elementos de un programa Java. En el caso de una aplicación web:



Instalación de Maven

Maven ya viene preinstalado en la máquina virtual del experto. La instalación en Linux es muy sencilla.

En primer lugar debemos descargar la última versión de la página web oficial[32] y descomprimirla en algún directorio del sistema. En el caso de la MV, lo hemos instalado en `/usr/local/maven`.

Maven es una aplicación Java, y utiliza la variable `JAVA_HOME` para encontrar la ruta del JDK. También es necesario añadir el directorio `bin` de Maven al `PATH` del sistema. Se pueden definir en el fichero de configuración `.profile` de un usuario. En nuestro caso hemos modificado el único usuario de la MV `expertojava`. La variabel de entorno `M2_HOME` es utilizada por IntelliJ para localizar la ubicación de Maven. El código que hemos añadido ha sido este:

Fichero .profile:

```
## Java
export JAVA_HOME=/usr/local/java
PATH=$JAVA_HOME/bin:$PATH

## Maven
export M2_HOME=/usr/local/maven
PATH=$PATH:/usr/local/maven/bin
```

Dependencias de librerías en proyectos Java

Una característica del desarrollo de proyectos Java es la gran cantidad de librerías (ficheros JAR) necesarios para compilar y ejecutar un proyecto. Todas las librerías que se importan deben estar físicamente tanto en la máquina en la que se compila el proyecto como en la que posteriormente se ejecuta.

[32] http://maven.apache.org/download.html

El proceso de mantener estas dependencias es tedioso y muy propenso a errores. Hay que obtener las librerías, cuidar que sean las versiones correctas, obtener las librerías de las que éstas dependen a su vez y distribuirlas todas ellas en todos los ordenadores de los desarrolladores y en los servidores en los que el proyecto se va a desplegar.

Por ejemplo, si nuestro proyecto necesita una implementación de JPA, como Hibernate, es necesario bajarse todos los JAR de Hibernate, junto con los JAR de los que depende, una lista de más de 15 ficheros. Es complicado hacerlo a mano y distribuir los ficheros en todos los ordenadores en los que el proyecto debe compilarse y ejecutarse. Para que Maven automatice el proceso sólo es necesario declarar la dependencia con este JAR en el fichero POM con las siguientes líneas:

```
...
   <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-entitymanager</artifactId>
      <version>3.5.6-Final</version>
   </dependency>
...
```

Maven se encarga de descargar todas las bibliotecas necesarias para un proyecto cuando ejecutamos el comando `mvn install`. Las guarda en el denominado *repositorio local*, el directorio oculto `.m2` en el directorio raíz del usuario, y después copia referencias a ellas en el proyecto.

El proceso de build de un proyecto

Los que hemos programado en C recordamos los ficheros `Makefile` en los que se especificaban las dependencias entre los distintos elementos de un proyecto y la secuencia de compilación necesaria para generar una librería o un ejecutable. En Java, el desarrollo de aplicaciones medianamente complejas es más complicado que en C. Estamos obligados a gestionar un gran número de recursos: código fuente, ficheros de configuración, librerías externas, librerías desarrolladas en la empresa, etc. Para gestionar este desarrollo es necesario algo de más nivel que las herramientas que proporciona Java (`javac`, `jar`, `rmic`, `java`, etc.)

¿En qué consiste el proceso de compilación y empaquetado en Java?. Básicamente en construir lo que Maven llama un *artefacto* (terminología de Maven que significa *fichero*) a partir de un proyecto Java definido con una estructura propia de Maven (apartado siguiente). Los posibles artefactos en los que podemos empaquetar un programa Java son:

Fichero JAR
: librería de clases o aplicación standalone. Contiene clases Java compiladas (.class) organizadas en paquetes, ficheros de recursos y (opcionalmente) otros ficheros JAR con bibliotecas usadas por las clases. En las aplicaciones enterprise, los EJB también se empaquetan en ficheros JAR que se despliegan en servidores de aplicaciones.

Fichero WAR
: aplicación web lista para desplegarse en un servidor web. Contiene un conjunto de clases Java, bibliotecas, ficheros de configuración y ficheros de distintos formatos que maneja el servidor web (HTML, JPG, etc.)

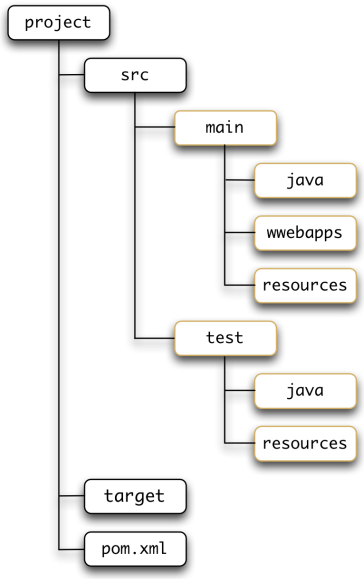
Fichero EAR
: aplicación enterprise que se despliega en un servidor de aplicaciones. Contiene bibliotecas, componentes EJB y distintas aplicaciones web (ficheros WAR).

Además, el ciclo de desarrollo de un proyecto es más complejo que esta construcción, ya que es necesario realizar un conjunto de tareas adicionales como gestionar las dependencias con librerías externas, integrar el código en repositorios de control de versiones (CVS, subversion o Git), lanzar tests o desplegar la aplicación en algún servidor de aplicaciones.

Podría pensarse que los entornos de desarrollo (Eclipse, Netbeans o IntelliJ) pueden dar una buena solución a la complejidad del proceso de construcción, pero no es así. Son imprescindibles para el desarrollo, pero no ayudan demasiado en la construcción del proyecto. La configuración de las dependencias se realiza mediante asistentes gráficos que no generan ficheros de texto comprensibles que podamos utilizar para comunicarnos con otros compañeros o equipos de desarrolladores y que pueden dar lugar a errores. El hecho de que sean entornos gráficos hacen complicado también usarlos en procesos de automatización y de integración continua.

Estructura de un proyecto Maven

La estructura de directorios de una aplicación web Maven es la que hemos visto anteriormente. Volvemos a mostrar aquí la figura:



El nombre del directorio raíz no influye en el proyecto Maven, podemos cambiarlo sin que afecte a ninguno de sus elementos. En ese directorio raíz se definen los siguientes directorios:

- `src`: código fuente del proyecto, tanto clases principales como clases de prueba. Dentro se define un directorio `main` y otro `test`, en donde van el código fuente de la aplicación y su código de prueba. Dentro de ambos se define un directorio `java` con los paquetes de código fuente de la aplicación, un directorio `webapps` con los ficheros HTML, JSP y de configuración de la aplicación web y un directorio `resources` en el que se dejan ficheros de configuración. Ambos directorios se añaden al *classpath*.
- `target`: clases compiladas y artefactos generados a partir del código fuente y del resto de ficheros del directorio `src`.
- fichero `pom.xml`: fichero con la descripción de los elementos necesarios para todo el ciclo de vida del proyecto: compilación, test, empaquetado, despliegue e instalación en el repositorio de la empresa.

POM: Project Object Model

El elemento más importante de un proyecto Maven, a parte de su estructura, es su fichero POM en el que se define completamente el proyecto. Este fichero define elementos XML preestablecidos que deben ser definidos para el proyecto concreto que estamos desarrollando. Viendo algunos de ellos podemos entender también más características de Maven.

Vamos a utilizar como ejemplo la versión inicial del POM del proyecto web que vamos a construir en un rato. Veamos su fichero `pom.xml`. Al comienzo nos encontramos con la cabecera XML y la definición del proyecto:

pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"  xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://
maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"> (1)
    <modelVersion>4.0.0</modelVersion>

    <groupId>org.expertojava</groupId>
    <artifactId>jbib-rest</artifactId>
    <version>1.0-SNAPSHOT</version>(2)
    <packaging>war</packaging> (3)

    <name>jbib-rest</name> (4)
```

(1) La primera definición `project xmlns` es común para todos los ficheros `pom.xml`. En ella se declara el tipo de esquema XML y la dirección donde se encuentra el fichero de esquema XML. Se utiliza para que los editores de XML puedan validar correctamente el fichero. Esta sintaxis depende de la versión de Maven que se esté utilizando.

(2) Después aparece la identificación del proyecto, en la que hay que definir el grupo que desarrolla el proyecto (*groupId*), el nombre del artefacto que genera el proyecto (*artifactId*), el tipo de empaquetamiento (*packaging*) y su versión (*version*). Estos campos representan las denominadas *coordenadas del proyecto* (hablaremos de ello más adelante). En nuestro caso son `org.expertojava:jbib-rest:war:1.0-SNAPSHOT`.

(3) En el atributo `packaging` debemos definir el tipo de empaquetado del *artefacto* resultante. En nuestro caso, será un fichero WAR que contendrá toda la aplicación web. Este artefacto se generará cuando hagamos un `mvn package`.

(4) Por último, el atributo `name` define el nombre lógico del proyecto.

A continuación se definen algunas propiedades del proyecto, que se utilizarán en los distintos procesos de Maven. En nuestro caso, por ahora, sólo la codificación de caracteres que estamos utilizando en el código fuente de nuestro proyecto:

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

Después vienen la definición de las dependencias del proyecto: librerías de las que dependen el proyecto. En nuestro caso:

- Librerías Java EE: `javax:javaee-web-api:7.0`

- Librarías para logs logs_ : `log4j:log4j:1.2.17` y `commons-logging:commons-logging:1.2`
- JUnit: `junit:junit:4.11`

```
<dependencies>
    <dependency>
        <groupId>javax</groupId>
        <artifactId>javaee-web-api</artifactId>
        <version>7.0</version>
        <scope>provided</scope>
    </dependency>
    <dependency>
        <groupId>log4j</groupId>
        <artifactId>log4j</artifactId>
        <version>1.2.17</version>
    </dependency>
    <dependency>
        <groupId>commons-logging</groupId>
        <artifactId>commons-logging</artifactId>
        <version>1.2</version>
    </dependency>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.11</version>
    </dependency>
</dependencies>
```

Por último, definimos algunas características de los procesos de Maven que construyen el proyecto, definiendo parámetros para los *pluging* de Maven que se encargan de ejecutarlos.

```
<build>
    <finalName>${project.name}</finalName> ❶
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>  ❷
            <artifactId>maven-compiler-plugin</artifactId>
            <version>3.1</version>
            <configuration>
                <source>1.7</source>
                <target>1.7</target>
            </configuration>
        </plugin>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId> ❸
            <artifactId>maven-war-plugin</artifactId>
            <version>2.3</version>
            <configuration>
                <failOnMissingWebXml>false</failOnMissingWebXml>
            </configuration>
        </plugin>
        <plugin>
            <groupId>org.wildfly.plugins</groupId> ❹
            <artifactId>wildfly-maven-plugin</artifactId>
```

```
                <version>1.0.2.Final</version>
                <configuration>
                    <hostname>localhost</hostname>
                    <port>9990</port>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

❶ En `finalName` definimos el nombre del *artefacto* generado cuando hagamos un `mvn package`. En nuestro caso será `jbib-web.war`.
❷ En el plugin `maven-compiler-plugin` declaramos la versión de Java con la que queremos que se compilen las clases. En nuestro caso, la versión 1.7.
❸ El plugin `maven-war-plugin` lo usamos únicamente para declarar que podemos no incluir un fichero `web.xml` en la aplicación.
❹ Por último, el plugin de Maven de Wildfly[33] `wildfly-maven-plugin` permite realizar el despliegue de la aplicación web en el servidor de aplicaciones. El despliegue se realiza con el comando `mvn wildfly:deploy`.

Repositorios Maven

Los proyectos software modernos necesitan un gran número de clases y librerías definidas en otros proyectos. Esos proyectos pueden ser otros desarrollados por nosotros en la empresa o librerías open source bajadas de Internet.

La tarea de mantener las dependencias de un proyecto es complicada, tanto para las dependencias entre nuestros proyectos como las dependencias con otros proyectos open source disponibles en Internet. Por ejemplo, si queremos utilizar un framework como Spring, tendremos que descargarnos no sólo los JAR desarrollados en el proyecto, sino también un buen número de otras librerías open source que usa. Cada librería es un fichero JAR. ¿Qué pasa si alguna de esas librerías ya las estamos usando y las tenemos ya descargadas? O, peor aún, ¿Qué pasa si estamos usando otras versiones de esas librerías en nuestros proyectos? ¿Podremos detectar los posibles conflictos?. Maven se encarga de gestionar estas dependencias directas y las dependencias transitivas mediante los ficheros POM:

- Descarga las librerías necesarias para construir el proyecto y los ficheros POM asociados a esas librerías
- Resuelve dependencias transitivas, librerías que dependen de librerías de las que dependen nuestro proyecto
- Resuelve conflictos entre librerías

Un elemento fundamental para gestionar las dependencias es poder identificar y nombrar un proyecto. En Maven el nombre de un proyecto se define mediante los siguientes elementos (que en Maven se denominan *coordenadas*):

groupId
: El grupo, compañía, equipo, organización, etc. Se utiliza una convención similar a la de los paquetes Java, comenzando por el nombre de dominio invertido de la organización que crea el proyecto. Por ejemplo, los `groupId` de la Apache Software Foundation comienzan con `org.apache`

[33] https://docs.jboss.org/wildfly/plugins/maven/latest/

artifactId
: Identificador único que representa de forma única el proyecto dentro del `groupId`

version
: Número de versión del proyecto, por ejemplo `1.3.5` o `1.3.6-beta-01`. Es posible utilizar la palabra `SNAPSHOT` en el número de versión para indicar que es una versión en desarrollo y que todavía no está lanzada. Se utiliza internamente en los proyectos en desarrollo. La idea es que antes de que terminemos el desarrollo de la versión 1.0 (o cualquier otro número de versión), utilizaremos el nombre `1.0-SNAPSHOT` para indicar que se trata de "1.0 en desarrollo".

packaging
: Tipo de empaquetamiento del proyecto. Por defecto es `jar`. Un tipo `jar` genera una librería JAR, un tipo `war` se refiere a una aplicación web.

En Maven un proyecto genera un artefacto. El artefacto puede ser un fichero JAR, WAR o EAR. El tipo de artefacto viene indicado en el tipo de empaquetamiento del proyecto.

El nombre final del fichero resultante de la construcción del proyecto es por defecto: `<artifactId>-<version>.<packaging>`.

Por ejemplo, Apache ha desarrollado el proyecto `commons-email` que proporciona una serie de utilidades para la gestión de correos electrónicos en Java. Sus coordenadas son:

```
org.apache.commons:commons-email:1.1:jar
```

El artefacto (fichero JAR) generado por el proyecto tiene como nombre `email-1.1.jar`

Cuando ejecutamos Maven por primera vez veremos que descarga un número de ficheros del repositorio remoto de Maven. Estos ficheros corresponden a plugins y librerías que necesita para construir el proyecto con el que estamos trabajando. Maven los descarga de un repositorio global a un repositorio local donde están disponibles para su uso. Sólo es necesario hacer esto la primera vez que se necesita la librería o el plugin. Las siguientes ocasiones ya está disponible en el repositorio local.

La direcciones en las que se encuentran los repositorios son las siguientes:

Repositorio central
: El repositorio central de Maven se encuentra en http://repo1.maven.org/maven2. Se puede acceder a la dirección con un navegador y explorar su estructura.

Repositorio local
: El repositorio local se encuentra en el directorio `${HOME}/.m2/repository`.

La estructura de directorios de los repositorios (tanto el central como el local) está directamente relacionada con las *coordenadas* de los proyectos. Los proyectos tienen la siguiente ruta, relativa a la raíz del repositorio:

```
/<groupId>/<artifactId>/<version>/<artifactId>-<version>.<packaging>
```

Por ejemplo, el artefacto `commons-email-1.1.jar`, con coordenadas `org.apache.commons:commons-email:1.1:jar` está disponible en la ruta:

```
/org/apache/commons/commons-email/1.1/commons-email-1.1.jar]]
```

Dependencias de versiones

El estándar de Maven para los números de versiones es muy importante, porque permite definir reglas para gestionar correctamente las dependencias en caso de conflicto. El número de versión de un proyecto se define por un número principal, un número menor y un número incremental. También es posible definir un calificador, para indicar una versión alfa o beta. Los números se separan por puntos y el calificador por un guión. Por ejemplo, el número `1.3.5-alpha-03` define un número de versión principal 1, la versión menor 3, la versión incremental de 5 y el calificador de `alpha-03`.

Maven compara las versiones de una dependencia utilizando este orden. Por ejemplo, la versión 1.3.4 representa un build más reciente que la 1.0.9. Los clasificadores se comparan utilizando comparación de cadenas. Hay que tener cuidado, porque `alpha10` es anterior a `alpha2`; habría que llamar al segundo `alpha02`.

Maven permite definir rangos de versiones en las dependencias, utilizando los operadores de rango exclusivos `(` , `)` o inclusivos `[` , `]`. Así, por ejemplo, si queremos indicar que nuestro proyecto necesita una versión de JUnit mayor o igual de 3.8, pero menor que 4.0, lo podemos indicar con el siguiente rango:

```
<version>[3.8,4.0)</version>
```

Si una dependencia transitiva necesita la versión 3.8.1, esa es la escoge Maven sin crear ningún conflicto.

Es posible también indicar rangos de mayor que o menor que dejando sin escribir ningún número de versión antes o después de la coma. Por ejemplo, `[4.0,)` representa cualquier número mayor o igual que 4.0, `(,2.0)` representa cualquier versión menor que la `2.0` y `[1.2]` significa sólo la versión `1.2` y ninguna otra.

Cuando dos proyectos necesitan dos versiones distintas de la misma librería, Maven intenta resolver el conflicto, descargándose la que satisface todos los rangos. Si no utilizamos los operadores de rango estamos indicando que preferimos esa versión, pero que podríamos utilizar alguna otra. Por ejemplo, es distinto especificar `3.1` y `[3.1]`. En el primer caso preferimos la versión 3.1, pero si otro proyecto necesitara la `3.2` Maven se descargaría esa. En el segundo caso exigimos que la versión descargada sea la `3.1`. Si otro proyecto especifica otra versión obligatoria, por ejemplo `3.2`, entonces el proyecto no se compilará.

La utilización de la palabra `SNAPSHOT` en una dependencia hace que Maven descargue al repositorio local la última versión disponible del artefacto. Por ejemplo, si declaramos que necesitamos la librería `foo-1.0-SNAPSHOT.jar` cuando construyamos el proyecto Maven intentará buscar en el repositorio remoto la última versión de esta librería, incluso aunque ya exista en el repositorio local. Si encuentra en el repositorio remoto la versión `foo-1.0.-20110506.110000-1.jar` (versión que fue generada el 2011/05/06 a las 11:00:00) la descarga y sustituye la que tiene en el local. De forma inversa, cuando ejecutamos el goal *install* y se despliega el artefacto en el servidor remoto, Maven sustituye el palabra `SNAPSHOT` por la fecha actual.

Gestión de dependencias

Hemos visto que una de las características principales de Maven es la posibilidad de definir las dependencias de un proyecto. En la sección `dependencies` del fichero POM se declaran las librerías necesarias para compilar, testear y ejecutar nuestra aplicación. Maven obtiene

estas dependencias del repositorio central o de algún repositorio local configurado por nuestra empresa y las guarda en el directorio `.$HOME/.m2/repository`. Si utilizamos la misma librería en un varios proyectos, sólo se descargará una vez, lo que nos ahorrará espacio de disco y tiempo. Y lo que es más importante, el proyecto será mucho más ligero y portable, porque no llevará incluidas las librerías que necesita para su construcción.

Ya hemos visto en apartados anteriores cómo se declaran las dependencias en el fichero POM. Cada dependencia se define de forma unívoca utilizando sus coordenadas. El mecanismo de declaración de las dependencias es el mismo para las dependencias de librerías externas como para las definidas dentro de la organización.

Para definir una dependencia hay que identificar también el número de versión que se quiere utilizar, utilizando la nomenclatura del apartado anterior. Por ejemplo, la siguiente dependencia especifica una versión 3.0 o posterior de `hibernate`.

```
<dependency>
   <groupId>org.hibernate</groupId>
   <artifactId>hibernate</artifactId>
   <version>[3.0,)</version>
</dependency>
```

Un concepto fundamental en Maven es el de dependencia transitiva. En los repositorios no solo se depositan los artefactos generados por los proyectos, sino también el fichero POM del proyecto. Y en ese fichero se definen las dependencias propias del proyecto. Por ejemplo, junto con el artefacto `hibernate-3.0.jar` se encuentra el fichero POM `hibernate-3.0.pom.xml` en el que se definen sus propias dependencias, librerías necesarias para Hibernate-3.0. Estas librerías son dependencias transitivas de nuestro proyecto. Si nuestro proyecto necesita Hibernate, e Hibernate necesita estas otra librería B, nuestro proyecto también necesita (de forma transitiva) la librería B. A su vez esa librería B tendrá también otras dependencias, y así sucesivamente.

Maven se encarga de resolver todas las dependencias transitivas y de descargar al respositorio local todos los artefactos necesarios para que nuestro proyecto se construya correctamente.

Otro elemento importante es el ámbito (*scope*) en el que se define la dependencia. El ámbito por defecto es *compile* y define librerías necesarias para la compilación del proyecto. También es posible especificar otros ámbitos. Por ejemplo *test*, indicando que la librería es necesaria para realizar pruebas del proyecto:

```
<dependency>
   <groupId>junit</groupId>
   <artifactId>junit</artifactId>
   <version>4.8.1</version>
   <type>jar</type>
   <scope>test</scope>
</dependency>
```

Otros ámbitos posibles son *provided* y *runtime*. Una dependencia se define *provided* cuando es necesaria para compilar la aplicación, pero que no se incluirá en el WAR y no será desplegada. Por ejemplo las APIs de servlets:

```
<dependency>
```

```
    <groupId>javax.servlet</groupId>
    <artifactId>servlet-api</artifactId>
    <version>2.4</version>
    <scope>provided</scope>
</dependency>
```

Las dependencias *runtime* son dependencias que no se necesitan para la compilación, sólo para la ejecución. Por ejemplo los drivers de JDBC para conectarse a la base de datos:

```
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>3.1.13</version>
    <scope>runtime</scope>
</dependency>
```

Una herramienta muy útil es el informe de dependencia. Este informe se genera cuando se ejecuta el objetivo `site`. Maven construye un sitio web con información sobre el proyecto y coloca el informe en el fichero `target/dependencies.html`:

```
$ mvn site
```

El informe muestra una lista de dependencias directas y transitivas y su ámbito.

El ciclo de vida de Maven

El concepto de ciclo de vida es central para Maven. El ciclo de vida de un proyecto Maven es una secuencia de fases que hay que seguir de forma ordenada para construir el artefacto final.

Las fases principales del ciclo de vida por defecto son:

validate
: valida que el proyecto es correcto y que está disponible toda la información necesaria

process-resources
: procesar el código fuente, por ejemplo para filtrar algunos valores

compile
: compila el código fuente del proyecto

test
: lanza los tests del código fuente compilado del proyecto utilizando el framework de testing disponible. Estos tests no deben necesitar que el proyecto haya sido empaquetado o desplegado

package
: empaqueta el código compilado del proyecto en un formato distribuible, como un JAR

integration-test
: procesa y despliega el paquete en un entorno en donde se pueden realizar tests de integración

verify
: lanza pruebas que verifican que el paquete es válido y satisface ciertos criterios de calidad
install: instala el paquete en el repositorio local, para poder ser usado como librería en otros proyectos locales

deploy
: realizado en un entorno de integración o de lanzamiento, copia el paquete final en el repositorio remoto para ser compartido con otros desarrolladores y otros proyectos.

Todas estas fases se lanzan especificándolas como parámetro en el comando `mvn`. El comando `mvn` hay que ejecutarlo estando en el directorio del proyecto. Si ejecutamos una fase, Maven se asegura que el proyecto pasa por todas las fases anteriores. Por ejemplo:

```
$ mvn install
```

Esta llamada realiza la compilación, los tests, el empaquetado los tests de integración y la instalación del paquete resultante en el repositorio local de Maven.

Para un listado completo de todas las opciones de un comando `mvn` se puede consultar la página de Apache Maven Introduction to the Build Lifecycle[34]

Ejecución de tests

Los tests de unidad son una parte importante de cualquier metodología moderna de desarrollo, y juegan un papel fundamental en el ciclo de vida de desarrollo de Maven. Por defecto, Maven obliga a pasar los tests antes de empaquetar el proyecto. Maven permite utilizar los frameworks de prueba JUnit y TestNG. Las clases de prueba deben colocarse en el directorio `src/test`.

Para ejecutar los tests se lanza el comando `mvn test`:

```
$ mvn test
[INFO] Scanning for projects...
...
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running org.expertojava.jbibrest.modelo.UsuarioTest
Tests run: 5, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.082 sec
Running org.expertojava.jbibrest.modelo.OperacionTest
Tests run: 5, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.141 sec
Tests run: 8, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.018 sec
Running org.expertojava.jbibrest.modelo.AvisoTest
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.012 sec

Results :

Tests run: 19, Failures: 0, Errors: 0, Skipped: 0
```

Maven compilará los tests si es necesario. Por defecto, los tests deben colocarse en el directorio `src/test` siguiendo una estructura idéntica a la estructura de clases del proyecto. Maven ejecutará todas las clases que comiencen o terminen con `Test` o que terminen con `TestCase`.

Los resultados detallados de los tests se producen en texto y en XML y se dejan en el directorio `target/surefire-reports`. Es posible también generar los resultados en HTML utilizando el comando:

[34] http://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html

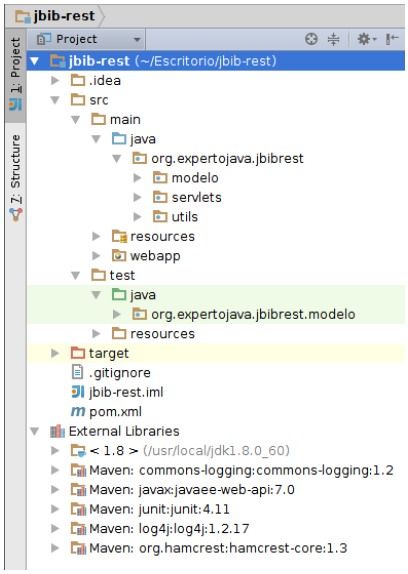
```
$ mvn surefire-report:report
```

El informe HTML se generará en el fichero `target/site/surefire-report.html`.

Uso de Maven en IntelliJ

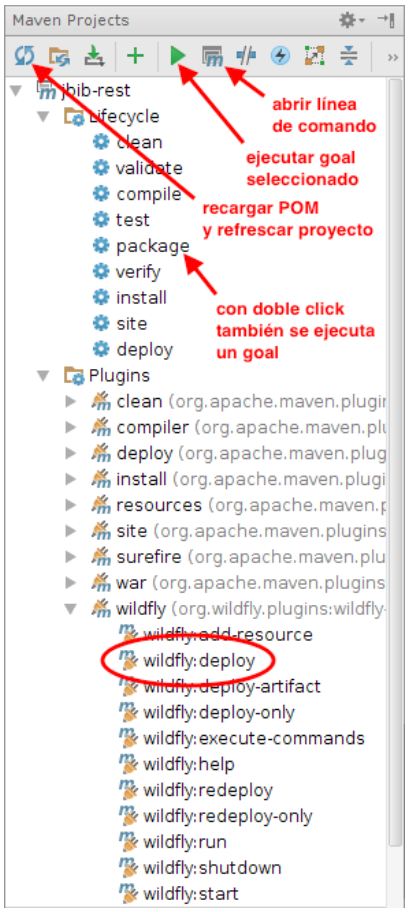
IntelliJ reconoce los proyectos Maven a través del POM. Para importar un proyecto Maven en el IDE debes pulsar en el POM del proyecto. IntelliJ analiza el POM, descarga todas las librerías necesarias, identifica los directorios de la aplicación y los configura como directorios de fuentes, de tests, etc.

En la siguiente imagen vemos la estructura de directorios y las librerías de una versión inicial del proyecto web cargado. Podemos ver este panel en la parte superior izquierda de IntelliJ.



- Bajo el directorio raíz vemos el directorio `src` con los subdirectorios `main` y `test`. También los ficheros `pom.xml` con el POM de Maven y el fichero `.gitignore` con los patrones a ignorar en el control de versiones. Y también se encuentra el directorio `.idea` y el fichero `jbib-rest.iml` propios del IDE.
- Debajo vemos las librerías declaradas en el POM y descargadas por Maven. IntelliJ las reconoce como librerías del proyecto y las incluirá en el paquete WAR cuando se realice el despliegue y ejecución del proyecto.

En la parte superior derecha de IntelliJ podemos ver el panel de Maven. Desde este panel podemos interactuar con el comando Maven.

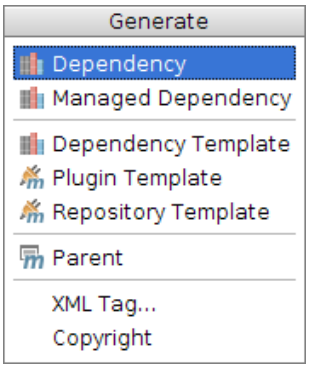


- Para ejecutar un *goal*, por ejemplo `package`, debemos seleccionar el objetivo y pulsar en el pequeño botón *play*, o hacer un doble click sobre el objetivo. Veremos que se abre en la parte inferior de IntelliJ un panel en el que se muestra la ejecución del comando Maven. El resultado es el mismo que si abrimos un terminal, nos movemos a la raíz del proyecto (que contiene el fichero POM) y ejecutamos desde línea de comando:

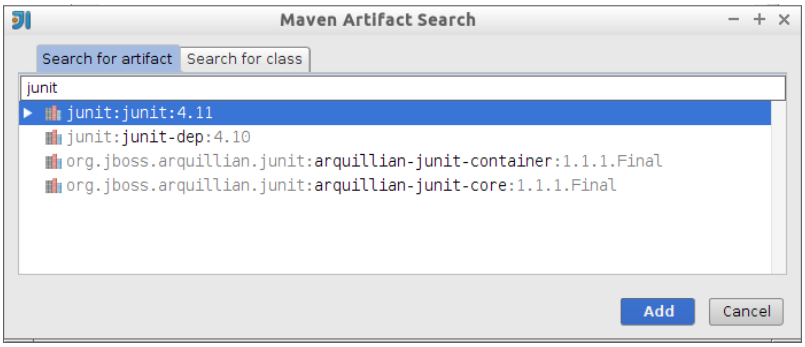
```
$ mvn test
```

- El plugin `org.wildfly.plugins` proporciona *goals* adicionales relacionados con el despliegue del WAR en el servidor de aplicaciones. Los veremos más adelante.
- El icono *Maven* sirve para abrir una ventana de diálogo en la que podemos lanzar el comando Maven de forma textual, escribiendo los parámetros adicionales que necesitemos
- El botón de refresco sirve para recargar el POM en IntelliJ y actualizar la estructura del proyecto y sus dependencias si hemos realizado algún cambio en el fichero.

Otra característica interesante de IntelliJ es que permite añadir dependencias en el POM de forma interactiva. Desde el fichero `pom.xml` pulsamos el botón derecho y seleccionamos *Generate…* (o pulsamos `Alt+Insertar`) y aparecerá el siguiente menú que permite modificar elementos del POM de forma interactiva:



Por ejemplo, seleccionando *Dependency* aparece una herramienta de búsqueda de librerías que permite filtrar por nombre y explorar las distintas versiones de las librerías disponibles en el repositorio central de Maven:



Maven con Git

Para crear un repositorio Git con un proyecto Maven, hay que hacer lo habitual: inicializar Git en la raíz del proyecto y añadir todos sus ficheros. Hay que tener cuidado de añadir en el repositorio sólo los ficheros fuente. Todos los ficheros de artefacto que crea Maven a partir de los ficheros fuente originales deben ser ignorados. Para ello basta con ignorar el directorio `target`.

El siguiente fichero `.gitignore` contiene las reglas que determinan los ficheros ignorados en un proyecto Maven en el que trabajamos con IntelliJ. Es recomendable ignorar también el fichero `.idea/workspace.xml` que contiene el estado de trabajo del IDE (pestañas abiertas, dimensiones de la ventana actual, etc.).

Fichero `.gitignore`

```
# ignore Maven generated target folders
target

# ignore IDEA files
.idea/workspace.xml
```

Cómo crear un proyecto Maven mínimo

Para empezar a desarrollar un proyecto con Maven necesitamos comenzar con un proyecto Maven mínimo, en el que ir añadiendo todo el código. Hemos visto que un proyecto Maven consiste en:

- un directorio de proyecto en el que se incluye un fichero POM con la descripción de las dependencias y otras características del proyecto
- la estructura de directorios propia de los proyectos Maven vista anteriormente

El directorio contenedor del fichero POM y de la estructura de directorios Maven puede tener cualquier nombre. El nombre del proyecto Maven sólo se define en el fichero POM. Esta característica simplifica el manejo de los proyectos Maven, haciéndolos independiente del nombre del directorio donde están contenidos.

Hay varias formas posibles de crear un proyecto Maven. A lo largo del curso utilizaremos cualquiera de estas opciones indistintamente.

Creación manual de directorios y POM

Una forma básica de crear esta estructura mínima es crear la estructura de directorios manualmente y un fichero POM mínimo en el que se defina el nombre del proyecto. Pero esta es una forma bastante tediosa, sobre todo por el trabajo de crear a mano la estructura de directorios.

Utilizando un arquetipo

Para crear el proyecto de forma más cómoda podemos hacerlo a partir de un *arquetipo Maven* (desde línea de comando con el comando `mvn archetype:generate`, o desde IntelliJ) o a partir de una plantilla inicial que ya tengamos creada. En el primer caso como parte del proceso de creación se pide el nombre del proyecto que estamos creando, para añadirla al fichero POM. En el segundo caso debemos de cambiar este nombre a mano en el fichero POM.

Para crear un proyecto Maven mínimo desde un arquetipo podemos usar el arquetipo `org.codehaus.mojo.archetypes:webapp-javaee7:1.1` o también el arquetipo de Adam Bien `com.airhacks:javaee7-essentials-archetype:1.3` (enlace GitHub[35]).

Desde una plantilla inicial en Bitbucket

También puedes tener una plantilla básica inicial en un repositorio Git como Bitbucket. En nuestro caso puedes usar la plantilla `ejemplo-webapp` que hemos dejado en Bitbucket[36]. Debes descargarla y cambiar el nombre del proyecto en el POM.

2.2. Paso a paso: creación del proyecto Git

Por hacer sencillo el proceso de creación vamos a hacer un fork del proyecto inicial que hemos dejado en Bitbucket en el que ya hemos creado la estructura de directorios y el fichero POM necesarios, junto con el fichero `.gitignore` necesario para Git.

1. Haz un fork en tu cuenta de Bitbucket del proyecto `java_ua/jbib-rest`
2. Descarga a tu ordenador este proyecto recién copiado. Puedes usar un `git clone` desde el terminal o la opción de IntelliJ *Check out from Version Control* > *Git*. Colócalo, por ejemplo, en el escritorio.

[35] https://github.com/AdamBien/javaee7-essentials-archetype
[36] https://bitbucket.org/java_ua/ejemplo-webapp/

2.3. Paso a paso: despliegue con Maven

El proyecto de aplicación web que vamos a desarrollar a lo largo del curso se compone de dos partes principales:

- Una aplicación Java que se despliega en un servidor Java EE (WildFly) e implementa un API REST que proporciona la lógica de negocio
- Una aplicación JavaScript que proporciona la interfaz de usuario que se ejecuta en el navegador

Vamos a comenzar a construir la aplicación Java, un artefacto WAR que contendrá distintos paquetes que iremos desarrollando a lo largo de estas sesiones.

Comenzaremos con una aplicación muy básica, similar a la que has desarrollado en la asignatura de *Componentes Web*, que contiene algunos elementos iniciales básicos:

- Fichero JSP con un formulario que envía una petición a un servlet
- Servlet que procesa los parámetros de la petición y gestiona algún error e invoca a una clase del modelo
- Clase del modelo que implementa una sencilla funcionalidad y realiza una mínima gestión de errores
- Logging
- Tests

Todo esto en un proyecto Maven con un POM que permite generar el WAR y desplegarlo en el servidor WildFly.

La mayor parte de esta aplicación básica ya está preparada en el proyecto que te has descargado.

1. Vamos a comenzar probando que el proyecto compila correctamente y que los tests pasan. Abre un terminal, vete al directorio de proyeto y lanza el comando `mvn test`:

```
$ cd jbib-web
$ mvn test
...
-------------------------------------------------------
 T E S T S
 -------------------------------------------------------
 Running org.expertojava.jbibrest.modelo.NombreTest
 06/11/2015 11:31:22 - DEBUG - Test
 getNombreDeberiaDevolverSraCuandoHombre
 06/11/2015 11:31:22 - DEBUG - Creada instancia de Nombre
 06/11/2015 11:31:22 - DEBUG - Test
 getNombreDeberiaDevolverSrCuandoHombre
 06/11/2015 11:31:22 - DEBUG - Creada instancia de Nombre
 Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.255
 sec

Results :

Tests run: 2, Failures: 0, Errors: 0, Skipped: 0
```

```
[INFO]
 ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO]
 ------------------------------------------------------------------------
[INFO] Total time: 3.162 s
[INFO] Finished at: 2015-11-06T11:31:22+01:00
[INFO] Final Memory: 14M/127M
[INFO]
 ------------------------------------------------------------------------
```

2. Vamos ahora a desplegar la aplicación web desde línea de comando. Abre otro terminal y lanza WildFly con el comando:

```
$ standalone.sh
```

Cuando haya arrancado WildFly, en el otro terminal ejecuta el *goal* `wildfly:deploy`. El proyecto se deberá desplegar correctamente en el servidor de aplicaciones:

```
$ cd jbibrest-expertojava
$ mvn wildfly:deploy

...
[INFO]
[INFO] --- maven-war-plugin:2.3:war (default-war) @ jbib-rest ---
[INFO] Packaging webapp
[INFO] Assembling webapp [jbib-rest] in [/home/expertojava/Escritorio/
jbib-rest0/target/jbib-rest]
[INFO] Processing war project
[INFO] Copying webapp resources [/home/expertojava/Escritorio/jbib-
rest0/src/main/webapp]
[INFO] Webapp assembled in [122 msecs]
[INFO] Building war: /home/expertojava/Escritorio/jbib-rest0/target/
jbib-rest.war
[INFO]
[INFO] <<< wildfly-maven-plugin:1.0.2.Final:deploy (default-cli) <
 package @ jbib-rest <<<
[INFO]
[INFO] --- wildfly-maven-plugin:1.0.2.Final:deploy (default-cli) @
 jbib-rest ---
nov 06, 2015 11:44:55 AM org.xnio.Xnio <clinit>
INFO: XNIO version 3.2.2.Final
nov 06, 2015 11:44:55 AM org.xnio.nio.NioXnio <clinit>
INFO: XNIO NIO Implementation Version 3.2.2.Final
nov 06, 2015 11:44:55 AM org.jboss.remoting3.EndpointImpl <clinit>
INFO: JBoss Remoting version 4.0.3.Final
[INFO]
 ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO]
 ------------------------------------------------------------------------
[INFO] Total time: 9.782 s
[INFO] Finished at: 2015-11-06T11:44:59+01:00
[INFO] Final Memory: 18M/178M
```

```
[INFO]
 ------------------------------------------------------------------------
```

3. Probamos ahora la aplicación desplegada. En un navegador accede a la URL *http://localhost:8080/jbib-rest/* en la que estará respondiendo la aplicación. Aparecerá un sencillo formulario con el que puedes comprobar que la aplicación está funcionando. Rellénalo y pulsa *Enviar*. Verás la petición en la URL y la página resultante con un saludo.
4. Prueba a editar la URL de la petición a mano, cambiando algunos parámetros:

```
http://localhost:8080/jbib-rest/holamundo?
nombre=Leia&edad=22&genero=mujer
```

5. Prueba a introducir parámetros erróneos como un género que no existe o una edad negativa. En algunos casos el servlet detectará el error y devolverá un error 400 (BAD REQUEST) y en otros el error se detectará en la clase Java que implementa la lógica de negocio y se generará una excepción en tiempo de ejecución y un mensaje en el log. El log está configurado para aparecer en la consola.
6. En IntelliJ repasa el código de la aplicación:
   - El fichero `index.jsp` en `src/main/webapp`
   - El servlet `org.expertojava.jbibrest.HolaMundo.java` en `src/main/java/`
   - La clase de modelo `org.expertojava.jbibrest.modelo.Nombre` también en el directorio de fuentes

Piensa las siguientes preguntas sobre la aplicación: ¿Quién construye la cadena de saludo? ¿Qué parámetros hay que pasar? ¿Qué prerequisitos deben cumplir esos parámetros? ¿Desde dónde se invoca a esa construcción? ¿Quién, qué códigos de error HTTP se devuelven y en qué casos?

7. Los tests merecen mención especial. Son muy sencillos, se encuentran en el fichero `org.expertojava.jbib.modelo.NombreTest.java` en el directorio `src/test/java`. Comprueban el método `getNombre()` del modelo.

```
package org.expertojava.jbibrest.modelo;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.junit.Test;

import static org.junit.Assert.*;

public class NombreTest {

    private static Log logger = LogFactory.getLog(NombreTest.class);

    @Test
    public void getNombreDeberiaDevolverSrCuandoHombre() throws
 Exception {
        logger.debug("Test getNombreDeberiaDevolverSrCuandoHombre");
        Nombre nombre = new Nombre("Jack", 40, Genero.hombre);
```

```
        String nombreStr = nombre.getNombre();
        assertTrue(nombreStr.equals("Sr. Jack (40 años)"));
    }

    @Test
    public void getNombreDeberiaDevolverSraCuandoMujer() throws
 Exception {
        logger.debug("Test getNombreDeberiaDevolverSraCuandoHombre");
        Nombre nombre = new Nombre("Kate", 30, Genero.mujer);
        String nombreStr = nombre.getNombre();
        assertTrue(nombreStr.equals("Sra. Kate (30 años)"));
    }
}
```

8. Elimina la aplicación WAR desplegada haciendo:

```
$ mvn wildfly:undeploy
```

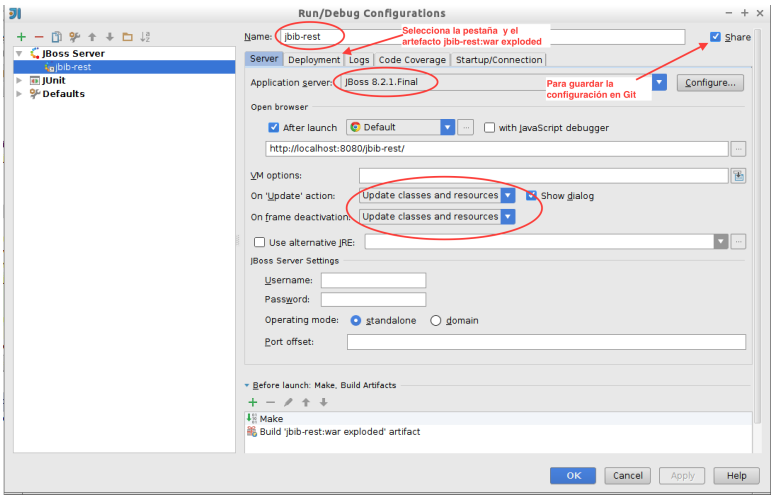
Para comprobar si el WAR está desplegado puedes conectarte a la URL: *http://localhost:9990/*, entrar en la consola de administración del servidor y seleccionar *Runtime* > *Manage Deployments*. Desde esa pantalla puedes gestionar las aplicaciones desplegadas.

1. Termina deteniendo el servidor WildFly haciendo Ctrl+c en el terminal

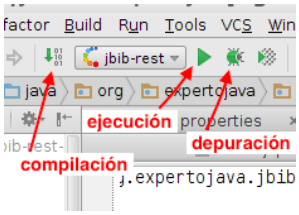
2.4. Paso a paso: despliegue con IntelliJ

Veamos ahora cómo hacer el despliegue de la aplicación web usando IntelliJ.

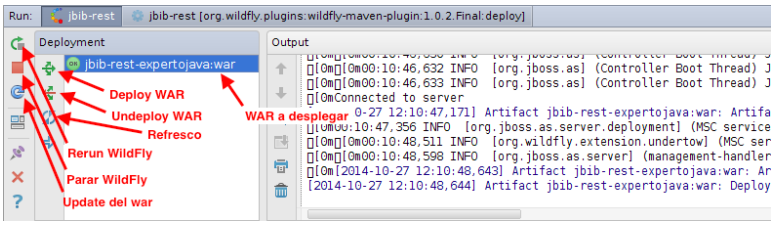
1. Empezamos abriendo el proyecto IntelliJ `jbib-rest`. En el repositorio Git se ha guardado también el fichero `.idea` que hay en la raíz del directorio, por lo que IntelliJ reconoce el proyecto y sus características. En el caso en que no estuviera este fichero habría que importarlo como un proyecto Maven, haciendo un doble click en el fichero POM.
2. Creamos una configuración de ejecución del tipo *JBoss Server* > *Local*. Le ponemos como nombre `jbib-rest`, seleccionamos *JBoss 8.1.0.Final* como servidor de aplicaciones. Pulsamos en la pestaña de *Deployment* y en el símbolo + seleccionamos *Artifact…* > *jbib-rest:war exploded*. Seleccionamos también *Update Classes and Resources* para permitir desplegar *en caliente* las clases compiladas en las que hemos realizado cambios y marcamos la casilla *Share* para que se guarde en el repositorio Git la configuración de ejecución:



3. Ahora pulsamos el botón de ejecución junto al desplegable con el nombre de la configuración de ejecución. Si queremos activar los cambios en caliente debemos pulsar el botón de depuración.



Veremos cómo se abre en la parte inferior el panel *Run*. Está dividido en dos zonas. En la zona derecha se muestra la consola del servidor de aplicaciones seleccionado en la configuración de ejecución (WildFly). El servidor se pone en marcha y vemos los mensajes que va escribiendo. En la parte izquierda está el panel de herramientas relacionadas con el despliegue de aplicación.



En la configuración de ejecución hemos seleccionado el artefacto `jbib-rest:war exploded`. IntelliJ compila las clases, las deja en el directorio `target/classes`, las copia en un directorio de despliegue de WildFly y abre una navegador en la página definida en la configuración de ejecución. Si todo ha funcionado bien, veremos la página con el formulario. Cuando introducimos los datos vemos el saludo.

4. Podríamos también seleccionar el artefacto `jbib-rest:war`. En este caso IntelliJ compilaría las clases y ahora las empaquetaría en el fichero `jbib-rest.war` (en el mismo directorio `target`).
5. Desde IntelliJ también podemos usar el panel de Maven y desplegar (o seleccionar cualquier otro objetivo) con él el WAR. Por ejemplo, una vez desplegado el WAR podemos pulsar en el objetivo *Plugins* > *wildfly* > *wildfly:undeploy* para eliminar el WAR. Cuando lo hacemos se abre una nueva pestaña en el panel de consola en la que aparece la salida del comando Maven. Podemos intentar cargar la página inicial para comprobar que la

aplicación ya no está disponible. Esto sólo funciona cuando se ha escogido el artefacto `jbib-rest:war`.

Después de modificar el estado de un artefacto (desplegarla o eliminarla) con Maven, su estado no se actualiza correctamente en el panel de ejecución. Debes pulsar el botón de refresco para actualizarlo.

2.5. Caso de estudio y modelo de dominio

Introducción

A partir de un supuesto básico de la gestión de una biblioteca, vamos a crear un caso de estudio completo que evolucionará conforme estudiemos las diferentes tecnologías de la plataforma Java Enterprise.

El objetivo de esta sesión es introducir el caso de estudio que vamos a desarrollar, obtener una visión global del proyecto, fijando los casos de uso y requisitos principales y definiendo el esqueleto inicial del problema.

Historias de usuario

Un instituto de educación secundaria nos ha encargado que desarrollemos una aplicación para la gestión de los préstamos realizados en la biblioteca del centro, lo que implica tanto una gestión de los libros como de los alumnos y profesores que realizan estos prestamos.

Tras una serie de entrevistas y reuniones con diferente personal del centro, hemos decidido hacer un prototipo inicial que servirá para probar el funcionamiento de la parte de la aplicación destinada a los *clientes* de la biblioteca: profesores y alumnos que van a poder solicitar préstamos de libros a través de la aplicación.

En concreto, las características que vamos a implementar en este prototipo serán las siguientes:

- La biblioteca contiene **libros**. El sistema debe guardar toda la información necesaria de cada libro: su título, autor, ISBN, etc. Puede existir más de un **ejemplar** de un mismo libro. Se quiere también guardar la información propia de cada ejemplar: fecha de adquisición, defectos que pueda tener, etc. También contamos con **recomendaciones** que ligan un libro origen con otros recomendados a partir de él (y un comentario por recomendación).
- Los **usuarios** de la biblioteca (profesores y alumnos) utilizarán la aplicación para realizar una serie de acciones sobre estos libros: consultar su disponibilidad, pedirlos prestados, consultar la lista de libros prestados, etc. En concreto:
  - Pedir prestado un ejemplar de un libro (que el personal entregará al usuario cuando éste se pase por el mostrador de la biblioteca).
  - Consultar el **estado de los libros** y sus ejemplares: un ejemplar puede estar prestado o disponible.
  - Consultar los libros que tiene prestados.
  - Consultar los libros de la biblioteca.
- La **fecha de devolución del préstamo** dependerá de si el usuario es alumno o profesor y empezará a contar a partir del momento en que el ejemplar se toma prestado. El número máximo de libros que puede tener en préstamo un usuario dependerá también de si es

profesor o alumno. Cuando un libro se devuelve, se borra el registro del préstamo del usuario y se crea un nuevo registro en el **histórico de préstamos**.

- Cuando el usuario se retrasa en la devolución de un préstamo se le creará una **multa**. Teniendo una multa no podrá pedir prestado ningún otro libro. Cuando termine de devolver todos los libros comenzará a descontar los días de penalización. El número de días de penalización será la suma de los retrasos en las devoluciones de todos los libros prestados. Cuando pasa la fecha de finalización, la multa se elimina del usuario y se crea un nuevo registro en el **histórico de multas**.

Estas funcionalidades las vamos a convertir más adelante en casos de uso y las vamos a implementar a lo largo del curso, conforme vaya avanzando el proyecto de integración.

Requisitos de información (IRQ)

Los requisitos de información resumen la información persistente que nos interesa almacenar relacionada con el sistema.

Respecto a un **usuario**, nos interesa almacenar:

- *Tipo de usuario*: profesor, alumno
- *Login* (obligatorio) y *passwod*
- Nombre y apellidos
- *Correo electrónico*
- Lista de *préstamos* actuales del usuario
- Multa actual (si existe) del usuario
- Datos referentes a su dirección, como son *calle*, *número*, *piso*, *ciudad* y *código postal*
- Si el usuario es alumno, necesitaremos guardar un *telefono de los padres*
- Si el usuario es profesor, necesitaremos el *nombre de su departamento*

Podremos obtener el **estado de un usuario**:

- **Activo**: Estado por defecto, puede tomar libros prestados
- **Moroso**: Tiene libros fuera de plazo por devolver, no puede tomar libros prestados
- **Multado**: Tiene una multa actual abierta, no puede tomar libros prestados

Respecto a un **libro**, nos interesa almacenar:

- *ISBN* (obligatorio)
- *Título y autor*
- *Número de páginas*
- *Número de ejemplares comprados*
- *Número de ejemplares disponibles*: cambiará conforme se presten y devuelvan ejemplares
- *URI de la portada*: dirección web de la imagen de donde se puede cargar la portada

Queremos que la aplicación realice también recomendaciones de libros, de forma que para un libro a prestar se muestre una lista de libros relacionados. Para ello tendremos **recomendaciones** que relacionarán un libro origen con un libro recomendado y un comentario.

Respecto a un **ejemplar**, almacenaremos:

- *Número de identificación del ejemplar* (obligatorio, código definido por el personal)
- *Identificación del libro* al que pertenece el ejemplar
- *Fecha de adquisición*
- *Observaciones*: texto sobre el estado del ejemplar

Un **préstamo** representa un ejemplar en posesión de un usuario (es un *préstamo activo*). En cada préstamo guardaremos:

_ *Ejemplar* del préstamo _ *Usuario* del préstamo - *Fecha de préstamo - Fecha en la que debería devolverse*

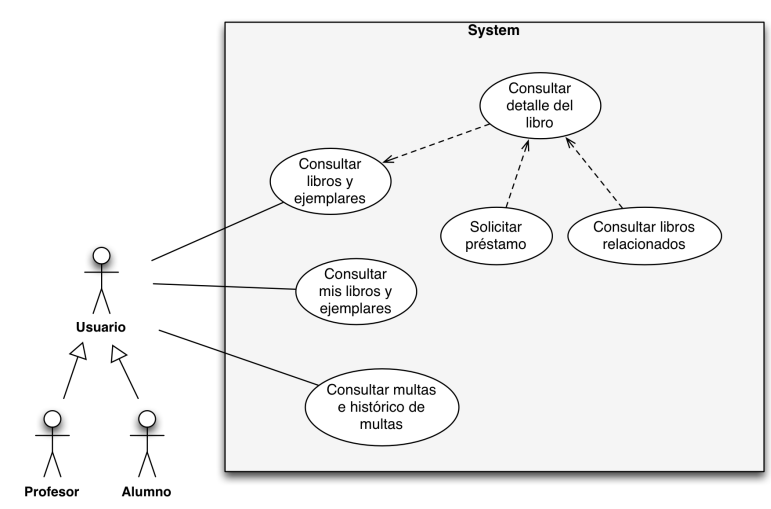
Cuando un usuario se retrase en la devolución de un libro, se le creará una **multa**. De cada multa nos interesa saber:

- *Usuario* que tiene la multa
- *Fecha de inicio*
- *Días acumulados* (para el caso en que se haya abierto una multa y todavía queden ejemplares por devolver)
- *Fecha de finalización*

También queremos guardar los históricos de préstamos y de multas. Los históricos guardarán los identificadores de los ejemplares, usuarios y multas, para poder realizar consultas históricas. El histórico de préstamo guardará también la fecha de devolución del ejemplar.

Casos de uso

Los casos de uso son bastante sencillos. Vamos a centrarnos sólo en la parte del usuario registrado en la biblioteca, dejando para otro momento la parte de la aplicación del bibliotecario en la que se realizaría una gestión (altas, bajas y modificaciones) de los libros, ejemplares y usuarios. El siguiente esquema muestra los casos de uso de un usuario logeado en el sistema:

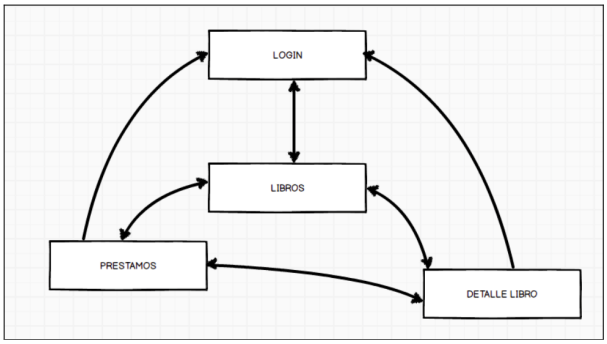


El **usuario** podrá consultar los libros disponibles en la biblioteca y obtener más información sobre aquellos en los que esté interesado, accediendo a la pantalla de detalle del libro. Desde

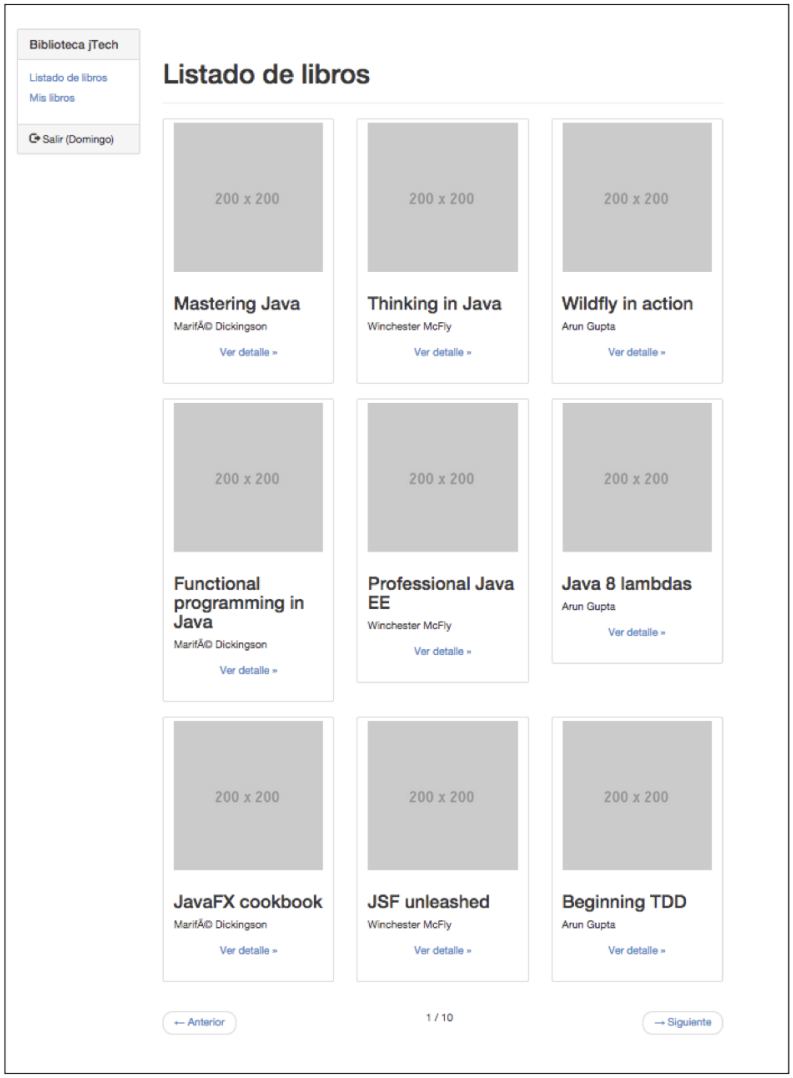
esta pantalla de detalle podrá solicitar el préstamo del libro y ver la información de libros recomendados relacionados con el actual.

Para hacerse una idea mejor del funcionamiento de la aplicación es conveniente hacer diseños iniciales o *mockups* de estas pantallas, junto con un esquema de navegación de las mismas. Los vemos a continuación.

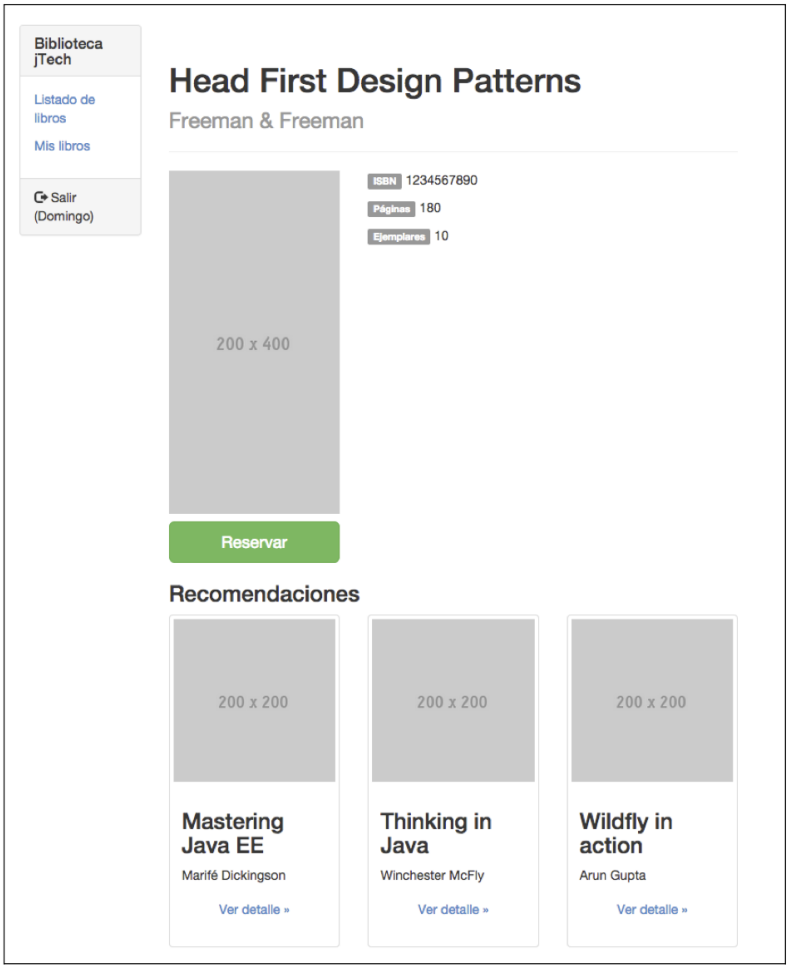
Esquema de navegación:



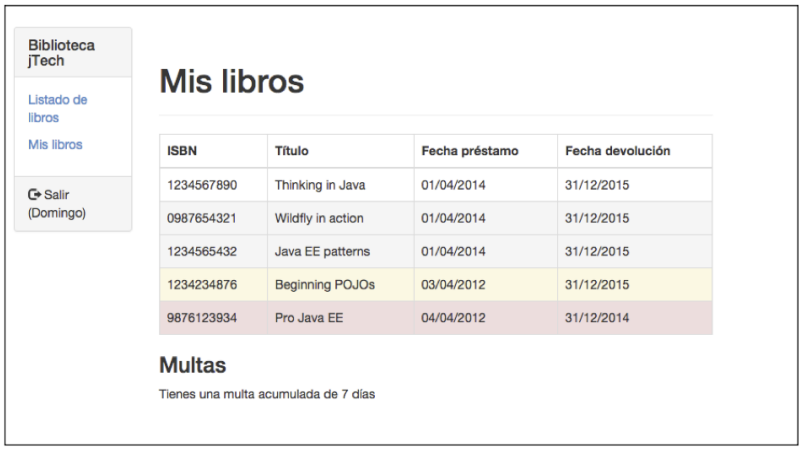
Pantalla con el listado de libros:



Pantalla con el detalle de libro:



Pantalla con los libros que tiene prestados un usuario:



Estos son bocetos iniciales de pantallas. A partir de ellas, del modelo de datos y de las reglas de negocio que veremos más adelante, diseñaremos el API REST con las funcionalidades que ofrecerá nuestro servicio.

Requisitos de restricción (CRQ)

Podemos resumir en la siguiente tabla las restricciones a aplicar a los casos de uso anteriores:

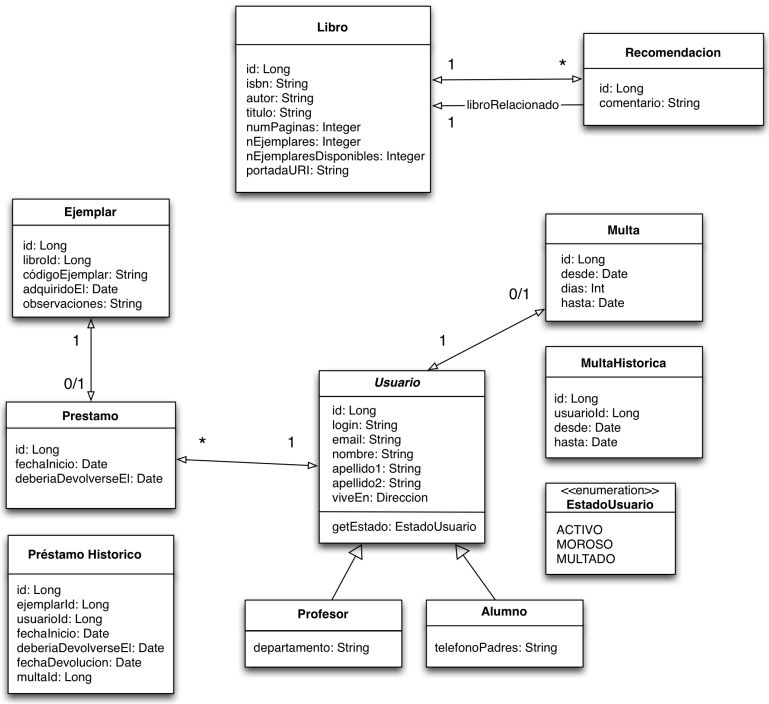
Tipo de usuario	Número máximo de préstamos	Días de préstamo
Alumno	5	7
Profesor	8	30

El máximo de libros prestados de un alumno es 6. Los libros prestados los tiene que devolver antes de 7 días.

En el momento que un usuario tenga una demora en la devolución de un préstamo, se considerará al usuario moroso y se le impondrá una penalización del doble de días de desfase durante los cuales no podrá ni reservar ni realizar préstamos de libros.

Modelo de clases

A partir de los requisitos y tras unas sesiones de modelado, hemos llegado al siguiente modelo de clases conceptual representado mediante el siguiente diagrama UML:



Utilizaremos un modelo de clases como punto de partida del modelo de datos. En la siguiente sesión construiremos el modelo de datos basándonos en este modelo de clases y utilizando JPA (Java Persistence API). Veremos que este enfoque se denomina ORM (Object Relational Mapping), porque permite definir una relación directa (*mapping*) entre clases Java y tablas de la base de datos. La relación entre clases Java y tablas se define por medio de anotaciones JPA añadidas en el código fuente de las clases.

Vemos que casi todas las clases tienen un atributo `id` de tipo `Long`. Será la clave primaria de la tabla asociada a la clase y será generado automáticamente por la base de datos. Durante un tiempo de la vida del objeto, antes de ser insertado en la base de datos, este identificador va a ser `null`. Tenemos que tener en cuenta esto a la hora de definir correctamente los métodos `equals` y `hashCode`.

¿Por dónde empezamos al hacer el diseño de la aplicación? ¿Por los datos o por las clases? Podemos empezar modelando los datos o las clases, y ambos modelos serán casi semejantes. Normalmente, la elección viene dada por la destreza del analista, si se siente más seguro comenzando por los datos, o con el modelo conceptual de clases. Otra opción es el modelado en paralelo, de modo que al finalizar ambos modelos, podamos compararlos y validar si hemos comprobado todas las restricciones. Daremos más detalles en la siguiente sesión.

Relaciones entre entidades por referencias y por identificador

En la próxima sesión vamos a utilizar el framework JPA para hacer persistentes las entidades. JPA mapea las entidades con tablas de la base de datos y los registros de estas tablas con instancias (objetos) de las clases entidad. Así, por ejemplo, cuando hagamos una consulta a la base de datos usando JPA el framework obtendrá la lista de registros de la base de datos (por ejemplo, libros) y los convertirá automáticamente a instancias de la clase correspiendente (en el ejemplo, la clase `Libro`). Además JPA **traerá a memoria** automáticamente todo el grafo de objetos con los que cada instancia tiene definida una relación (veremos más adelante que esto se puede hacer de forma perezosa o *lazy*).

Para evitar que la gestión en memoria de todo el grafo de objetos penalize demasiado el redimiento de la aplicación se puede utilizar una técnica de partición del grafo de relaciones en las denominadas **agregaciones** (o *aggregates* en ingles). Una agregación es un conjunto de entidades que se van a cargar en memoria, formando un grafo de objetos que la aplicación debe mantener.

La separación de distintas agregaciones permitiría separar la aplicación en distintos microservicios.

2.6. Desarrollo e implementación (parte guiada, 0,5 puntos)

El objetivo de la sesión de hoy es crear todas las clases anteriores (clases de dominio) en el paquete `org.expertojava.jbibrest.modelo` del proyecto inicial que ya te has descargado y has probado que funciona correctamente. Estas clases definirán los tipos de datos básicos que utilizaremos para trabajar con la capa de persistencia y de lógica de negocio de nuestra API REST.

A lo largo de las siguientes sesiones del proyecto construiremos nuevos módulos necesarios para el API REST con los métodos de negocio y la aplicación JavaScript que construye la interfaz de usuario.

Comentamos a continuación los pasos a seguir para desarrollar el esqueleto del dominio del proyecto, que incluye la clase Libro, algunas clases auxiliares, excepciones y pruebas. Al final de esta guía paso a paso tendrás una versión inicial del programa. Deberás entonces terminar de implementar el resto.

Es muy importante en esta parte guiada que no te limites a copiar y pegar el código, sino que reflexiones sobre lo que hace.

Clase abstracta común a todas las entidades `ClaseDominio`

Las clases de dominio representan las entidades que van a hacerse persistentes y con las que van a trabajar las capas de persistencia y de lógica de dominio de la aplicación. En

las siguientes sesiones, cuando veamos JPA, veremos cómo se podrán definir la capa de persistencia de la aplicación directamente a partir de estas clases y cómo se utilizarán para encapsular los datos pasados como parámetros y devueltos por las funciones de la capa de negocio implementadas por componentes EJB.

> Una entidad representa un concepto en el dominio que está definido por su identidad más que por sus atributos. Aunque la identidad de la entidad permanece fija durante toda su vida, sus atributos pueden cambiar. Una entidad debe definir una relación de igualdad basada en su identidad.

Para asegurarnos que todas las entidades de nuestro dominio tienen estas características, definimos una clase abstracta, que será la clase padre de todas las clases de dominio. Definiremos en esa clase el identificador de todas las entidades (de tipo `Long`, por simplificar) y los métodos `equals()` y `hashCode()` basados en comparar este identificador.

El código de `equals` y `hashCode` permite que los identificadores sean `null`. Si dos identificadores son `null`, `equals` devolverá `true`.

org.expertojava.jbibrest.modelo.ClaseDomino

```
package org.expertojava.jbibrest.modelo;

public abstract class ClaseDominio {
    private Long id;

    protected void setId(Long id) { this.id = id; }
    public Long getId() { return id;}

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;

        ClaseDominio that = (ClaseDominio) o;

        return !(id != null ? !id.equals(that.id) : that.id != null);

    }

    @Override
    public int hashCode() {
        return id != null ? id.hashCode() : 0;
    }

}
```

Todas las entidades las vamos a definir dentro del paquete `org.expertojava.jbibrest.modelo`. Cada objeto de dominio se compone de sus atributos, relaciones y de todos los *getter/setter* que encapsulan al objeto.

Clases de domino `Libro` y `Recomendacion`

Vamos a empezar definiendo las clases `Libro` y `Recomendacion`. Empezamos por la clase `Libro`:

org.expertojava.jbibrest.modelo.Libro

```
package org.expertojava.jbibrest.modelo;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import java.util.HashSet;
import java.util.Set;


public class Libro extends ClaseDominio {
    private String isbn;
    private String autor;
    private String titulo;
    private Integer numPaginas;
    private int nEjemplares = 0;
    private int nEjemplaresDisponibles = 0;
    private String portadaURI;
    private Set<Recomendacion> recomendaciones = new
 HashSet<Recomendacion>();

    private static Log logger = LogFactory.getLog(Libro.class);

    public Libro(String isbn) {
        this.isbn = isbn;
        logger.debug("Nueva instancia de Libro: " + isbn);
    }

    public String getIsbn() {
        return isbn;
    }

    public void setIsbn(String isbn) {
        this.isbn = isbn;
    }

    public String getAutor() {
        return autor;
    }

    public void setAutor(String autor) {
        this.autor = autor;
    }

    public String getTitulo() {
        return titulo;
    }

    public void setTitulo(String titulo) {
        this.titulo = titulo;
    }

    public Integer getNumPaginas() {
        return numPaginas;
    }

    public void setNumPaginas(Integer numPaginas) {
```

```
        this.numPaginas = numPaginas;
    }

    public int getnEjemplares() {
        return nEjemplares;
    }

    public void setnEjemplares(int nEjemplares) {
        this.nEjemplares = nEjemplares;
    }

    public int getnEjemplaresDisponibles() {
        return nEjemplaresDisponibles;
    }

    public void setnEjemplaresDisponibles(int nEjemplaresDisponibles) {
        this.nEjemplaresDisponibles = nEjemplaresDisponibles;
    }

    public String getPortadaURI() {
        return portadaURI;
    }

    public void setPortadaURI(String portadaURI) {
        this.portadaURI = portadaURI;
    }


    public Set<Recomendacion> getRecomendaciones() {
        return recomendaciones;
    }

    public void setRecomendaciones(Set<Recomendacion> recomendaciones) {
        this.recomendaciones = recomendaciones;
    }

    @Override
    public String toString() {
        return "Libro{" +
                "id=" + this.getId() +
                ", isbn='" + this.getIsbn() + '\'' +
                ", autor='" + this.getAutor() + '\'' +
                ", titulo='" + this.getTitulo() + '\'' +
                ", numPaginas=" + this.getNumPaginas() +
                ", numEjemplares=" + this.getnEjemplares() +
                ", numEjemplaresDisponibles="
 + this.getnEjemplaresDisponibles() +
                ", portadaURI='" + this.getPortadaURI() + '\'' +
                '}';
    }
}
```

Y definimos la clase `Recomendacion`:

org.expertojava.jbibrest.modelo.Recomendacion

```
package org.expertojava.jbibrest.modelo;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

public class Recomendacion extends ClaseDominio {
    private Libro libro;
    private Libro libroRelacionado;
    private String comentario;
    private static Log logger = LogFactory.getLog(Recomendacion.class);

    public Recomendacion(Libro libro, Libro libroRelacionado) {
        if (libro == null) {
            logger.error("libro es null");
            throw new IllegalArgumentException("Error al crear
 recomendación: libro null");
        }
        if (libroRelacionado == null) {
            logger.error("libroRelacionado es null");
            throw new IllegalArgumentException("Error al crear
 recomendación: libroRelacionado null");
        }
        this.libro = libro;
        this.libroRelacionado = libroRelacionado;
    }

    public Libro getLibro() {
        return libro;
    }

    public void setLibro(Libro libro) {
        this.libro = libro;
    }

    public Libro getLibroRelacionado() {
        return libroRelacionado;
    }

    public void setLibroRelacionado(Libro libroRelacionado) {
        this.libroRelacionado = libroRelacionado;
    }

    public String getComentario() {
        return comentario;
    }

    public void setComentario(String comentario) {
        this.comentario = comentario;
    }
}
```

Todos los métodos se pueden generar usando el asistente de IntelliJ, con la opción *Generate…*
.

Logging

La configuración de logging se define en los siguientes ficheros. Que deben estar en el directorio `resources`, tanto de la carpeta de fuentes como de la carpeta de tests. El nivel de logging está en INFO en la carpeta de fuentes y a DEBUG en la de tests.

src/main/resources/commons-loggin.properties

```
org.apache.commons.logging.Log=org.apache.commons.logging.impl.Log4JLogger
```

src/main/resources/log4j.properties

```
# Coloca el nivel root del logger en INFO (muestra mensajes de INFO hacia
 arriba)
log4j.rootLogger=INFO, A1

# A1 se redirige a la consola
log4j.appender.A1=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.Target=System.out
log4j.appender.A1.layout=org.apache.log4j.PatternLayout
log4j.appender.A1.layout.ConversionPattern=%d{dd/MM/yyyy HH:mm:ss} - %p -
 %m %n
```

src/test/resources/log4j.properties

```
# Coloca el nivel root del logger en DEBUG (muestra mensajes de DEBUG
 hacia arriba)
log4j.rootLogger=DEBUG, A1

# A1 se redirige a la consola
log4j.appender.A1=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.Target=System.out
log4j.appender.A1.layout=org.apache.log4j.PatternLayout
log4j.appender.A1.layout.ConversionPattern=%d{dd/MM/yyyy HH:mm:ss} - %p -
 %m %n
```

Primeros tests

Definimos los dos primeros test de la clase `org.expertojava.jbibrest.model.LibroTest` en el directorio `src/test/java`, con los que probamos el funcionamiento correcto de la igualdad en la clase `Libro` y el funcionamiento correcto del constructor.

src/test/java/org/expertojava/jbibrest/modelo/LibroTest.java

```
package org.expertojava.jbibrest.modelo;

import org.junit.Test;

import java.util.HashSet;

import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertFalse;
import static org.junit.Assert.assertTrue;

public class LibroTest {
```

```
    @Test
    public void compruebaEqualsDevuelveTrueConMismoId() {
        Libro libro1 = new Libro("123456789");
        Libro libro2 = new Libro("123456789");
        libro1.setId(1L);
        libro2.setId(1L);
        assertTrue(libro1.equals(libro2));
        libro2.setId(2L);
        assertFalse(libro1.equals(libro2));
    }

    @Test
    public void compruebaValoresPorDefectoEnNuevoLibro() {
        Libro libro = new Libro("123456789");
        assertEquals(0, libro.getnEjemplares());
        assertEquals(0, libro.getnEjemplaresDisponibles());
        assertEquals(0, libro.getRecomendaciones().size());
    }
}
```

Ejecutamos el test con el botón derecho sobre la clase o el paquete y *Run tests*. También los podemos lanzar usando el *goal* correspondiente de Maven.

Jerarquía de clases `Usuario`, `Alumno`, `Profesor`

Empezamos ahora a implementar la jerarquía de clases usuario. En primer lugar la clase padre abstracta `Usuario` que hereda de `ClaseDominio` y que define los atributos comunes

src/main/java/org/expertojava/jbibrest/modelo/Usuario.java

```
package org.expertojava.jbibrest.modelo;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

import java.util.Date;
import java.util.HashSet;
import java.util.Set;

public abstract class Usuario extends ClaseDominio {
    private String login;
    private String eMail;
    private String nombre;
    private String apellido1;
    private String apellido2;

    private static Log logger = LogFactory.getLog(Usuario.class);

    public String getLogin() {
        return login;
    }

    public void setLogin(String login) {
        this.login = login;
    }
```

```
    public String geteMail() {
        return eMail;
    }

    public void seteMail(String eMail) {
        this.eMail = eMail;
    }

    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }

    public String getApellido1() {
        return apellido1;
    }

    public void setApellido1(String apellido1) {
        this.apellido1 = apellido1;
    }

    public String getApellido2() {
        return apellido2;
    }

    public void setApellido2(String apellido2) {
        this.apellido2 = apellido2;
    }
}
```

Definimos las clases hijas, junto con su constructores:

src/main/java/org/expertojava/jbibrest/modelo/Profesor.java

```
package org.expertojava.jbibrest.modelo;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

public class Profesor extends Usuario {
    private String departamento;

    private static Log logger = LogFactory.getLog(Profesor.class);

    public Profesor(String login) {
        this.setLogin(login);
    }

    public String getDepartamento() {
        return departamento;
    }

    public void setDepartamento(String departamento) {
```

```
        this.departamento = departamento;
    }

    public String toString() {
        return this.getLogin() + " : "
                + this.getNombre() + " "
                + this.getApellido1() + " "
                + this.getApellido2() + " "
                + "(PROFESOR)";
    }

}
```

src/main/java/org/expertojava/jbibrest/modelo/Alumno.java

```
package org.expertojava.jbibrest.modelo;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

public class Alumno extends Usuario {
    private String telefonoPadres;

    private static Log logger = LogFactory.getLog(Alumno.class);

    public Alumno(String login) {
        this.setLogin(login);
    }

    public String getTelefonoPadres() {
        return telefonoPadres;
    }

    public void setTelefonoPadres(String telefonoPadres) {
        this.telefonoPadres = telefonoPadres;
    }

    public String toString() {
        return this.getLogin() + " : "
                + this.getNombre() + " "
                + this.getApellido1() + " "
                + this.getApellido2() + " "
                + "(ALUMNO)";
    }
}
```

Gestión de excepciones

Es conveniente definir una clase de excepciones para encapsular en ellas todos los errores generados en la aplicación. Vamos a definir una excepción genérica de tipo *unchecked* (`BibliotecaException`), que será la excepción padre de todas las excepciones de aplicación de la biblioteca.

En principio la definimos *unchecked* porque todos los errores que vamos a capturar tienen que ver con el mal uso del API. En general, un método debe realizar su funcionalidad y

terminar correctamente cuando todo ha funcionado bien. Se lanzará una excepción si algo falla. Por ejemplo, cuando definamos un método `prestar(libro,usuario)` lanzaremos excepciones cuando no se cumplan las condiciones que hacen que el libro pueda ser prestado al usuario. Al lanzar excepciones no chequeadas permitimos que el programador chequee las condiciones antes de llamar al método y no tenga que obligatoriamente capturar una excepción que sabemos que no se va a producir. Si es necesario más adelante añadiremos una excepción *checked*.

Definimos las excepciones en el paquete `org.expertojava.jbibrest.utils`.

src/main/java/org/expertojava/jbibrest/utils/BibliotecaException.java

```
package org.expertojava.jbibrest.utils;

public class BibliotecaException extends RuntimeException {
    public static final String EJEMPLAR_NO_DISPONIBLE = "Ejemplar no
 disponible";
    public static final String NO_HAY_EJEMPLARES_DISPONIBLES = "No hay
 ejemplares disponibles";
    public static final String USUARIO_NO_EXISTENTE = "Usuario no
 existente";
    public static final String EJEMPLAR_NO_EXISTENTE = "Ejemplar no
 existente";
    public static final String USUARIO_NO_ACTIVO = "Usuario no activo";
    public static final String USUARIO_NO_TIENE_EJEMPLAR = "Usuario no
 tiene ejemplar";

    public BibliotecaException() {
        super();
    }

    public BibliotecaException(String message) {
        super(message);
    }

    public BibliotecaException(String message, Throwable cause) {
        super(message, cause);
    }
}
```

Podemos observar como, al sobrecargar el constructor con los parámetros `{String, Throwable}`, nuestra excepción permitirá su uso como *Nested Exception*. También que hemos definido unas constantes relacionadas con el dominio, que usaremos más adelante.

Métodos de utilidad

Agrupamos distintos métodos de utilidad en la clase `Utils.java`. Por ahora todo son métodos relacionados con fechas.

src/main/java/org/expertojava/jbibrest/utils/Utils.java

```
package org.expertojava.jbibrest.utils;

import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Calendar;
```

```
import java.util.Date;

public class Utils {

    static public Date stringToDate(String fechaStr) {
        Date fecha = null;
        try {
            fecha = new SimpleDateFormat("yyyy-MM-dd").parse(fechaStr);
            return fecha;
        } catch (ParseException e) {
            e.printStackTrace();
        }
        return fecha;
    }

    static public Date sumaDias(Date fecha, int dias) {
        Calendar cal = Calendar.getInstance();
        cal.setTime(fecha);
        cal.add(Calendar.DATE, dias);
        return cal.getTime();
    }

    static public long diferenciaDias(Date fecha1, Date fecha2) {
        long dif = Math.abs(fecha1.getTime() - fecha2.getTime());
        long difDias = dif / (24 * 60 * 60 * 1000);
        return difDias;
    }
}
```

Reglas de negocio

Es común agrupar las reglas de negocio de una aplicación en una o más clases (dependiendo de los diferentes subsistemas de la aplicación), para evitar que estén dispersas por la aplicación y acopladas a un gran número de clases.

En nuestro caso, vamos a crear un *Singleton*, al que llamaremos `BibliotecaBR` (BR = *Business Rules*). En principio, los valores estarán escritos directamente sobre la clase, pero en un futuro podríamos querer leer los valores de las reglas de negocio de un fichero de configuración).

El código inicial de nuestras reglas de negocio será el siguiente:

src/main/java/org/expertojava/jbibrest/utils/BibliotecaBR.java

```
package org.expertojava.jbibrest.utils;


import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.expertojava.jbibrest.modelo.*;

import java.util.Date;

/**
 * Reglas de Negocio de la Biblioteca BR = Business Rules
 * <p>
```

```
 * Lo implementamos como un singleton por si algun dia queremos leer las
 * constantes desde un fichero de configuración, lo podemos hacer desde el
 * constructor del singleton
 */
public class BibliotecaBR {
    private int numDiasPrestamoAlumno = 7;
    private int numDiasPrestamoProfesor = 30;
    private int cupoOperacionesAlumno = 5;
    private int cupoOperacionesProfesor = 8;

    private static Log logger = LogFactory.getLog(BibliotecaBR.class);

    private static BibliotecaBR me = new BibliotecaBR();

    private BibliotecaBR() {
        logger.debug("Creada instancia de "
 + this.getClass().getSimpleName());
    }

    public static BibliotecaBR getInstance() {
        return me;
    }

    /**
     * Devuelve el numero de dias de plazo que tienen un usuario para
     * devolver un prestamo (Alumno = 7 , Profesor = 30)
     *
     * @param usuario objeto Usuario
     * @return numero de dias del prestamo en función de la clase de
     * Usuario, Alumno o Profesor
     * @throws BibliotecaException el usuario no es de la clase Alumno ni
 Profesor
     */
    public int calculaNumDiasPrestamo(Usuario usuario)
            throws BibliotecaException {
        if (usuario instanceof Alumno) {
            return numDiasPrestamoAlumno;
        } else if (usuario instanceof Profesor) {
            return numDiasPrestamoProfesor;
        } else {
            String msg = "Solo los alumnos y profesores pueden tener " +
                    "libros prestados";
            logger.error(msg);
            throw new BibliotecaException(msg);
        }
    }

    /**
     * Valida que el número de días que se ha tardado en devolver
     * un libro es inferior o igual que el plazo máximo
     *
     * @param usuario objeto Usuario
     * @param numDias número de días sin devolver
     * @throws BibliotecaException fuera de plazo
     * @throws BibliotecaException el tipo del usuario no es el esperado
     */
    public void compruebaNumDiasPrestamo(Usuario usuario, int numDias)
```

```
            throws BibliotecaException {
        String msg;
        if (!(usuario instanceof Alumno)
                && !(usuario instanceof Profesor)) {
            msg = "Solo los alumnos y profesores pueden tener libros " +
                    "prestados";
            logger.error(msg);
            throw new BibliotecaException(msg);
        }
        if ((usuario instanceof Alumno && numDias >
                numDiasPrestamoAlumno) ||
                (usuario instanceof Profesor && numDias >
                        numDiasPrestamoProfesor)) {
            msg = "Devolución fuera de plazo";
            logger.error(msg);
            throw new BibliotecaException(msg);
        }
    }

    /**
     * Devuelve el número máximo de préstamos que
     * puede realizar un determinado tipo de usuario
     *
     * @param usuario objeto UsuarioDomain
     * @return número máximo de operaciones del tipo de usuario
     * @throws BibliotecaException el tipo del usuario no es el esperado
     */
    public int cupoOperaciones(Usuario usuario)
            throws BibliotecaException {
        if (usuario instanceof Alumno)
            return cupoOperacionesAlumno;
        else if (usuario instanceof Profesor)
            return cupoOperacionesProfesor;
        else {
            String msg = "Solo los alumnos y profesores pueden tener
libros prestados";
            logger.error(msg);
            throw new BibliotecaException(msg);
        }
    }

    /**
     * Valida que el número de prestamos realizadas por un determinado
     * tipo de usuario se inferior o igual al cupo definido
     *
     * @param usuario objeto Usuario
     * @param numOp   número de operación que ya tiene realizadas
     * @throws BibliotecaException el cupo de operacion esta lleno
     * @throws BibliotecaException el tipo del usuario no es el esperado
     */
    public void compruebaCupoOperaciones(Usuario usuario, int numOp)
            throws BibliotecaException {
        String msg;
        if (!(usuario instanceof Alumno)
                && !(usuario instanceof Profesor)) {
            msg = "Solo los alumnos y profesores pueden tener libros
prestados";
```

```
            logger.error(msg);
            throw new BibliotecaException(msg);
        }
        if ((usuario instanceof Alumno && numOp >
                cupoOperacionesAlumno) ||
                (usuario instanceof Profesor && numOp >
cupoOperacionesProfesor)) {
            msg = "El cupo de operaciones posibles esta lleno";
            logger.error(msg);
            throw new BibliotecaException(msg);
        }
    }

    /**
     * Devuelve una fecha de devolución de un préstamo,
     * sumando a la fecha del parámetro el número de días de préstamo
     * del tipo de usuario
     */
    public Date fechaDevolucionPrestamo(Usuario usuario, Date fechaActual)
{
        return
Utils.sumaDias(fechaActual, this.calculaNumDiasPrestamo(usuario));
    }

    /**
     * Devuelve los días de penalización de un préstamo retrasado
     */
    public long diasPenalizacion(Prestamo prestamo) {
        return Utils.diferenciaDias(prestamo.getDeberiaDevolverseEl(),
prestamo.getDevuelto());
    }

    /**
     * Comprobación de si un préstamo es moroso (está pendiente de
devolver y ya ha
     * pasado la fecha de devolución
     */

    public boolean esPrestamoActivo(Prestamo prestamo) {
        return prestamo.getDevuelto() == null;
    }

    public boolean esPrestamoRetrasado(Prestamo prestamo, Date
fechaActual) {
        return prestamo.getDeberiaDevolverseEl().before(fechaActual);
    }

    public boolean esPrestamoMoroso(Prestamo prestamo, Date fechaActual) {
        return this.esPrestamoActivo(prestamo)
&& this.esPrestamoRetrasado(prestamo, fechaActual);
    }
}
```

Ampliamos los tests

Añadimos nuevos tests que comprueban el funcionamiento de algunas reglas de negocios.

src/test/java/org/expertojava/jbibrest/modelo/BibliotecaBRTest.java

```
package org.expertojava.jbibrest.modelo;

import org.expertojava.jbibrest.utils.BibliotecaBR;
import org.expertojava.jbibrest.utils.BibliotecaException;
import org.expertojava.jbibrest.utils.Utils;
import org.junit.Test;

import java.sql.PreparedStatement;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;

import static org.junit.Assert.*;

public class BibliotecaBRTest {

    //
    // Profesor
    //

    @Test
    public void compruebaNumDiasPrestamoProfesor() {
        int diasProfesor = BibliotecaBR.getInstance()
                .calculaNumDiasPrestamo(
                        new Profesor("alan.turing"));
        assertEquals(30, diasProfesor);
    }

    @Test
    public void compruebaCupoOperacionesProfesor() {
        int cupoProfesor = BibliotecaBR.getInstance().cupoOperaciones(
                new Profesor("alan.turing"));
        assertEquals(cupoProfesor, 8);
    }

    @Test
    public void compruebaNoSaltaExcepcionCupoOperacionesProfesorCorrecto()
 {
        try {
            Profesor profesor =
                    new Profesor("alan.turing");
            BibliotecaBR.getInstance()
                    .compruebaCupoOperaciones(profesor, 8);
            BibliotecaBR.getInstance()
                    .compruebaCupoOperaciones(profesor, 1);
        } catch (BibliotecaException e) {
            fail("No debería fallar - el cupo de operaciones del" +
                    " PROFESOR es correcto");
        }
    }

    @Test(expected = BibliotecaException.class)
    public void compruebaSaltaExcepcionCupoOperacionesProfesorIncorrecto()
            throws BibliotecaException {
        BibliotecaBR.getInstance()
```

```
                .compruebaCupoOperaciones(
                        new Profesor("alan.turing"), 9);
    }

    //
    // Alumno
    //

    @Test
    public void compruebaNumDiasPrestamoAlumno() {
        int diasAlumno =
BibliotecaBR.getInstance().calculaNumDiasPrestamo(
                new Alumno("juan.perez"));
        assertEquals(7, diasAlumno);
    }

    @Test
    public void compruebaCupoOperacionesAlumno() {
        int cupoAlumno = BibliotecaBR.getInstance().cupoOperaciones(
                new Alumno("juan.perez"));
        assertEquals(cupoAlumno, 5);
    }

    @Test
    public void compruebaNoSaltaExcpecionCupoOperacionesAlumnoCorrecto() {
        try {
            Alumno alumno = new Alumno("juan.perez");

BibliotecaBR.getInstance().compruebaCupoOperaciones(alumno, 5);

BibliotecaBR.getInstance().compruebaCupoOperaciones(alumno, 1);
        } catch (BibliotecaException e) {
            fail("No debería fallar - el cupo de operaciones del ALUMNO es
correcto");
        }
    }

    @Test(expected = BibliotecaException.class)
    public void compruebaSaltaExcepcionCupoOperacionesAlumnoIncorrecto()
            throws BibliotecaException {
        BibliotecaBR.getInstance().compruebaCupoOperaciones(new Alumno
                ("juan.perez"), 6);
    }

    //
    // Test préstamos y fechas de devolución
    // Los tienes que implementar
    //
    // @Test
    // public void
compruebaFechaDevolucionPrestamoEs7DiasMasTardeParaAlumno() {}
    //
    // @Test
    // public void
compruebaDiasPenalizacionPrestamoEsDiasDeRetrasoPrestamo() {}
    //
    // @Test
```

```
    // public void
 compruebaPrestamoEsMorosoCuandoFechaFinalizacionPosteriorFechaDevolucion()
 {}
    //
    // private Prestamo creaPrestamo(Usuario usuario, String
 fechaInicioStr) {
    //    Date fechaInicio = Utils.stringToDate(fechaInicioStr);
    //    Date fechaDevolucion =
 BibliotecaBR.getInstance().fechaDevolucionPrestamo(usuario, fechaInicio);
    //    Ejemplar ejemplar = new Ejemplar("0001", 1L);
    //    Prestamo prestamo = new Prestamo(usuario, ejemplar, fechaInicio,
 fechaDevolucion);
    //    return prestamo;
    // }
}
```

Página web de prueba

Como último paso de esta parte guiada del proyecto vamos a cambiar la página frontal para probar la aplicación se despliega correctamente. Modificamos la página `index.jsp` para recoger una cadena de texto que simula un ISBN de un libro

src/webapp/index.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
    <head>
        <title>Start Page</title>
        <meta http-equiv="Content-Type" content="text/html;
 charset=UTF-8">
    </head>
    <body>
        <h1>Hello World!</h1>
        <form action="<%=request.getContextPath()%>/holamundo">
        <p>ISBN: <input type="text" name="isbn"></p>
        <input type="submit" value="Enviar">
        </form>
    </body>
</html>
```

Y cambiamos también el servlet `holamundo` para que haga una acción muy sencilla como crear un libro y mostrarlo:

src/main/java/org/expertojava/jbibrest/servlets/HolaMundo.java

```
package org.expertojava.jbibrest.servlets;


import org.expertojava.jbibrest.modelo.Libro;

import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;
```

```
import java.io.PrintWriter;

@WebServlet(name="holamundo", urlPatterns="/holamundo")
public class HolaMundo extends HttpServlet {


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

    }

    protected void doGet(HttpServletRequest request,
                         HttpServletResponse response) throws
            ServletException, IOException {
        response.setContentType("text/html");
        String isbnStr = request.getParameter("isbn");

        // Comprobamos entradas no nulas

        int errorStatus = 0;
        String errorMsg = "";


        if (isbnStr == null) {
            errorStatus = HttpServletResponse.SC_BAD_REQUEST;
            errorMsg = "Faltan parámetros en la petición";
        }

        if (errorStatus == 0) {
            // Llamamos al modelo para construir la respuesta
            Libro libro = new Libro(isbnStr);
            libro.setAutor("Kent Beck");
            libro.setTitulo("Extreme Programming Explained");
            PrintWriter out = response.getWriter();
            out.println("<!DOCTYPE HTML PUBLIC \"" +
                    "-//W3C//DTD HTML 4.0 " +
                    "Transitional//EN\">");
            out.println("<HTML>");
            out.println("<BODY>");
            out.println("<h1>Creado un nuevo libro</h1>");
            out.println("<p>");
            out.println(libro.toString());
            out.println("</p>");
            out.println("</BODY>");
            out.println("</HTML");
        }
        // errorStatus > 0
        else  {
            response.setStatus(errorStatus);
            PrintWriter out = response.getWriter();
            out.println(errorMsg);
        }
    }
}
```

Con esto terminamos la parte guiada.

Los tests estarán funcionando correctamente y la página web de prueba deberá mostrar el libro creado correctamente. Puedes borrar las clases `Nombre`, `Genero` y `NombreTest` del principio de la clase.

2.7. Desarrollo e implementación (parte no guiada, 1 punto)

1. Para terminar el ejercicio debes implementar el resto del modelo de dominio:

   - Completa el resto de clases del dominio, ampliando la clase `Usuario` con las nuevas entidades

     # Las relaciones *X-a-muchos* las definimos del tipo `Set`. De esta forma nos aseguramos que no existen objetos duplicados en las relaciones. La identidad en un conjunto se define con el método `equals` de sus elementos (definido anteriormente).

   - Define la clase `Direccion` como una clase normal (no una subclase de `ClaseDomino`)

2. Completa los tests de las clases y de `BibliotecaBR`.

   En los tests de las clases de dominio debemos comprobar las relaciones de igualdad, errores de inicialización y algunas funcionalidades relacionadas con las reglas de negocio. Por ejemplo, a continuación puedes ver el test de la clase `Prestamo`:

   src/test/java/org/expertojava/jbibrest/modelo/UsuarioTest.java

```
package org.expertojava.jbibrest.modelo;

import javafx.scene.Parent;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.expertojava.jbibrest.utils.BibliotecaBR;
import org.expertojava.jbibrest.utils.Utils;
import org.junit.Test;

import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;

import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertFalse;
import static org.junit.Assert.assertTrue;

public class UsuarioTest {

    private static Log logger = LogFactory.getLog(UsuarioTest.class);

    @Test
    public void compruebaEqualsDevuelveTrueConMismoId() {
        Usuario usuario1 = new Alumno("juan.perez");
        Usuario usuario2 = new Alumno("ana.garcia");
        usuario1.setId(1L);
        usuario2.setId(1L);
        assertTrue(usuario1.equals(usuario2));
        usuario2.setId(2L);
        assertFalse(usuario1.equals(usuario2));
```

```
    }

    @Test
    public void compruebaEqualsDevuelveFalseConDistintoTipo() {
        Usuario usuario1 = new Alumno("juan.perez");
        Usuario usuario2 = new Profesor("alan.turing");
        usuario1.setId(1L);
        usuario2.setId(1L);
        assertFalse(usuario1.equals(usuario2));
    }

    @Test
    public void compruebaGetPrestamosDevolveColeccionVacia() {
        Usuario usuario = new Profesor("alan.turing");
        assertEquals(usuario.getPrestamos().size(), 0);
    }

    @Test
    public void compruebaUsuarioConMultaTieneEstadoMultado() {
        Usuario usuario = new Alumno("juan.perez");
        Date fechaActual = new Date();

assertTrue(usuario.getEstado(fechaActual).equals(EstadoUsuario.ACTIVO));
        Date fechaFutura =
BibliotecaBR.getInstance().fechaDevolucionPrestamo(usuario,
fechaActual);
        Multa multa = new Multa(usuario, fechaActual, fechaFutura);
        usuario.setMulta(multa);

assertTrue(usuario.getEstado(fechaActual).equals(EstadoUsuario.MULTADO));
    }

    @Test
    public void compruebaUsuarioConPrestamoVencidoTieneEstadoMoroso() {
        Usuario usuario = new Alumno("juan.perez");
        Prestamo prestamo = creaPrestamo(usuario, "2015-10-01");
        // La fecha de devolución será 10-01 más 7 días -> 2015-10-08
        usuario.getPrestamos().add(prestamo);
        // Fecha actual posterior a la de devolución
        Date fechaActual = Utils.stringToDate("2015-10-10");

assertTrue(usuario.getEstado(fechaActual).equals(EstadoUsuario.MOROSO));
    }

    private Prestamo creaPrestamo(Usuario usuario, String
fechaInicioStr) {
        Date fechaInicio = Utils.stringToDate(fechaInicioStr);
        Date fechaDevolucion =
BibliotecaBR.getInstance().fechaDevolucionPrestamo(usuario,
fechaInicio);
        Ejemplar ejemplar = new Ejemplar("0001", 1L);
        Prestamo prestamo = new Prestamo(usuario, ejemplar,
fechaInicio, fechaDevolucion);
        return prestamo;
    }
}
```

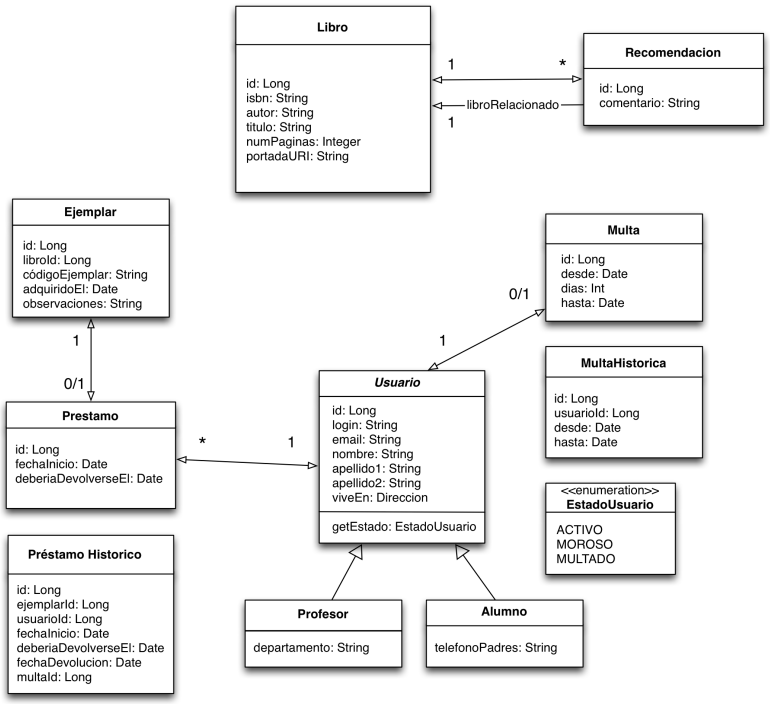
A continuación puedes ver una imagen de cómo queda el aspecto del panel del proyecto cuando hayas terminado el ejercicio:

3. (2,5 puntos) Capa de persistencia y capa de negocio

En esta segunda sesión del proyecto web, una vez definidas las clases de modelo de la biblioteca, vamos a desarrollar la capa de persistencia de la aplicación usando JPA y clases DAO y la capa de métodos de negocio utilizando beans gestionados CDIs.

Recordemos que el objetivo de la aplicación es gestionar una biblioteca de un centro educativo y proporcionar distintas funcionalidades a los usuarios de la biblioteca y a sus bibliotecarios.

Una modificación que hay realizar en el proyecto resultante de la fase anterior: hay que eliminar los atributos `nEjemplares` y `nEjemplaresDisponibles` las clases `Libro`. Así se independiza totalmente el *agregado* `Libro` y `Recomendacion` del *agregado* `Usuario`, `Multa`, `Prestamo` y `Ejemplar`. También eran campos redundantes: el número de ejemplares totales y disponibles los podemos obtener mediante una consulta a la entidad `Ejemplar`. El diagrama de clases resultante se muestra en la siguiente figura.



3.1. API del servicio

Es conveniente remarcar que el objetivo final del desarrollo de la parte del servidor es proporcionar todas las funcionalidades de la aplicación en forma de API RESTful, que implementaremos en la sesión 3.

En concreto, las funcionalidades que vamos a proporcionar son las siguientes, agrupadas por la restricción de acceso. La autentificación y la restricción de acceso la implementaremos también en la sesión 3.

Operaciones restringidas a usuarios:

- Obtener información actual de un usuario: se pasa el *login del usuario* y se obtienen sus datos.

- Obener lista de préstamos de un usuario: se pasa el *identificador del usuario* y se devuelve la colección de préstamos que tiene prestados.
- Obtener lista de libros de un usuario: se pasa el *identificador del usuario* y se devuelve la colección de libros que tiene prestados.
- Solicitar un préstamo de un libro: se pasa el *identificador del usuario* y el *identificador del libro* y se realiza el préstamo de un ejemplar si el ejemplar está disponible. Se lanza un error si no hay ejemplares disponibles, si el usuario está multado o si supera el número de préstamos permitidos. Si se ha realizado el préstamo con éxito se devuelve el nuevo préstamo.
- Realizar la devolución de un ejemplar: se pasa el *identificador de usuario* y el *identificador de ejemplar*, se crea un préstamo histórico con sus datos (y se elimina el préstamo activo). Por último, se crea una multa si la devolución está fuera de plazo. Se devuelve un enumerado con el resultado de la devolución: `DEVOLUCIÓN_CORRECTA` o `DEVOLUCIÓN_FUERA_DE_PLAZO`. entidades asociado, si lo hubiera)

Operaciones no restringidas:

- Listado de libros: se devuelve la lista de todos los libros
- Búsqueda de libros por autor: se pasa una cadena de texto y se devuelve la lista de libros que tienen un autor cuyo nombre contiene esa cadena de texto
- Búsqueda de libros por título: se pasa una cadena de texto y se devuelve la lista de libros cuyo título contiene esa cadena de texto
- Búsqueda de libro por identificador: se pasa el *identificador del libro* y se devuelven los detalles del libro
- Búsqueda de libro por *ISBN*: se pasa el *ISBN del libro* y se devuelven los detalles del libro con ese ISBN
- Libros recomendados: se pasa el *identificador del libro* y un número *n* de recomendaciones deseadas. Se devuelve la lista de, como máximo, *n* recomendaciones asociadas al libro inicial.
- Número ejemplares de un libro: se pasa el *identificador del libro* y se devuelve el número de ejemplares totales de ese libro.
- Número ejemplares disponibles: se pasa el *identificador del libro* y se devuelve el número de ejemplares disponibles de ese libro.

Vamos a centrar el desarrollo de la capa de servicios y de persistencia en estas funcionalidades utilizando:

- Entidades JPA con anotaciones y *Bean Validation*
- Capa de DAOs sobre las entidades con el CRUD y las búsquedas de la entidad
- Capa de servicios con *beans gestionados* y CDIs

Realizaremos también un número considerable de pruebas:

- Pruebas con DbUnit de las entidades
- Pruebas con Arquillian de los DAO y la capa de servicio

Vamos a realizar una primera iteración guiada en la que implementaremos algunas de las funcionalidades relacionadas con libros. Después deberás implementar el resto de funcionalidades.

3.2. Iteración 1

En esta primera iteración vamos a implementar las dos funcionalidades:

- Listado de libros: se devuelve la lista de todos los libros
- Libros recomendados: se pasa el *identificador del libro* y un número *n* de recomendaciones deseadas. Se devuelve la lista de esos *n* libros que más se han prestado junto con el libro inicial.

Capa de persistencia

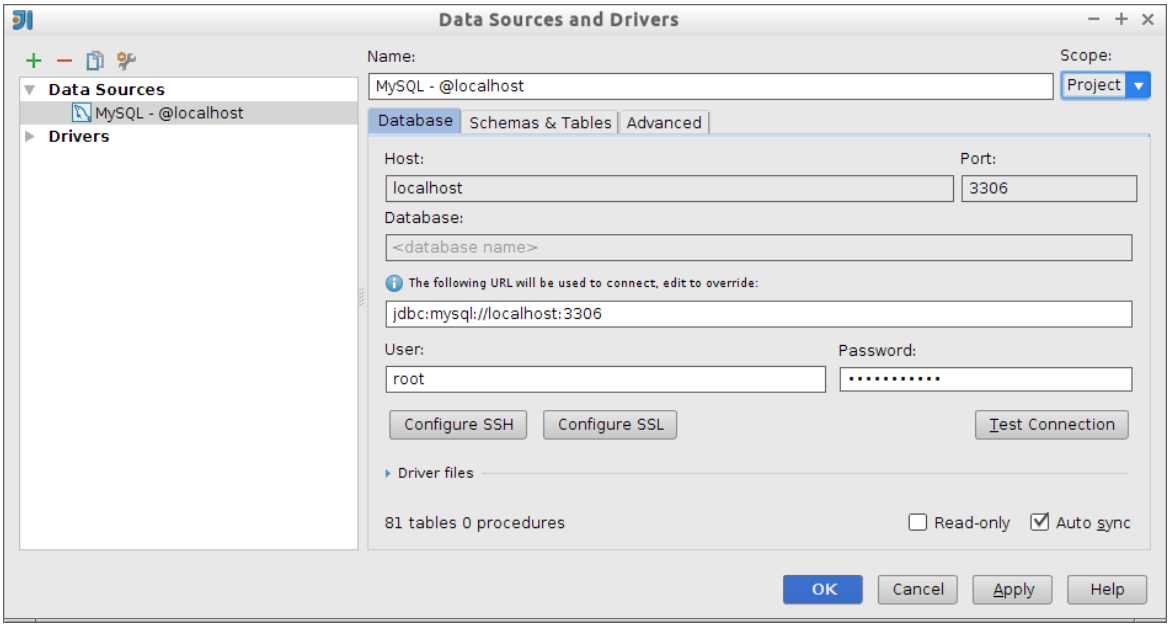
En la capa de persistencia tenemos que convertir las clases de modelo en entidades, utilizando las anotaciones JPA y *bean validation* necesarias. Crearemos también las clases DAO que encapsulan las operaciones sobre estas entidades.

Creación de la base de datos

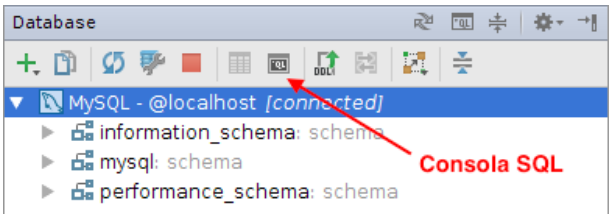
Antes de empezar a trabajar con el código debemos crear la base de datos `biblioteca` en nuestro ordenador y la fuente de datos `BibliotecaDS` en nuestro servidor WildFly en donde vamos a hacer las pruebas.

Para crear la base de datos desde IntelliJ utiliza el panel *Database* situado en el lateral derecho:

1. Abre el panel *Database* en la parte derecha.
2. Crea una nueva conexión con la base de datos MySQL con la opción + > *Data Source* > *MySQL*
3. Inicializa los parámetros de la conexión, sólo tienes que indicar el usuario `root` y la contraseña `expertojava`. Aparecerá también un aviso indicando que no está descargado el driver de acceso a MySQL, pincha el enlace y lo descargará e instalará.



Una vez configurada la conexión, vamos a utilizarla para crear la base de datos `biblioteca`. En el panel de base de datos podremos ver un desplegable con las bases de datos existentes. Para crear la nueva base de datos abre la consola SQL pulsando el icono correspondiente del panel de base de datos:



Y ejecuta el comando:

```
CREATE DATABASE biblioteca;
```

Verás que se ha creado una base de datos con ese nombre bajo las ya existentes por defecto en MySQL.

Otra forma de crear la base de datos es hacerlo desde línea de comando:

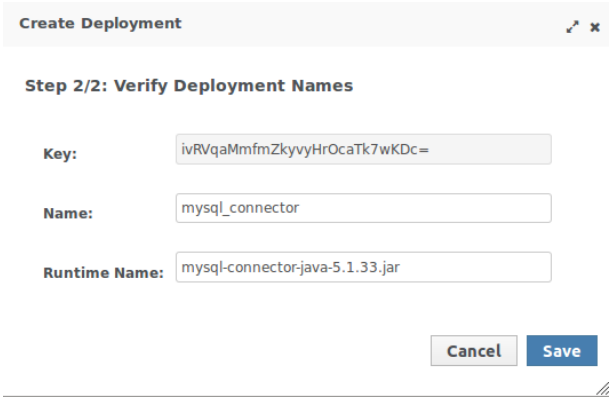
```
$ echo "CREATE DATABASE biblioteca" > create.sql
$ mysql -u root -p"expertojava" < create.sql
```

Configuración de la fuente de datos MySQL en WildFly

Recuerda que para trabajar con JPA en una aplicación web el acceso a la base de datos hay que hacerlo a través de una **fuente de datos** creada y gestionada por el servidor de aplicaciones.

Vamos a ver cómo configurar una fuente de datos MySQL en WildFly. Algunos de los pasos siguientes no serán necesarios si estás trabajando con el mismo servidor de aplicaciones con el que has hecho los módulos de JPA o de Componentes EJB. Los incluimos por completitud, por si estás trabajando con un servidor recién instalado.

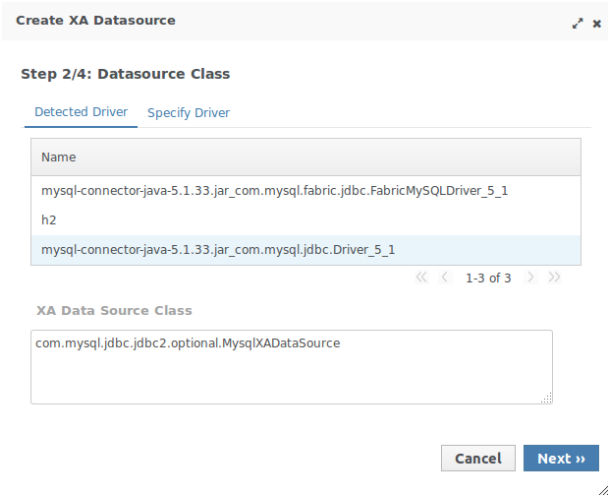
1. En primer lugar localiza el driver MySQL `mysql-connector-java-5.1.33.jar` (lo puedes encontrar en el repositorio local de Maven `.m2/repository/mysql/mysql-connector-java/5.1.33/`
2. Conéctate a la consola de administración de WildFly y selecciona la opción *Runtime* > *Manage Deployments* > *Add* y añade el JAR. Ponle como nombre *mysql_connector* (no es importante)



1. Pulsa en el botón *En/Disable* para activar el driver
2. En *Configuration* > *Connector* > *Datasources* > *XA DATASOURCES* tenemos que crear una nueva fuente de datos. Pulsa el botón *Add* e introduce los siguientes nombres:

- *Name*: `BibliotecaDS`
- *JNDI Name*: `java:/datasources/BibliotecaDS`

3. Selecciona el driver que acabamos de añadir y escribe como nombre de clase XA DataSource: `com.mysql.jdbc.jdbc2.optional.MysqlXADataSource`:

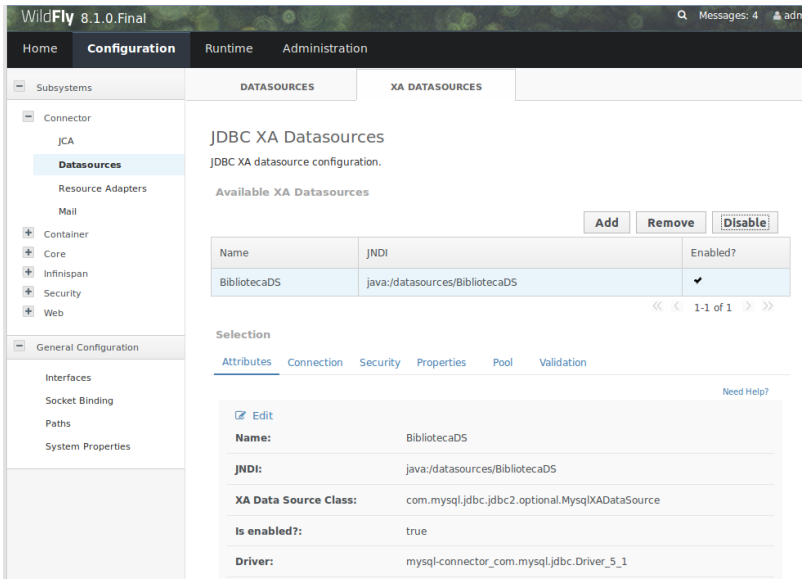


1. Añade la propiedad `URL` y el valor `jdbc:mysql://localhost:3306/biblioteca`
2. Y en la última pantalla define el usuario y la contraseña de la conexión

- *Username*: `root`
- *Password*: `expertojava`

Y prueba la conexión pulsando el botón.

1. Por último activamos la fuente de datos:



Configuración de JPA

Vamos a añadir al proyecto la configuración JPA:

- Dependencias en el POM
- Fichero `persistence.xml` en la configuración de ejecución
- Fichero `persistence.xml` en la configuración de test

Empezamos con el POM de Maven. Hay que añadir al fichero `pom.xml` las dependencias relacionadas con Hibernate y DbUnit, para poder ejecutar los tests. El fichero completo es el siguiente:

pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://
maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>org.expertojava</groupId>
    <artifactId>jbib-rest</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>war</packaging>

    <name>jbib-rest</name>

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

    <dependencies>
        <dependency>
            <groupId>javax</groupId>
            <artifactId>javaee-web-api</artifactId>
            <version>7.0</version>
            <scope>provided</scope>
        </dependency>

        <!-- Añadimos el scope provided para que no de error el
        elemento provider del persistence.xml de runtime -->

        <dependency>
            <groupId>org.hibernate</groupId>
            <artifactId>hibernate-entitymanager</artifactId>
            <version>4.3.7.Final</version>
            <scope>test,provided</scope>
        </dependency>

        <!-- Tests -->

        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.11</version>
            <scope>test</scope>
        </dependency>

        <!-- Logging -->

        <dependency>
```

```
        <groupId>commons-logging</groupId>
        <artifactId>commons-logging</artifactId>
        <version>1.2</version>
    </dependency>

    <dependency>
        <groupId>log4j</groupId>
        <artifactId>log4j</artifactId>
        <version>1.2.17</version>
        <scope>test</scope>
    </dependency>

    <dependency>
        <groupId>org.slf4j</groupId>
        <artifactId>slf4j-simple</artifactId>
        <version>1.7.12</version>
    </dependency>

    <!-- JPA -->

    <dependency>
        <groupId>org.dbunit</groupId>
        <artifactId>dbunit</artifactId>
        <version>2.5.0</version>
        <scope>test</scope>
    </dependency>

    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>5.1.33</version>
        <scope>test</scope>
    </dependency>

    <dependency>
        <groupId>org.hibernate.javax.persistence</groupId>
        <artifactId>hibernate-jpa-2.1-api</artifactId>
        <version>1.0.0.Final</version>
        <scope>test</scope>
    </dependency>

    <dependency>
        <groupId>org.slf4j</groupId>
        <artifactId>slf4j-api</artifactId>
        <version>1.7.7</version>
        <scope>test</scope>
    </dependency>

    <!-- Hibernate validator -->

    <dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-validator</artifactId>
        <version>5.1.3.Final</version>
        <scope>test</scope>
    </dependency>

    <dependency>
```

```
            <groupId>javax.el</groupId>
            <artifactId>javax.el-api</artifactId>
            <version>2.2.4</version>
            <scope>test</scope>
        </dependency>

        <dependency>
            <groupId>org.glassfish.web</groupId>
            <artifactId>el-impl</artifactId>
            <version>2.2</version>
            <scope>test</scope>
        </dependency>

    </dependencies>

    <build>
        <finalName>${project.name}</finalName>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.1</version>
                <configuration>
                    <source>1.7</source>
                    <target>1.7</target>
                </configuration>
            </plugin>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-war-plugin</artifactId>
                <version>2.3</version>
                <configuration>
                    <failOnMissingWebXml>false</failOnMissingWebXml>
                </configuration>
            </plugin>
            <plugin>
                <groupId>org.wildfly.plugins</groupId>
                <artifactId>wildfly-maven-plugin</artifactId>
                <version>1.0.2.Final</version>
                <configuration>
                    <hostname>localhost</hostname>
                    <port>9990</port>
                </configuration>
            </plugin>
        </plugins>
    </build>

</project>
```

En el fichero `src/main/resources/META-INF/persistence.xml` definimos la configuración de JPA y definimos la unidad de persistencia `biblioteca`.

Incluimos la declaración de las clases de entidad que vamos a implementar en esta primera iteración.

src/main/resources/META-INF/persistence.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0"
             xmlns="http://java.sun.com/xml/ns/
persistence" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="
         http://java.sun.com/xml/ns/persistence
         http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
    <persistence-unit name="biblioteca-datasource">

        <provider>org.hibernate.jpa.HibernatePersistenceProvider</
provider>

        <jta-data-source>java:/datasources/BibliotecaDS</jta-data-source>

        <class>org.expertojava.jbibrest.modelo.Libro</class>
        <class>org.expertojava.jbibrest.modelo.Recomendacion</class>

        <properties>

  <property name="hibernate.id.new_generator_mappings" value="false"/>
            <property name="hibernate.dialect"
                      value="org.hibernate.dialect.MySQLDialect"/>
            <property name="hibernate.hbm2ddl.auto" value="update" />
            <property name="hibernate.show_sql" value="true" />
        </properties>
    </persistence-unit>
</persistence>
```

Y en la configuración de test añadimos la configuración de JPA que se va a utilizar para lanzar las pruebas. El fichero es `src/test/resources/META-INF/persistence.xml`

src/test/resources/META-INF/persistence.xml

```
<persistence version="2.1"
             xmlns="http://xmlns.jcp.org/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence">

    <persistence-unit name="biblioteca-local"
                      transaction-type="RESOURCE_LOCAL">

        <provider>org.hibernate.jpa.HibernatePersistenceProvider</
provider>

        <class>org.expertojava.jbibrest.modelo.Libro</class>
        <class>org.expertojava.jbibrest.modelo.Recomendacion</class>

        <properties>

            <!-- JPA properties -->
            <property name="javax.persistence.jdbc.driver"
                      value="com.mysql.jdbc.Driver"/>
            <property name="javax.persistence.jdbc.url"
                      value="jdbc:mysql://localhost:3306/biblioteca"/>
            <property name="javax.persistence.jdbc.user" value="root"/>
            <property name="javax.persistence.jdbc.password"
```

```
                    value="expertojava"/>

            <!-- Hibernate properties -->
            <property name="hibernate.dialect"
                      value="org.hibernate.dialect.MySQLDialect"/>
            <property name="hibernate.show_sql" value="true"/>
            <property name="hibernate.format_sql" value="false"/>

            <property name="hibernate.hbm2ddl.auto" value="create"/>
        </properties>
    </persistence-unit>
</persistence>
```

Clases de entidades

Vamos ahora a modificar un pequeño conjunto de clases del modelo, añadiendo las anotaciones necesarias para convertirlas en clases de entidad JPA. En concreto, las clases son:

- `Libro`
- `Recomendacion`
- `ClaseDominio`

Añadiremos anotaciones JPA y anotaciones *Bean Validation*.

La clase `ClaseDominio` es la superclase de todas las entidades, en donde se define la clave primaria y los métodos `equals` y `hashCode` basados en esa clave primaria. Debemos añadir la anotación `@MappedSuperclass` para indicar a JPA que incluya sus atributos en todas las clases hijas. Además, añadimos las anotaciones relacionadas con la clave primaria.

src/main/java/org/expertojava/jbibrest/modelo/ClaseDominio.java

```
package org.expertojava.jbibrest.modelo;

import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.MappedSuperclass;

@MappedSuperclass
public abstract class ClaseDominio {
    @Id
    @GeneratedValue
    private Long id;

    protected void setId(Long id) { this.id = id; }
    public Long getId() { return id;}

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;

        ClaseDominio that = (ClaseDominio) o;

        return !(id != null ? !id.equals(that.id) : that.id != null);
```

```
    }

    @Override
    public int hashCode() {
        return id != null ? id.hashCode() : 0;
    }
}
```

Listamos a continuación las otras clases.

src/main/java/org/expertojava/jbibrest/modelo/Libro.java

```
package org.expertojava.jbibrest.modelo;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.FetchType;
import javax.persistence.OneToMany;
import javax.validation.constraints.Min;
import javax.validation.constraints.NotNull;
import java.util.HashSet;
import java.util.Set;

@Entity
public class Libro extends ClaseDominio {
    @NotNull
    @Column(unique=true, nullable = false)
    private String isbn;
    private String autor;
    private String titulo;
    @Min(0)
    private Integer numPaginas;
    private String portadaURI;
    @OneToMany(mappedBy = "libro", fetch = FetchType.EAGER)
    private Set<Recomendacion> recomendaciones = new
 HashSet<Recomendacion>();

    private static Log logger = LogFactory.getLog(Libro.class);

    public Libro() {}

    public Libro(String isbn) {
        this.isbn = isbn;
    }

    public String getIsbn() {
        return isbn;
    }

    public void setIsbn(String isbn) {
        this.isbn = isbn;
    }
```

```
    public String getAutor() {
        return autor;
    }

    public void setAutor(String autor) {
        this.autor = autor;
    }

    public String getTitulo() {
        return titulo;
    }

    public void setTitulo(String titulo) {
        this.titulo = titulo;
    }

    public Integer getNumPaginas() {
        return numPaginas;
    }

    public void setNumPaginas(Integer numPaginas) {
        this.numPaginas = numPaginas;
    }

    public String getPortadaURI() {
        return portadaURI;
    }

    public void setPortadaURI(String portadaURI) {
        this.portadaURI = portadaURI;
    }

    public Set<Recomendacion> getRecomendaciones() {
        return recomendaciones;
    }

    // Actualización de la relación a-muchos recomendaciones
    // No hay que actualizar la relación inversa, porque
    // estamos haciendo una inicialización, no un cambio, y la
    // recomendación ya está creada con el libro
    public void añadeRecomendacion(Recomendacion recomendacion) {
        this.getRecomendaciones().add(recomendacion);
    }

    @Override
    public String toString() {
        return "Libro{" +
                "id=" + this.getId() +
                ", isbn='" + this.getIsbn() + '\'' +
                ", autor='" + this.getAutor() + '\'' +
                ", titulo='" + this.getTitulo() + '\'' +
                ", numPaginas=" + this.getNumPaginas() +
                ", portadaURI='" + this.getPortadaURI() + '\'' +
                '}';
    }
}
```

src/main/java/org/expertojava/jbibrest/modelo/Recomendacion.java

```
package org.expertojava.jbibrest.modelo;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

import javax.persistence.*;
import javax.validation.constraints.NotNull;

@Entity
public class Recomendacion extends ClaseDominio {
    @NotNull
    @ManyToOne
    @JoinColumn(nullable = false)
    private Libro libro;
    @NotNull
    @OneToOne
    @JoinColumn(nullable = false)
    private Libro libroRelacionado;
    private String comentario;

    private static Log logger = LogFactory.getLog(Recomendacion.class);

    public Recomendacion() {}

    public Recomendacion(Libro libro, Libro libroRelacionado) {
        String msg;
        if (libro == null) {
            msg = "Error al crear recomendación: libro null";
            logger.error(msg);
            throw new IllegalArgumentException(msg);
        }
        if (libroRelacionado == null) {
            msg = "Error al crear recomendación: libroRelacionado null";
            logger.error(msg);
            throw new IllegalArgumentException(msg);
        }
        this.libro = libro;
        this.libroRelacionado = libroRelacionado;
    }

    public Libro getLibro() {
        return libro;
    }

    public Libro getLibroRelacionado() {
        return libroRelacionado;
    }

    public void setLibroRelacionado(Libro libroRelacionado) {
        this.libroRelacionado = libroRelacionado;
    }

    public String getComentario() {
        return comentario;
    }
```

```
    public void setComentario(String comentario) {
        this.comentario = comentario;
    }
}
```

Pruebas con DbUnit

Vamos a probar estas dos primeras entidades utilizando DbUnit. Empezamos por crear el fichero `src/test/resources/dbunit/dataset1.xml` con un conjunto de datos de prueba.

src/test/resources/dbunit/dataset1.xml

```
<?xml version='1.0' encoding='UTF-8'?>
<dataset>
    <Libro id="1" titulo="Patterns Of Enterprise Application Architecture"
           autor="Martin
 Fowler" isbn="0321127420" numPaginas="533" portadaURI="0321127420.jpg"/>
    <Libro id="2" titulo="Clean Code" autor="Robert C. Martin"
           isbn="0132350882" numPaginas="288" portadaURI="0132350882.jpg"/
>
    <Libro id="3" titulo="Test Driven Development" autor="Kent Beck"
           isbn="0321146530" numPaginas="192" portadaURI="0321146530.jpg"/
>
    <Libro id="4" titulo="Extreme Programming Explained" autor="Kent Beck"
           isbn="0321278658" numPaginas="224" portadaURI="0321278658.jpg"/
>
    <Libro id="5" titulo="The Art of Computer Programming" autor="Donald
 E. Knuth"
           isbn="0201896834" numPaginas="144" portadaURI="0321278658.jpg"/
>

    <Recomendacion id="1" libro_id="2" libroRelacionado_id = "3"/>
    <Recomendacion id="2" libro_id="2" libroRelacionado_id = "1"/>
    <Recomendacion id="3" libro_id="3" libroRelacionado_id = "2"/>
</dataset>
```

Para empezar definimos un test sencillo que carga el dataset y comprueba el método `find` de la entidad. Los creamos en el fichero `src/test/java/org.expertojava.jbibrest.persistencia.TestsLibroAggregate.java`.

src/test/java/org.expertojava.jbibrest.persistencia.TestsLibroAggregate.java

```
package org.expertojava.jbibrest.persistencia;


import org.dbunit.database.DatabaseConfig;
import org.dbunit.database.DatabaseConnection;
import org.dbunit.database.IDatabaseConnection;
import org.dbunit.dataset.IDataSet;
import org.dbunit.dataset.xml.FlatXmlDataSetBuilder;
import org.dbunit.ext.mysql.MySqlDataTypeFactory;
import org.dbunit.operation.DatabaseOperation;
import org.expertojava.jbibrest.modelo.Libro;
import org.junit.AfterClass;
```

```
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
import javax.validation.ConstraintViolationException;
import java.sql.Connection;
import java.sql.DriverManager;

import static org.junit.Assert.assertTrue;
import static org.junit.Assert.fail;

//
// Tests del aggregate formado por las entidades Libro y Recomendacion
//

public class TestsLibroAggregate {
    private static EntityManagerFactory emf;
    private static IDatabaseConnection connection;
    private static IDataSet dataset;

    @BeforeClass
    public static void initAllTests() {
        try {
            // Inicializamos sólo una vez el emf antes de todos los tests
            emf = Persistence.createEntityManagerFactory("biblioteca-
local");

            // Inicializamos la conexión a la BD necesaria para
            // que DBUnit cargue los datos de los tests
            Class.forName("com.mysql.jdbc.Driver");
            Connection jdbcConnection = (Connection) DriverManager
                    .getConnection(
                            "jdbc:mysql://localhost:3306/biblioteca",
                            "root", "expertojava");
            connection = new DatabaseConnection(jdbcConnection);

            // 2 líneas para eliminar el warning
            DatabaseConfig dbConfig = connection.getConfig();

 dbConfig.setProperty(DatabaseConfig.PROPERTY_DATATYPE_FACTORY, new
 MySqlDataTypeFactory());

            // Inicializamos el dataset
            FlatXmlDataSetBuilder flatXmlDataSetBuilder =
                    new FlatXmlDataSetBuilder();
            flatXmlDataSetBuilder.setColumnSensing(true);
            dataset = flatXmlDataSetBuilder.build(Thread.currentThread()
                    .getContextClassLoader()
                    .getResourceAsStream("dbunit/dataset1.xml"));
        } catch (Exception ex) {
            ex.printStackTrace();
            fail("Excepción al inicializar el emf y DbUnit");
        }
    }
```

```
    // Se ejecuta antes de cada test
    @Before
    public void cleanDB() throws Exception {
        // Se hace un "clean insert" de los datos de prueba
        // definidos en el fichero XML. El "clean insert" vacía las
        // tablas de los datos de prueba y después inserta los datos
        DatabaseOperation.CLEAN_INSERT.execute(connection, dataset);
    }

    /*
     * Tests de entidades Libro y Recomendacion
     */

    @Test
    public void findLibroIdTest() {
        EntityManager em = emf.createEntityManager();
        Libro libro = em.find(Libro.class, 2L);
        em.close();

        assertTrue(libro.getIsbn().equals("0132350882"));
        assertTrue(libro.getTitulo().equals("Clean Code"));
        assertTrue(libro.getAutor().equals("Robert C. Martin"));
        assertTrue(libro.getNumPaginas().equals(288));
        assertTrue(libro.getPortadaURI().equals("0132350882.jpg"));
    }

    // Se ejecuta una vez después de todos los tests
    @AfterClass
    public static void closeEntityManagerFactory() throws Exception {
        // Borramos todos los datos y cerramos la conexión
        //DatabaseOperation.DELETE_ALL.execute(connection, dataset);
        if (emf != null)
            emf.close();
    }
}
```

Nos aseguramos de que el test funciona y añadimos algunos más, que comprueben la creación de entidades, las relaciones y el *Bean Validation*:

```
    @Test
    public void createLibroTest() {
        String isbn = "123456789";

        EntityManager em = emf.createEntityManager();
        em.getTransaction().begin();
        Libro libro = new Libro(isbn);
        em.persist(libro);
        em.getTransaction().commit();

        Long libroId = libro.getId();

        em = emf.createEntityManager();
        Libro libroRecuperado = em.find(Libro.class, libroId);
        assertTrue(libroRecuperado.getIsbn().equals(isbn));
        em.close();
```

```
    }

    @Test
    public void numRecomendacionesTest() {

        EntityManager em = emf.createEntityManager();
        Libro libro = em.find(Libro.class, 2L);
        em.close();

        assertTrue(libro.getRecomendaciones().size() == 2);
    }

    @Test(expected = ConstraintViolationException.class)
    public void errorValidacionNumPaginasNegativasTest() {
        Long id = 2L;

        EntityManager em = emf.createEntityManager();
        try {
            em.getTransaction().begin();
            Libro libro = em.find(Libro.class, id);
            libro.setNumPaginas(-1);
            em.flush();
            em.getTransaction().commit();
        } catch (ConstraintViolationException ex){
            em.getTransaction().rollback();
            throw ex;
        }
        em.close();
    }
```

Clases DAO

Añadimos ahora las clases DAO correspondientes a las entidades anteriores. Lo hacemos en el paquete `org.expertojava.jbibrest.persistencia`:

- `org.expertojava.jbibrest.persistencia.Dao`
- `org.expertojava.jbibrest.persistencia.LibroDAO`
- `org.expertojava.jbibrest.persistencia.Recomendacion`

La clase abstracta `Dao`:

src/main/java/org/expertojava/jbibrest/persistencia/Dao.java

```
package org.expertojava.jbibrest.persistencia;

import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;

abstract class Dao<T, K> {

    @PersistenceContext
    EntityManager em;

    public T create(T t) {
        em.persist(t);
        em.flush();
```

```
        em.refresh(t);
        return t;
    }

    public T update(T t) {
        return (T) em.merge(t);
    }

    public void delete(T t) {
        t = em.merge(t);
        em.remove(t);
    }

    public abstract T find(K id);
}
```

La clase `LibroDao`:

src/main/java/org/expertojava/jbibrest/persistencia/LibroDao.java

```
package org.expertojava.jbibrest.persistencia;

import org.expertojava.jbibrest.modelo.Libro;

import javax.persistence.Query;
import javax.persistence.TypedQuery;
import java.util.ArrayList;
import java.util.List;

public class LibroDao extends Dao<Libro, Long> {
    String FIND_ALL_LIBROS = "SELECT l FROM Libro l ";

    @Override
    public Libro find(Long id) {
        return em.find(Libro.class, id);
    }

    public List<Libro> listAllLibros() {
        TypedQuery<Libro> query = em.createQuery(FIND_ALL_LIBROS,
 Libro.class);
        return query.getResultList();
    }
}
```

Y la clase `RecomendacionDao` en donde se guardan recomendaciones que relacionan un libro con otro

src/main/java/org/expertojava/jbibrest/persistencia/RecomendacionDao.java

```
package org.expertojava.jbibrest.persistencia;


import org.expertojava.jbibrest.modelo.Recomendacion;

import javax.persistence.Query;
import java.util.List;
```

```
public class RecomendacionDao extends Dao<Recomendacion, Long> {
    String FIND_ALL_RECOMENDACIONES = "SELECT r FROM Recomendacion r ";

    @Override
    public Recomendacion find(Long id) {
        return em.find(Recomendacion.class, id);
    }

    public List<Recomendacion> listAllRecomendaciones() {
        Query query = em.createQuery(FIND_ALL_RECOMENDACIONES);
        return (List<Recomendacion>) query.getResultList();
    }
}
```

Estos DAO los vamos a inyectar usando CDIs. Para que CDI funcione correctamente debes crear el fichero `beans.xml` vacío en el directorio `main/webapp/WEB-INF`:

src/main/webapp/WEB-INF/beans.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans
        xmlns="http://xmlns.jcp.org/xml/ns/javaee"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
                      http://xmlns.jcp.org/xml/ns/javaee/beans_1_1.xsd"
        bean-discovery-mode="all">
</beans>
```

Pruebas con Arquillian

Habrás visto que en la clase `Dao` se inyecta la unidad de persistencia. Por ello necesitamos probar las clases Dao dentro del servidor de aplicaciones, para lo que utilizaremos Arquilian. También lo utilizaremos para probar los métodos de servicio.

Para incluir Arquilian debemos seguir los siguientes pasos:

En primer lugar, añadimos las dependencias necesarias en el fichero POM del proyecto Maven:

pom.xml

```
    ...

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <version.arquillian-persistence>1.0.0.Alpha7</version.arquillian-
persistence>
    </properties>



    <dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>org.jboss.arquillian</groupId>
                <artifactId>arquillian-bom</artifactId>
```

```
                <version>1.1.10.Final</version>
                <scope>import</scope>
                <type>pom</type>
            </dependency>
        </dependencies>
    </dependencyManagement>

...

        <!-- Arquillian -->

        <dependency>
            <groupId>org.jboss.arquillian.junit</groupId>
            <artifactId>arquillian-junit-container</artifactId>
            <scope>test</scope>
        </dependency>

        <dependency>
            <groupId>org.wildfly</groupId>
            <artifactId>wildfly-arquillian-container-remote</artifactId>
            <version>8.1.0.Final</version>
            <scope>test</scope>
        </dependency>

        <!-- Arquillian persistence --> (1)

        <dependency>
            <groupId>org.jboss.arquillian.extension</groupId>
            <artifactId>arquillian-persistence-api</artifactId>
            <version>${version.arquillian-persistence}</version>
            <scope>test</scope>
        </dependency>

        <dependency>
            <groupId>org.jboss.arquillian.extension</groupId>
            <artifactId>arquillian-persistence-core</artifactId>
            <version>${version.arquillian-persistence}</version>
            <scope>test</scope>
        </dependency>

        <dependency>
            <groupId>org.jboss.arquillian.extension</groupId>
            <artifactId>arquillian-persistence-dbunit</artifactId>
            <version>${version.arquillian-persistence}</version>
            <scope>test</scope>
        </dependency>

        <dependency>
            <groupId>org.jboss.arquillian.extension</groupId>
            <artifactId>arquillian-persistence-spi</artifactId>
            <version>${version.arquillian-persistence}</version>
            <scope>test</scope>
        </dependency>

        ...
```

❶ Las librerías de *Arquillian Persistence* van a hacer posible utilizar DbUnit para cargar el dataset en la realización de los tests dentro del servidor WildFly.

Después añadimos en el directorio `test/resources` el fichero `persistence-datasource.xml` que es una copia del `persistence.xml` usado en tiempo de ejecución. Lo usaremos para que Arquillian lo copie en el war que despliega en el servidor de aplicaciones.

Configuramos el modo de cargar la base de datos con `update` para mantener los datos que introduce DbUnit en la ejecución de los tests de persistencia.

test/resources/persistence-datasource.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0"
             xmlns="http://java.sun.com/xml/ns/
persistence" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="
        http://java.sun.com/xml/ns/persistence
        http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
    <persistence-unit name="biblioteca-datasource">

        <provider>org.hibernate.jpa.HibernatePersistenceProvider</
provider>

        <jta-data-source>java:/datasources/BibliotecaDS</jta-data-source>
        <class>org.expertojava.jbibrest.modelo.Libro</class>
        <class>org.expertojava.jbibrest.modelo.Recomendacion</class>

        <properties>

  <property name="hibernate.id.new_generator_mappings" value="false"/>
            <property name="hibernate.dialect"
                      value="org.hibernate.dialect.MySQLDialect"/>
            <property name="hibernate.hbm2ddl.auto" value="create" /> ❶
            <property name="hibernate.show_sql" value="true" />
        </properties>
    </persistence-unit>
</persistence>
```

❶ Configuramos el modo de cargar la base de datos con `create` para limpiar la base de datos

Por último, definimos dos tests iniciales con Arquillian. El primero para comprobar las clases Dao se inyectan correctamente y el segundo para comprobar un primer método sencillo del DAO.

src/test/java/org.jbibrest.persistencia.TestsLibroAggregateArquillian.java

```
package org.expertojava.jbibrest.persistencia;

import org.expertojava.jbibrest.modelo.Libro;
import org.expertojava.jbibrest.modelo.Recomendacion;
import org.expertojava.jbibrest.utils.BibliotecaException;
import org.jboss.arquillian.container.test.api.Deployment;
import org.jboss.arquillian.junit.Arquillian;
import org.jboss.arquillian.persistence.Cleanup;
import org.jboss.arquillian.persistence.CleanupStrategy;
import org.jboss.arquillian.persistence.TestExecutionPhase;
```

```
import org.jboss.arquillian.persistence.UsingDataSet;
import org.jboss.shrinkwrap.api.ShrinkWrap;
import org.jboss.shrinkwrap.api.asset.EmptyAsset;
import org.jboss.shrinkwrap.api.spec.JavaArchive;
import org.junit.Test;
import org.junit.runner.RunWith;

import javax.inject.Inject;

import static org.junit.Assert.assertNotNull;
import static org.junit.Assert.assertTrue;

@RunWith(Arquillian.class)
public class TestsLibroAggregateArquillian {

    @Inject
    LibroDao libroDao;
    @Inject
    RecomendacionDao recomendacionDao;

    @Deployment
    public static JavaArchive createDeployment() {
        return ShrinkWrap.create(JavaArchive.class)
                .addPackage(Libro.class.getPackage())
                .addPackage(Recomendacion.class.getPackage())
                .addPackage(LibroDao.class.getPackage())
                .addPackage(RecomendacionDao.class.getPackage())
                .addPackage(BibliotecaException.class.getPackage())
                .addAsResource("persistence-datasource.xml",
                        "META-INF/persistence.xml")
                .addAsManifestResource(EmptyAsset.INSTANCE, "beans.xml");
    }

    @Test
    public void libroDaoNoEsNullTest() {
        assertNotNull(libroDao);
    }

    @UsingDataSet("dbunit/dataset1.xml")
    @Cleanup(phase = TestExecutionPhase.BEFORE,  strategy =
 CleanupStrategy.USED_TABLES_ONLY)
    @Test
    public void findLibroIdTest() {
        Libro libro = libroDao.find(2L);
        assertTrue(libro.getIsbn().equals("0132350882"));
        assertTrue(libro.getTitulo().equals("Clean Code"));
        assertTrue(libro.getAutor().equals("Robert C. Martin"));
        assertTrue(libro.getNumPaginas().equals(288));
        assertTrue(libro.getPortadaURI().equals("0132350882.jpg"));
    }

}
```

Pruebas de las clases Dao

Una vez configurado correctamente Arquillian realizamos algunas pruebas adicionales en los DAO, sin ser demasiado exhaustivos. Por ejemplo, podemos probar:

- Método `find`
- Métodos de actualización de las relaciones y sus inversas
- Consultas
- Restricciones *Bean Validation*

src/test/java/org/jbibrest/persistencia/TestsLibroAggregateArquillian.java

```
    @UsingDataSet("dbunit/dataset1.xml")
    @Cleanup(phase = TestExecutionPhase.BEFORE,  strategy =
CleanupStrategy.USED_TABLES_ONLY)
    @Test
    public void numRecomendacionesTest()  { ❶
        Libro libro = libroDao.find(2L);
        assertTrue(libro.getRecomendaciones().size() == 2);
    }

    @UsingDataSet("dbunit/dataset1.xml")
    @Cleanup(phase = TestExecutionPhase.BEFORE,  strategy =
CleanupStrategy.USED_TABLES_ONLY)
    @Test
    public void createLibroTest() { ❷
        String isbn = "123456789";

        Libro libro = new Libro(isbn);
        Libro libroCreado = libroDao.create(libro);
        Long libroId = libroCreado.getId();
        Libro libroRecuperado = libroDao.find(libroId);
        assertTrue(libroRecuperado.getIsbn().equals(isbn));
    }

    @UsingDataSet("dbunit/dataset1.xml")
    @Cleanup(phase = TestExecutionPhase.BEFORE,  strategy =
CleanupStrategy.USED_TABLES_ONLY)
    @Test
    public void findRecomendacionTest() { ❸
        Recomendacion recomendacion = recomendacionDao.find(1L);
        assertTrue(recomendacion.getLibro().getTitulo().equals("Clean
Code"));

assertTrue(recomendacion.getLibroRelacionado().getTitulo().equals("Test
Driven Development"));
    }

    @UsingDataSet("dbunit/dataset1.xml")
    @Cleanup(phase = TestExecutionPhase.BEFORE,  strategy =
CleanupStrategy.USED_TABLES_ONLY)
    @Test
    public void añadeRecomendacionTest() { ❹
        Libro libro3 = libroDao.find(3L);
        Libro libro1 = libroDao.find(1L);
        Recomendacion recomendacion = new Recomendacion(libro3, libro1);
```

```
        recomendacion = recomendacionDao.create(recomendacion);

        // actualizamos la relación en memoria
        libro3.añadeRecomendacion(recomendacion);

        assertTrue(libro3.getRecomendaciones().size() == 2);
        assertTrue(libro3.getRecomendaciones().contains(recomendacion));
    }
}
```

Las anotaciones de Arquillian permiten usar DbUnit desde dentro del servidor:

- `@UsingDataSet`: Inserta el conjunto de prueba indicado. También envuelve el test dentro de una transacción.
- `@Cleanup`: define el modo de limpieza de los datos de prueba. En nuestro caso, antes de cada ejecución de test se limpiarán las tablas definidas en el conjunto de prueba y se insertaran los datos datos de prueba (con el `@UsingDataSet`).

> ¡¡No te limites a copiar y pegar!! Es imprescindible que leas bien el código y entiendas qué se está probando en cada test.

Capa de lógica de negocio

Una vez definida la capa de persistencia, vamos a implementar los siguientes métodos de negocio en esta primera iteración:

- Listado de libros: se devuelve la lista de todos los libros
- Libros recomendados: se pasa el *identificador del libro* y un número *n* de recomendaciones deseadas. Se devuelve la lista de esas *n* recomendaciones (o menos, si no hay suficientes recomendaciones)

Vamos a definir los métodos de negocio como método de un bean gestionado que anotaremos con `@Transactional` para proporcional transaccionalidad en las llamdas a las distintas clases DAO.

Vamos a probar la capa de negocio de dos formas. Primero lo haremos accediendo al método desde un servlet y mostrando los resultados en una página web. Y después usaremos usaremos Arquillian.

Clase de servicio

Creamos en el paquete `org.expertojava.jbibrest.servicio` la clase `LibroServicio`, con la anotación `@Transactional` contieniendo los métodos `listaLibros()` y `listaRecomendaciones()`. Obtenemos los DAO por CDI:

src/main/java/org/expertojava/jbibrest/servicio/LibroServicio.java

```
package org.expertojava.jbibrest.servicio;

import org.expertojava.jbibrest.modelo.Ejemplar;
import org.expertojava.jbibrest.modelo.Libro;
import org.expertojava.jbibrest.modelo.Recomendacion;
import org.expertojava.jbibrest.persistencia.LibroDao;
import org.expertojava.jbibrest.persistencia.RecomendacionDao;
```

```
import javax.inject.Inject;
import javax.transaction.Transactional;
import java.util.ArrayList;
import java.util.List;

@Transactional
public class LibroServicio {
    @Inject
    LibroDao libroDao; ❶
    @Inject
    RecomendacionDao recomendacionDao; ❷

    public Libro buscaLibroPorId(Long id) {
        if (id == null) {
            throw new IllegalArgumentException("Id no puede ser null");
        }
        return libroDao.find(id);
    }

    public List<Recomendacion> listaRecomendaciones(Long libroId, int n) {
        List<Recomendacion> listaDevuelta = new ArrayList<>();
        Libro libro = libroDao.find(libroId);
        int anyadidos = 0;
        for (Recomendacion rec : libro.getRecomendaciones()) {
            if (anyadidos == n) break;
            listaDevuelta.add(rec);
            anyadidos++;
        }
        return listaDevuelta;
    }

    public List<Libro> listaLibros() {
        return libroDao.listAllLibros();
    }
}
```

<1><2> Obtenemos los DAO por inyección de dependencias

Tal y como hemos visto en el módulo de JPA, el bean gestionado obtiene los DAO por inyección de dependencias y los utiliza en los métodos de negocio. Cada método de negocio se ejecuta dentro de una transacción JTA que crea automáticamente el servidor de aplicaciones.

Prueba desde un servlet

Añadimos el fichero `src/main/webapp/index.jsp` desde donde lanzamos las peticiones a los servlets:

src/main/webapp/index.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
    <head>
        <title>Start Page</title>
        <meta http-equiv="Content-Type" content="text/html">
    </head>
```

```
    <body>
        <h1>Pruebas métodos de negocio</h1>
        <hr/>
        <p><a href="<%=request.getContextPath()%>/listalibros">Listado de
 libros</a></p>
        <hr/>
        <form action="<%=request.getContextPath()%>/listarecomendaciones">
            <p>Identificador del
 libro: <input type="text" name="libroId"></p>
            <p>Número de
 recomendaciones: <input type="text" name="numRecomendaciones"></p>
            <input type="submit" value="Enviar">
        </form>
        </form>
        <hr/>
    </body>
</html>
```

Y añadimos los dos servlets que realizan la llamada a los métodos de negocio. En primer lugar `src/main/java/org.expertojava.jbibrest.ListadoLibros`:

src/main/java/org/expertojava/jbibrest/ListadoLibros.java

```
package org.expertojava.jbibrest.servlets;

import org.expertojava.jbibrest.modelo.Libro;
import org.expertojava.jbibrest.servicio.LibroServicio;

import javax.inject.Inject;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.List;

@WebServlet(name="listaLibros", urlPatterns="/listaLibros")
public class ListadoLibros extends HttpServlet {

    @Inject
    LibroServicio libroServicio; ❶

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

    }

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

        List<Libro> listaLibros
                = libroServicio.listaLibros(); ❷
```

```
        response.setContentType("text/html");

        PrintWriter out = response.getWriter();
        out.println("<!DOCTYPE HTML PUBLIC \"" +
                "-//W3C//DTD HTML 4.0 " +
                "Transitional//EN\">");
        out.println("<HTML>");
        out.println("<BODY>");
        out.println("<ul>");
        for (Libro libro : listaLibros ) {
            out.println("<li>");
            out.println(libro.getId() + " - " + libro.getAutor() + " : " +
 libro.getTitulo());
            out.println("</li>");
        }
        out.println("</ul>");
        out.println("</BODY>");
        out.println("</HTML");
    }
}
```

❶❶❶nyección del bean gestionado
❷❷❷Llamada al método de negocio del bean

Y en segundo lugar `src/main/java/org.expertojava.jbibrest.ListaRecomendaciones`:

src/main/java/org/expertojava/jbibrest/ListaRecomendaciones.java

```
package org.expertojava.jbibrest.servlets;

import org.expertojava.jbibrest.modelo.Libro;
import org.expertojava.jbibrest.modelo.Recomendacion;
import org.expertojava.jbibrest.servicio.LibroServicio;

import javax.inject.Inject;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.List;

@WebServlet(name="listaRecomendaciones", urlPatterns="/
listaRecomendaciones")
public class ListaRecomendaciones extends HttpServlet {

    @Inject
    LibroServicio libroServicio;

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

```
    }

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

        Long libroId = Long.valueOf(request.getParameter("libroId"));
        int nRec =
 Integer.valueOf(request.getParameter("numRecomendaciones"));

        List<Recomendacion> listaRecomendaciones
                = libroServicio.listaRecomendaciones(libroId, nRec);
        Libro libro = libroServicio.buscaLibroPorId(libroId);

        response.setContentType("text/html");

        PrintWriter out = response.getWriter();
        out.println("<!DOCTYPE HTML PUBLIC \"" +
                "-//W3C//DTD HTML 4.0 " +
                "Transitional//EN\">");
        out.println("<HTML>");
        out.println("<BODY>");

        out.println("<p> El libro " + libro.getTitulo() +
                " se ha prestado junto con: </p>");
        out.println("<ul>");
        for (Recomendacion recomendacion : listaRecomendaciones ) {
            out.println("<li>");
            out.println(recomendacion.getLibroRelacionado().getTitulo());
            out.println("</li>");
        }
        out.println("</ul>");
        out.println("</BODY>");
        out.println("</HTML");
    }
}
```

Prueba desde Arquillian

También probamos los métodos de negocio desde Arquillian.

Creamos la clase `TestsLibroServicio.java` en el paquete `org.expertojava.jbibrest.servicio`:

src/test/java/org/expertojava/jbibrest/servicio/TestsLibroServicio.java

```
package org.expertojava.jbibrest.servicio;

import org.expertojava.jbibrest.modelo.Libro;
import org.expertojava.jbibrest.modelo.Recomendacion;
import org.expertojava.jbibrest.persistencia.LibroDao;
import org.expertojava.jbibrest.persistencia.RecomendacionDao;
import org.expertojava.jbibrest.utils.BibliotecaException;
import org.jboss.arquillian.container.test.api.Deployment;
import org.jboss.arquillian.junit.Arquillian;
import org.jboss.arquillian.persistence.UsingDataSet;
import org.jboss.shrinkwrap.api.ShrinkWrap;
```

```
import org.jboss.shrinkwrap.api.asset.EmptyAsset;
import org.jboss.shrinkwrap.api.spec.JavaArchive;
import org.junit.Test;
import org.junit.runner.RunWith;

import javax.inject.Inject;

import java.util.List;

import static junit.framework.Assert.assertTrue;
import static junit.framework.TestCase.assertNotNull;

@RunWith(Arquillian.class)
public class TestsLibroServicio {

    @Inject
    LibroServicio libroServicio;

    @Deployment
    public static JavaArchive createDeployment() {
        return ShrinkWrap.create(JavaArchive.class)
                .addPackage(Libro.class.getPackage())
                .addPackage(LibroDao.class.getPackage())
                .addPackage(Recomendacion.class.getPackage())
                .addPackage(RecomendacionDao.class.getPackage())
                .addPackage(LibroServicio.class.getPackage())
                .addPackage(BibliotecaException.class.getPackage())
                .addAsResource("persistence-datasource.xml",
                        "META-INF/persistence.xml")
                .addAsManifestResource(EmptyAsset.INSTANCE, "beans.xml");
    }

    @Test
    public void libroServicioNoEsNullTest() {
        assertNotNull(libroServicio);
    }

    @UsingDataSet("dbunit/dataset1.xml")
    @Test
    public void buscaLibroPorIdTest() {
        Libro libro = libroServicio.buscaLibroPorId(4L);
        assertTrue(libro.getTitulo().equals("Extreme Programming
 Explained"));
    }

    @UsingDataSet("dbunit/dataset1.xml")
    @Test
    public void listaRecomendacionesTest() {
        List<Recomendacion> recomendaciones =
 libroServicio.listaRecomendaciones(2L, 3);
        assertTrue(recomendaciones.size() == 2);
    }
}
```

3.3. Resto de la sesión

Una vez que hemos realizado de forma guiada la primera iteración, debes completar el resto del proyecto siguiendo las pautas anteriores. Dejamos en el tintero algunas funcionalidades, para no hacer demasiado extensa la aplicación.

Funcionalidades

Operaciones en la clase `LibroServicio`:

- Creación de libro: se pasan cadenas con isbn, autor y título y se crea una nueva entidad, que se devuelve.
- Búsqueda de libros por autor: se pasa una cadena de texto y se devuelve la lista de libros que tienen un autor cuyo nombre contiene esa cadena de texto.
- Búsqueda de libros por título: se pasa una cadena de texto y se devuelve la lista de libros cuyo título contiene esa cadena de texto.
- Búsqueda de libro por *ISBN*: se pasa el *ISBN del libro* y se devuelve el libro con ese ISBN o `null` si no existe.

Operaciones en la clase `EjemplarServicio`:

- Número ejemplares de un libro: se pasa el *identificador del libro* y se devuelve el número de ejemplares totales de ese libro.
- Número ejemplares disponibles: se pasa el *identificador del libro* y se devuelve el número de ejemplares disponibles de ese libro.

Operaciones en la clase `UsuarioServicio`:

- Obtener información actual de un usuario: se pasa el *login del usuario* y se obtienen sus datos (un objeto de tipo `Usuario`): su identificador, sus préstamos activos, su multa, etc.
- Obener lista de préstamos de un un usuario: se pasa el *identificador del usuario* y devuelve la colección de préstamos del usuario.
- Solicitar un préstamo de un libro: se pasa el *identificador del usuario* y el *identificador del libro* y se realiza el préstamo de un ejemplar disponible si existe alguno. Se lanza un error si no hay ejemplares disponibles, si el usuario está multado o si supera el número de préstamos permitidos. Si se ha realizado el préstamo con éxito se devuelven los datos del nuevo préstamo.
- Realizar la devolución de un ejemplar: se pasa el *identificador de ejemplar*, se cierra el préstamo (se elimina de la lista de préstamos activos y se crea un préstamo histórico) y se crea una multa si la devolución está fuera de plazo. Se devuelve un enumerado con el resultado de la devolución: `DEVOLUCIÓN_CORRECTA` o `DEVOLUCIÓN_FUERA_DE_PLAZO`.

Ejemplos de pruebas

Listamos a continuación algunos ejemplos parciales de las clases de tests.

Los datos completos de prueba:

/src/test/resources/dbunit/dataset2.xml

```
<?xml version='1.0' encoding='UTF-8'?>
<dataset>

    <!-- Libros -->

    <Libro id="1" titulo="Patterns Of Enterprise Application Architecture"
           autor="Martin
 Fowler" isbn="0321127420" numPaginas="533" portadaURI="0321127420.jpg"/>
    <Libro id="2" titulo="Clean Code" autor="Robert C. Martin"
           isbn="0132350882" numPaginas="288" portadaURI="0132350882.jpg"/
>
    <Libro id="3" titulo="Test Driven Development" autor="Kent Beck"
           isbn="0321146530" numPaginas="192" portadaURI="0321146530.jpg"/
>
    <Libro id="4" titulo="Extreme Programming Explained" autor="Kent Beck"
           isbn="0321278658" numPaginas="224" portadaURI="0321278658.jpg"/
>
    <Libro id="5" titulo="The Art of Computer Programming" autor="Donald
 E. Knuth"
           isbn="0201896834" numPaginas="144" portadaURI="0321278658.jpg"/
>

    <!-- Recomendaciones -->

    <Recomendacion id="1" libro_id="2" libroRelacionado_id = "3"/>
    <Recomendacion id="2" libro_id="2" libroRelacionado_id = "1"/>
    <Recomendacion id="3" libro_id="3" libroRelacionado_id = "2"/>

    <!-- Usuarios -->

    <Usuario id="1" tipo="PROFESOR" login="vicente.casamayor"/>
    <Usuario id="2" tipo="ALUMNO" login="antonio.perez" nombre="Antonio"
             apellido1="Pérez" apellido2="Martínez"/>
    <Usuario id="3" tipo="ALUMNO" login="anabel.garcia" nombre="Anabel"
             apellido1="Garcia" apellido2="Sierra"/>

    <!-- Ejemplares -->

    <Ejemplar id="1" codigoEjemplar="001" fechaAdquisicion="2014-10-01"
              libroId="1"/>
    <Ejemplar id="2" codigoEjemplar="002" fechaAdquisicion="2014-10-01"
              libroId="1"/>
    <Ejemplar id="3" codigoEjemplar="001" fechaAdquisicion="2014-11-01"
              libroId="2"/>
    <Ejemplar id="4" codigoEjemplar="001" fechaAdquisicion="2014-11-21"
              libroId="3"/>
    <Ejemplar id="5" codigoEjemplar="003" fechaAdquisicion="2014-10-01"
              libroId="1"/>

    <!-- Prestamos activos -->

    <Prestamo id="2" ejemplar_id="4" usuario_id="2" fecha="2014-12-01"
              deberiaDevolverseEl="2014-12-05"/>
    <Prestamo id="3" ejemplar_id="2" usuario_id="1" fecha="2014-11-01"
              deberiaDevolverseEl="2014-11-30"/>
    <Prestamo id="4" ejemplar_id="3" usuario_id="1" fecha="2014-11-01"
```

```
            deberiaDevolverseEl="2014-11-30"/>

    <!-- Multas -->

    <Multa id="1" desde="2014-12-02" hasta="2015-12-12" usuario_id="3"/>

</dataset>
```

Algunos tests de la clase `TestsUsuarioAggregate` que prueba las entidades del agregado `Usuario`:

src/test/java/org/expertojava/jbibrest/persistencia/TestsUsuarioAggregate.java

```
package org.expertojava.jbibrest.persistencia;

// Imports

//
// Tests del aggregate formado por las entidades Usuario, Prestamo
// Ejemplar y Multa
//

public class TestUsuarioAggregate {
    private static EntityManagerFactory emf;
    private static IDatabaseConnection connection;
    private static IDataSet dataset;

    @BeforeClass
    public static void initAllTests() {
        try {
            // Inicializamos sólo una vez el emf antes de todos los tests
            emf = Persistence.createEntityManagerFactory("biblioteca-
local");

            // Inicializamos la conexión a la BD necesaria para
            // que DBUnit cargue los datos de los tests
            Class.forName("com.mysql.jdbc.Driver");
            Connection jdbcConnection = DriverManager
                    .getConnection(
                            "jdbc:mysql://localhost:3306/biblioteca",
                            "root", "expertojava");
            connection = new DatabaseConnection(jdbcConnection);

            // 2 líneas para eliminar el warning
            DatabaseConfig dbConfig = connection.getConfig();

 dbConfig.setProperty(DatabaseConfig.PROPERTY_DATATYPE_FACTORY, new
 MySqlDataTypeFactory());

            // Inicializamos el dataset
            FlatXmlDataSetBuilder flatXmlDataSetBuilder =
                    new FlatXmlDataSetBuilder();
            flatXmlDataSetBuilder.setColumnSensing(true);
            dataset = flatXmlDataSetBuilder.build(Thread.currentThread()
                    .getContextClassLoader()
                    .getResourceAsStream("dbunit/dataset2.xml"));
        } catch (Exception ex) {
```

```
            ex.printStackTrace();
            fail("Excepción al inicializar el emf y DbUnit");
        }
    }

    // Se ejecuta antes de cada test
    @Before
    public void cleanDB() throws Exception {
        // Se hace un "clean insert" de los datos de prueba
        // definidos en el fichero XML. El "clean insert" vacía las
        // tablas de los datos de prueba y después inserta los datos
        DatabaseOperation.CLEAN_INSERT.execute(connection, dataset);
    }

    @Test
    public void cargaDataSetTest() {
        assertTrue(true);
    }

    @Test
    public void usuarioRecuperadoContienePrestamos() {
        EntityManager em = emf.createEntityManager();
        Usuario usuario = em.find(Usuario.class, 1L);
        em.close();
        assertTrue(usuario.getPrestamos().size() == 2);
    }

    @Test
    public void usuarioRecuperadoContieneMulta() {
        EntityManager em = emf.createEntityManager();
        Usuario usuario = em.find(Usuario.class, 3L);
        em.close();
        assertTrue(usuario.getMulta().getId() == 1L);
    }

    @Test
    public void eliminaPrestamoDeUsuarioyEjemplar() {
        EntityManager em = emf.createEntityManager();
        em.getTransaction().begin();
        Usuario usuario = em.find(Usuario.class, 1L);
        Ejemplar ejemplar = em.find(Ejemplar.class, 2L);
        Prestamo prestamo = em.find(Prestamo.class, 3L);
        usuario.quitaPrestamo(prestamo);
        ejemplar.quitaPrestamo();
        em.remove(prestamo);
        em.getTransaction().commit();
        em.close();

        // Comprobamos que la relación se ha actualizado en memoria

        assertFalse(usuario.getPrestamos().contains(prestamo));
        assertNull(ejemplar.getPrestamo());

        // Comprobamos que la relación se ha actualizado en BD

        em = emf.createEntityManager();
        prestamo = em.find(Prestamo.class, 3L);
```

```
        usuario = em.find(Usuario.class, 1L);
        ejemplar = em.find(Ejemplar.class, 2L);
        em.close();

        assertFalse(usuario.getPrestamos().contains(prestamo));
        assertNull(ejemplar.getPrestamo());
        assertNull(prestamo);
    }

    @Test(expected = BibliotecaException.class)
    public void
eliminaPrestamoDeUsuarioLanzaExcepcionCuandoPrestamoNoEsDeUsuario() {
        EntityManager em = emf.createEntityManager();
        try {
            em.getTransaction().begin();
            Usuario usuario = em.find(Usuario.class, 1L);
            Prestamo prestamo = em.find(Prestamo.class, 2L);
            usuario.quitaPrestamo(prestamo);
            em.getTransaction().commit();
        } catch (Exception ex) {
            em.getTransaction().rollback();
            throw ex;
        }
    }

    @Test
    public void
prestamoCreadoConUsuarioyEjemplarActualizaCorrectamenteEntidades() {
        EntityManager em = emf.createEntityManager();
        em.getTransaction().begin();
        Ejemplar ejemplar = em.find(Ejemplar.class, 1L);
        Usuario usuario = em.find(Usuario.class, 2L);
        Date fechaInicio = Utils.stringToDate("08-11-2015");
        Date fechaDevolucion =
BibliotecaBR.getInstance().fechaDevolucionPrestamo(usuario, fechaInicio);
        Prestamo prestamo = new Prestamo(usuario, ejemplar, fechaInicio,
fechaDevolucion);
        em.persist(prestamo);
        em.flush();
        Long prestamoId = prestamo.getId();
        usuario.añadePrestamo(prestamo);
        ejemplar.setPrestamo(prestamo);
        em.getTransaction().commit();
        em.close();

        // Comprobamos que la relación se ha actualizado en memoria

        assertTrue(ejemplar.getPrestamo().equals(prestamo));
        assertTrue(usuario.getPrestamos().contains(prestamo));

        // Comprobamos que la relación se ha actualizado en BD

        em = emf.createEntityManager();
        prestamo = em.find(Prestamo.class, prestamoId);
        ejemplar = em.find(Ejemplar.class, 1L);
        usuario = em.find(Usuario.class, 2L);
        em.close();
```

```
        assertTrue(ejemplar.getPrestamo().equals(prestamo));
        assertTrue(usuario.getPrestamos().contains(prestamo));
    }

    // Se ejecuta una vez después de todos los tests
    @AfterClass
    public static void closeEntityManagerFactory() throws Exception {
        // Borramos todos los datos y cerramos la conexión
        //DatabaseOperation.DELETE_ALL.execute(connection, dataset);
        if (emf != null)
            emf.close();
    }
}
```

La clase `TestsUsuarioAggregateArquillian.java` contiene las pruebas de los DAOs del agregado `Usuario`:

src/test/java/org/expertojava/jbibrest/persistencia/TestsUsuarioAggregateArquillian.java

```
package org.expertojava.jbibrest.persistencia;

import org.expertojava.jbibrest.modelo.*;
import org.expertojava.jbibrest.utils.BibliotecaException;
import org.jboss.arquillian.container.test.api.Deployment;
import org.jboss.arquillian.junit.Arquillian;
import org.jboss.arquillian.junit.InSequence;
import org.jboss.arquillian.persistence.Cleanup;
import org.jboss.arquillian.persistence.CleanupStrategy;
import org.jboss.arquillian.persistence.TestExecutionPhase;
import org.jboss.arquillian.persistence.UsingDataSet;
import org.jboss.shrinkwrap.api.ShrinkWrap;
import org.jboss.shrinkwrap.api.asset.EmptyAsset;
import org.jboss.shrinkwrap.api.spec.JavaArchive;
import org.junit.Test;
import org.junit.runner.RunWith;

import javax.inject.Inject;
import javax.persistence.EntityManager;

import java.util.List;

import static org.junit.Assert.*;
import static org.junit.Assert.assertFalse;
import static org.junit.Assert.assertNull;

@RunWith(Arquillian.class)
public class TestUsuarioAggregateArquillian {

    @Inject
    UsuarioDao usuarioDao;
    @Inject
    PrestamoDao prestamoDao;
    @Inject
    EjemplarDao ejemplarDao;
    @Inject
```

```
    MultaDao multaDao;

    @Deployment
    public static JavaArchive createDeployment() {
        return ShrinkWrap.create(JavaArchive.class)
                .addPackage(Alumno.class.getPackage())
                .addPackage(Profesor.class.getPackage())
                .addPackage(Direccion.class.getPackage())
                .addPackage(Multa.class.getPackage())
                .addPackage(Prestamo.class.getPackage())
                .addPackage(Usuario.class.getPackage())
                .addPackage(Dao.class.getPackage())
                .addPackage(EjemplarDao.class.getPackage())
                .addPackage(MultaDao.class.getPackage())
                .addPackage(PrestamoDao.class.getPackage())
                .addPackage(UsuarioDao.class.getPackage())
                .addPackage(BibliotecaException.class.getPackage())
                .addAsResource("persistence-datasource.xml",
                        "META-INF/persistence.xml")
                .addAsManifestResource(EmptyAsset.INSTANCE, "beans.xml");
    }

    @UsingDataSet("dbunit/dataset2.xml")
    @Cleanup(phase = TestExecutionPhase.BEFORE, strategy =
CleanupStrategy.USED_TABLES_ONLY)
    @Test
    public void daosNoSonNullTest() {
        assertNotNull(usuarioDao);
        assertNotNull(prestamoDao);
        assertNotNull(ejemplarDao);
        assertNotNull(multaDao);
    }

    @UsingDataSet("dbunit/dataset2.xml")
    @Cleanup(phase = TestExecutionPhase.BEFORE, strategy =
CleanupStrategy.USED_TABLES_ONLY)
    @Test
    public void usuarioRecuperadoContienePrestamos() {
        Usuario usuario = usuarioDao.find(1L);
        assertTrue(usuario.getPrestamos().size() == 2);
    }

    @Cleanup(phase = TestExecutionPhase.BEFORE,  strategy =
CleanupStrategy.USED_TABLES_ONLY)
    @UsingDataSet("dbunit/dataset2.xml")
    @Test
    @InSequence(1)
    public void eliminaPrestamoDeUsuarioyEjemplar() {
        Usuario usuario = usuarioDao.find(1L);
        Ejemplar ejemplar = ejemplarDao.find(2L);
        Prestamo prestamo = prestamoDao.find(3L);
        usuario.quitaPrestamo(prestamo);
        usuarioDao.update(usuario);
        ejemplar.quitaPrestamo();
        ejemplarDao.update(ejemplar);
        prestamoDao.delete(prestamo);
```

```
        // Comprobamos que la relación se ha actualizado en memoria

        assertFalse(usuario.getPrestamos().contains(prestamo));
        assertNull(ejemplar.getPrestamo());
    }

    // Ponemos la anotación NONE para que no se borren los datos de las
tablas
    // y podamos comprobar el resultado de la acción anterior en la que se
ha
    // eliminado un préstamo. Utilizamos también InSequence para asegurar
que
    // los tests se ejecutan en el orden correcto.
    @Cleanup(phase = TestExecutionPhase.NONE)
    @Test
    @InSequence(2)
    public void compruebaPrestamoEliminado() {
        Prestamo prestamo = prestamoDao.find(3L);
        Usuario usuario = usuarioDao.find(1L);
        Ejemplar ejemplar = ejemplarDao.find(2L);

        assertFalse(usuario.getPrestamos().contains(prestamo));
        assertNull(ejemplar.getPrestamo());
        assertNull(prestamo);
    }

    @Cleanup(phase = TestExecutionPhase.BEFORE,  strategy =
CleanupStrategy.USED_TABLES_ONLY)
    @UsingDataSet("dbunit/dataset2.xml")
    @Test
    @InSequence(3)
    public void eliminaMultaDeUsuario() {
        Usuario usuario = usuarioDao.find(3L);
        Multa multa = usuario.getMulta();
        assertTrue(multa.getId() == 1L);
        usuario.quitaMulta();
        usuarioDao.update(usuario);
        multaDao.delete(multa);
        assertNull(usuario.getMulta());
    }

    // Igual que antes, en el test compruebaPrestamoEliminado
    @Cleanup(phase = TestExecutionPhase.NONE)
    @Test
    @InSequence(4)
    public void compuebaMultaEliminada() {
        Multa multa = multaDao.find(1L);
        Usuario usuario = usuarioDao.find(3L);
        assertNull(usuario.getMulta());
        assertNull(multa);
    }

    // Pruebas EjemplarDao

    ...
}
```

La clase `TestsUsuarioServicio` contiene las pruebas de los métodos de negocio de `UsuarioServicio`:

src/test/java/org/expertojava/jbibrest/servicio/TestsUsuarioServicio.java

```
package org.expertojava.jbibrest.servicio;

import org.expertojava.jbibrest.modelo.*;
import org.expertojava.jbibrest.persistencia.*;
import org.expertojava.jbibrest.utils.BibliotecaBR;
import org.expertojava.jbibrest.utils.BibliotecaException;
import org.expertojava.jbibrest.utils.FechaActualStub;
import org.jboss.arquillian.container.test.api.Deployment;
import org.jboss.arquillian.junit.Arquillian;
import org.jboss.arquillian.persistence.Cleanup;
import org.jboss.arquillian.persistence.CleanupStrategy;
import org.jboss.arquillian.persistence.TestExecutionPhase;
import org.jboss.arquillian.persistence.UsingDataSet;
import org.jboss.shrinkwrap.api.ShrinkWrap;
import org.jboss.shrinkwrap.api.asset.EmptyAsset;
import org.jboss.shrinkwrap.api.spec.JavaArchive;
import org.junit.Assert;
import org.junit.Test;
import org.junit.runner.RunWith;

import javax.inject.Inject;
import java.util.List;

import static junit.framework.Assert.assertTrue;
import static junit.framework.TestCase.assertNotNull;

@RunWith(Arquillian.class)
public class TestsUsuarioServicio {

    @Inject
    UsuarioServicio usuarioServicio;

    @Deployment
    public static JavaArchive createDeployment() {
        return ShrinkWrap.create(JavaArchive.class)
                .addPackage(Alumno.class.getPackage())
                .addPackage(Profesor.class.getPackage())
                .addPackage(Direccion.class.getPackage())
                .addPackage(Multa.class.getPackage())
                .addPackage(Prestamo.class.getPackage())
                .addPackage(Usuario.class.getPackage())
                .addPackage(EjemplarDao.class.getPackage())
                .addPackage(MultaDao.class.getPackage())
                .addPackage(PrestamoDao.class.getPackage())
                .addPackage(UsuarioDao.class.getPackage())
                .addPackage(BibliotecaException.class.getPackage())
                .addPackage(UsuarioServicio.class.getPackage())
                .addAsResource("persistence-datasource.xml",
                        "META-INF/persistence.xml")
                .addAsManifestResource(EmptyAsset.INSTANCE, "beans.xml");
    }
```

```
    @UsingDataSet("dbunit/dataset2.xml")
    @Cleanup(phase = TestExecutionPhase.BEFORE, strategy =
CleanupStrategy.USED_TABLES_ONLY)
    @Test
    public void usuarioServicioNoEsNullTest() {
        assertNotNull(usuarioServicio);
    }

    @UsingDataSet("dbunit/dataset2.xml")
    @Cleanup(phase = TestExecutionPhase.BEFORE, strategy =
CleanupStrategy.USED_TABLES_ONLY)
    @Test
    public void buscaUsuarioPorIdTest() {
        Usuario usuario = usuarioServicio.buscaUsuarioPorId(1L);
        Assert.assertTrue(usuario.getPrestamos().size() == 2);
        assertNotNull(usuarioServicio);
    }

    @UsingDataSet("dbunit/dataset2.xml")
    @Cleanup(phase = TestExecutionPhase.BEFORE, strategy =
CleanupStrategy.USED_TABLES_ONLY)
    @Test
    public void recuperaUsuarioTest() {
        Usuario usuario =
usuarioServicio.recuperarUsuario("vicente.casamayor");
        Assert.assertTrue(usuario.getPrestamos().size() == 2);
        assertNotNull(usuarioServicio);
    }

    @UsingDataSet("dbunit/dataset2.xml")
    @Cleanup(phase = TestExecutionPhase.BEFORE, strategy =
CleanupStrategy.USED_TABLES_ONLY)
    @Test
    public void realizarPrestamoTest() {
        GeneradorFechaActualStub generadorFechaActual = new
GeneradorFechaActualStub(); ❶

generadorFechaActual.setFechaActual(Utils.stringToDate("2014-11-02"));

        // Actualizamos el generador de fecha actual en BibliotecaBR para
que el
        // método de negocio obtenga esa fecha

BibliotecaBR.getInstance().setFechaActualGenerator(generadorFechaActual);

        Prestamo prestamo = usuarioServicio.solicitaPrestamo(1L, 1L);
        Usuario usuario = usuarioServicio.buscaUsuarioPorId(1L);
        assertTrue(prestamo.getEjemplar().getLibroId() == 1);
        assertTrue(usuario.getPrestamos().contains(prestamo));
    }
}
```

❶ Gestión de la fecha actual para poder fijarla en los tests (ver el apartado siguiente)

Gestión de la fecha actual en los tests

La fecha actual es un elemento cambiante de cualquier aplicación. Es importante poder *fijarla* en los tests para que la ejecución de éstos no sea dependiente de la fecha del sistema. Existen varias formas de solucionar este problema. Una de ellas es definir una función auxiliar en la aplicación que devuelva la fecha del sistema por defecto y modificar esta función en los tests.

Listamos a continuación las clases que usamos.

Interfaz `GeneradorFechaActual.java`

src/main/java/org/expertojava/jbibrest/utils/GeneradorFechaActual.java

```
package org.expertojava.jbibrest.utils;

import java.util.Date;

public interface GeneradorFechaActual {
    public Date getFechaActual();
}
```

Implementación `GeneradorFechaActualReal.java`:

En el código principal de la aplicación definimos la implementación que devuelve la fecha real:

src/main/java/org/expertojava/jbibrest/utils/GeneradorFechaActualReal.java

```
package org.expertojava.jbibrest.utils;

import java.util.Date;

public class GeneradorFechaActualReal implements GeneradorFechaActual {
    public Date getFechaActual() {
        return new Date();
    }
}
```

Implementación `GeneradorFechaActualStub.java`

En el código de tests definimos la implementación *mockup* que permite fijar la fecha a la que nos interesa:

src/test/java/org/expertojava/jbibrest/utils/GeneradorFechaActualStub.java

```
package org.expertojava.jbibrest.utils;

import java.util.Date;

public class GeneradorFechaActualStub implements GeneradorFechaActual {
    Date fecha = null;

    public void setFechaActual(Date fecha) {
        this.fecha = fecha;
    }
```

```
    public Date getFechaActual()  {
       return fecha;
    }
}
```

Singleton `BibliotecaBR`

El generador de fechas lo guardamos en el *singleton* `BibliotecaBR`, y lo inicializamos por defecto al generador de la fecha real:

src/main/java/org / expertojava / jbibrest / utils / BibliotecaBR.java

```
public class BibliotecaBR {

    ...

    // Generador de fecha actual
    GeneradorFechaActual generadorFechaActual = new
 GeneradorFechaActualReal();

    ...

    public void setGeneradorFechaActual(GeneradorFechaActual
 generadorFechaActual) {
        this.generadorFechaActual = generadorFechaActual;
    }

    public GeneradorFechaActual getGeneradorFechaActual() {
        return this.generadorFechaActual;
    }
    ...
}
```

Uso en la aplicación

En la aplicación debemos obtener la fecha actual consultando el objeto guardado en `BibliotecaBR`. De esta forma, obtendremos la fecha real o la fecha fijada por el test, según en qué contexto se esté ejecutando el código:

```
GeneradorFechaActual generadorFechaActual =
 BibliotecaBR.getInstance().getGeneradorFechaActual();
Date fechaActual = generadorFechaActual.getFechaActual();
```

Actualización de la fecha en el test

En el test definimos una implementación del generador que nos permite fijar la fecha, definimos la fecha actual y lo inyectamos en `BibliotecaBR` antes de llamar al método de servicio:

```
GeneradorFechaActualStub generadorFechaActual = new
 GeneradorFechaActualStub();
generadorFechaActual.setFechaActual(Utils.stringToDate("2014-11-02"));

// Actualizamos el generador de fecha actual en BibliotecaBR para que el
```

```
// método de negocio obtenga esa fecha
BibliotecaBR.getInstance().setFechaActualGenerator(generadorFechaActual);

Prestamo prestamo = usuarioServicio.solicitaPrestamo(1L, 1L); ❶
Usuario usuario = usuarioServicio.buscaUsuarioPorId(1L);
assertTrue(prestamo.getEjemplar().getLibroId() == 1);
assertTrue(usuario.getPrestamos().contains(prestamo));
```

❶ Llamamos al método de servicio habiendo inyectado la fecha 2014-11-02 como fecha actual. Si no lo hiciéramos así, se produciría un error al intentar el usuario tomar un libro prestado, porque tiene préstamos que vencen en 2014 y si se obtuviera la fecha actual real su estado sería moroso.

3.4. Entrega

Debes completar las funcionalidades y las pruebas del proyecto. Repasamos la lista de cosas por hacer:

1. Añadir las anotaciones JPA a las clases restantes
   - `Usuario` (abstracta) y su relación de herencia con `Alumno` y `Profesor`
   - `Ejemplar`
   - `Prestamo`
   - `Multa`
   - `PrestamoHistorico`
   - `MultaHistorica`
2. Crear las clases DAO
   - `UsuarioDao`
   - `EjemplarDao`
   - `PrestamoDao`
   - `MultaDao`
   - `PrestamoHistoricoDao`
   - `MultaHistoricaDao`
3. Añadir los métodos de negocio necesarios en las clases `LibroServicio` y `UsuariServicio`
4. Deberás crear y completar los tests correspondientes para comprobar que funcionan correctamente las entidades, los DAOs y los métodos de negocio:
   - `Persistencia.LibroAggregateTests.java` - Pruebas de las entidades del agregado `Libro`
   - `Persistencia.LibroAggregateArquillianTests.java` - Pruebas de los DAO del agregado `Libro`
   - `Persistencia.UsuarioAggregateTests.java` - Pruebas de las entidades del agregado `Usuario`
   - `Persistencia.UsuarioAggregateArquillianTests.java` - Pruebas de los DAO del agregado `Usuario`

- `Servicio.LibroServicioTests.java` - Pruebas clase `LibroServicio`
- `Servicio.UsuarioServicioTests.java` - Pruebas clase `UsuarioServicio`

5. Debes también añadir servlets adicionales con los que probar manualmente la aplicación

La fecha de entrega del proyecto será el **jueves 28 de enero**.

Debes dar premiso de lectura en el repositorio `jbib-rest-expertojava` al usuario **entregas-expertojava** y confirmar la entrega en Moodle.

4. (1,5 puntos) Servicio REST

En esta sesión continuaremos trabajando con la parte del servidor del proyecto web. Implementaremos la capa REST del proyecto, utilizando JAX-RS para definir el API y basándonos en las funcionalidades implementadas en la capa de negocio de la sesión anterior. Las características básicas del API REST a desarrollar serán:

- Interfaz REST para las funcionalidades implementadas en la capa de negocio por las clases de servicio (UsuarioServicio y LibroServicio)
- Trabajamos con objetos JSON
- Implementación del login y las restricciones de seguridad usando el método BASIC
- Conversión de las excepciones generadas por las capas inferiores en códigos de error HTTP

Todas las clases necesarias las crearemos en el paquete `org.expertojava.jbibrest.rest`.

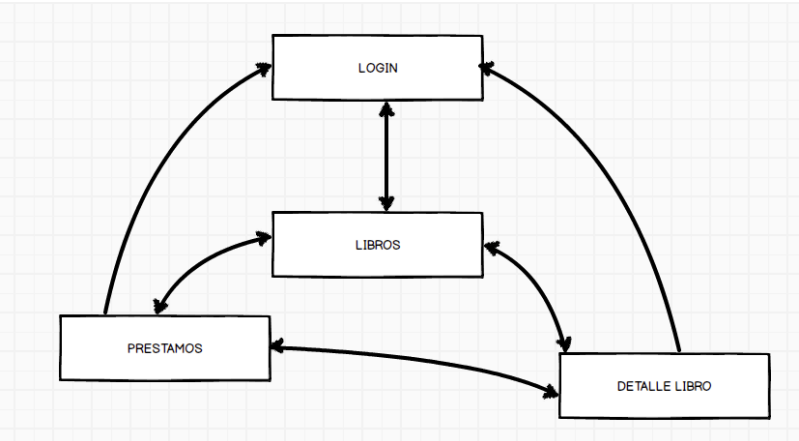
Todas las URIs de las peticiones tendrán el prefijo `/jbib-rest/api/`. Por ejemplo: `GET http://<servidor>/jbib-rest/api/libros/0321127420`, en donde `<servidor>` será el *host* y puerto desde donde se servirán las peticiones REST. Por ejemplo si ejecutamos el proyecto en nuestra máquina tendrá el valor *localhost:8080*.

4.1. Pantallas y esquema de navegación del cliente

Para poner en contexto el API REST que vamos a desarrollar, es conveniente considerar cómo va a ser utilizada desde la aplicación cliente. La aplicación cliente tendrá las siguientes pantallas:

- Login: pantalla inicial en la que accederemos a la biblioteca mediante un login y *password*
- Listado de libros: muestra los libros pertenecientes a la biblioteca,
- Detalle de libro: muestra información detallada sobre un libro en concreto
- Listado de préstamos activos y multa activa: muestra el listado de préstamos y multa activa del usuario

Se podrá navegar entre ellas según el siguiente esquema:

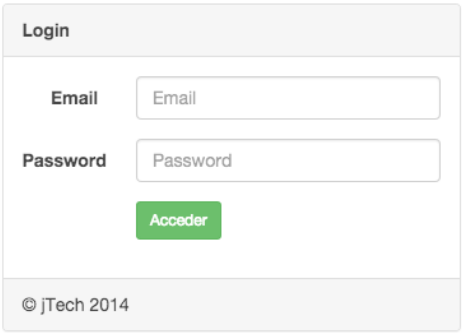


La capa rest estará formada por dos recursos: **libros** y **usuarios**, que implementarán los servicios a los que se accederá desde las pantallas anteriores.

Pasemos ahora a comentar cómo se podrá "navegar" por dichas pantallas, según la información que proporcionan los servicios rest.

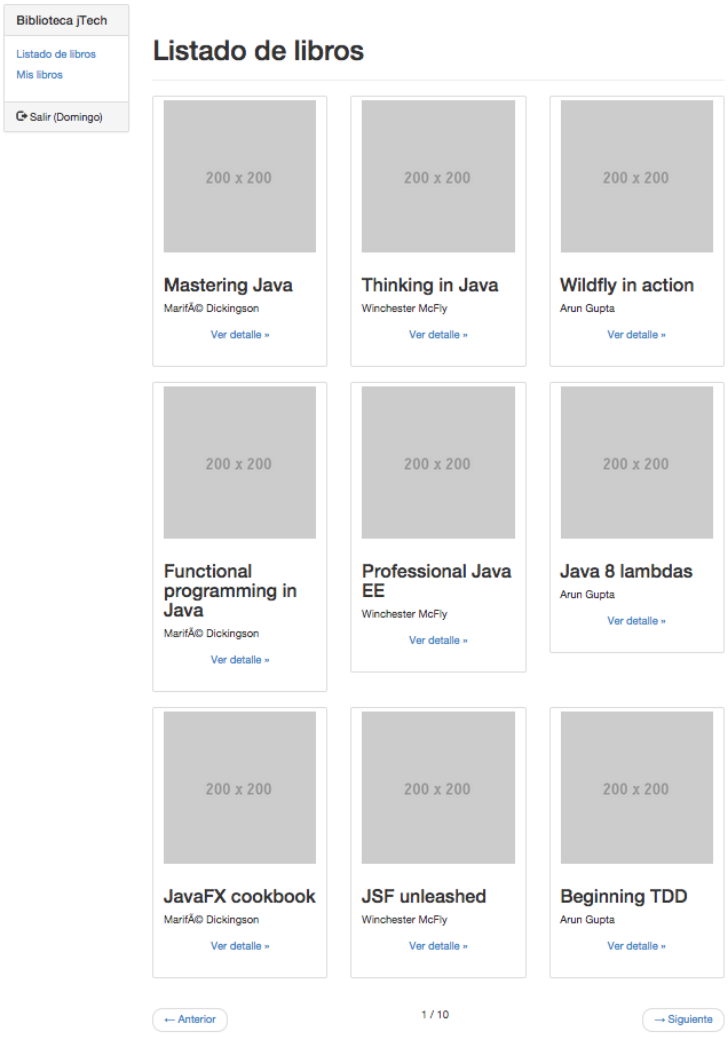
Acceso a la bibilioteca: pantalla de login

Para acceder a la biblioteca, el usuario tendrá que proporcionar su login y password. Desde esta pantalla de inicio, se invocará al método `GET /usuarios/{usuario}`. `{usuario}` es el login del usuario, no su ID. Si devuelve error 401 nos mantenemos en la página. Cualquier error 401 en cualquier otra petición redirige aquí, Si devuelve información del usuario, vamos a la siguiente pantalla de listado de libros.



Listado de libros

Se muestran todos los libros de la biblioteca y se hace la paginación en el cliente. La información sobre los libros la proporciona la operación `GET /libros`, como se aprecia en la siguiente figura:

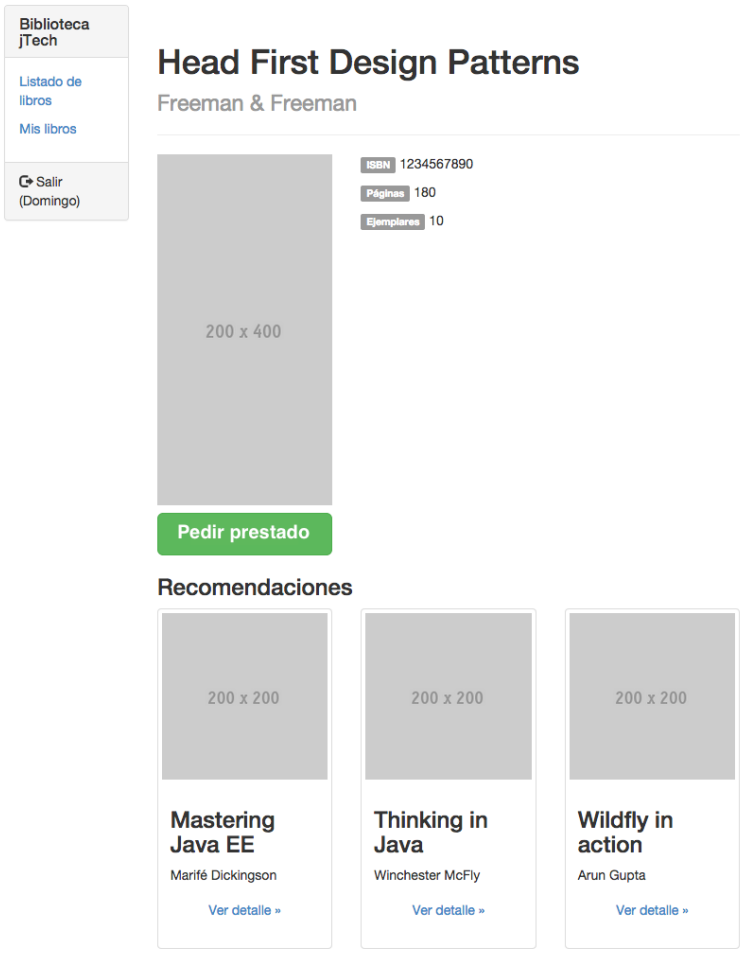


A partir de dicho listado, el usuario puede que quiera solicitar información más detallada de alguno de los libros, lo que nos llevará a la pantalla en la que mostramos el detalle de un libro.

Detalle de libro

En esta pantalla se muestran los detalles de uno de los libros, y además una lista de recomendaciones (libros recomendados por los lectores del libro mostrado). La información mencionada será proporcionada por las operaciones `GET /libros/{id}` y `GET /libros/{id}/recomendaciones?numrec=3` respectivamente. Aprovecharemos el mismo componente del listado de libros para las recomendaciones.

El botón de pedir préstamo sólo aparecerá si el libro tiene ejemplares disponibles (lo debe indicar la información devuelta por la operación `GET /libros`), y si el usuario logeado en el cliente no tiene multas pendientes. En este caso la petición de un préstamo estará implementada con una operación `POST /usuarios/{id}/prestamos` Si la petición POST a prestamos devuelve un error se mostrará un alert indicando el motivo del error.



Libros prestados y multa activa

A esta pantalla podemos acceder desde cualquiera de las anteriores. Muestra el listado de préstamos activos del usuario, así como la multa activa (si la hubiese). En la tabla, saldrán en rojo aquellos libros cuya fecha de devolución ha vencido, y en amarillo aquellos cuya fecha de devolución sea igual o inferior a tres días.

La información de esta pantalla será proporcionada por la operación `GET /usuarios/{id}/prestamos`

4.2. API REST

Las operaciones sobre el recurso **Libros** son:

- GET /libros?autor=nombre
- GET /libros?titulo=tituloLibro
- GET /libros/{id}
- GET /libros/{id}/recomendaciones?num=1

Las operaciones sobre el recurso **Usuarios** son:

- GET /usuarios/{id}
- GET /usuarios/{id}/prestamos
- POST /usuarios/{id}/prestamos
- POST /usuarios/{id}/devoluciones

Recurso **Libros**

Veamos con más detalle cada una de las operaciones sobre este recurso.

- `GET /libros`: Devuelve la lista de todos los libros. Cada ítem de libro contiene la URI poder acceder a dicho libro.

Es posible solicitar un listado de libros de un autor (`GET /libros?autor=autorLibro`). También se puede pedir solamente un listado de libros cuyo título contenga el valor del parámetro "titulo" (`GET /libros?título=tituloLibro`).

No se puede solicitar un listado por título y autor a la vez, es decir `GET /libros?titulo=tituloLibro&?autor=autorLibro`. En este caso generaremos la excepción *RestException.BUSQUEDA_POR_TITULO_Y_AUTOR_NO_IMPLEMENTADA*, con el mensaje "Búsqueda por título y autor no implementada"

Se podrá devolver una lista vacía en el caso de que no haya libros en la biblioteca

Ejemplo:

Solicitud del listado de libros del autor "Martin Fowler"

```
GET http://localhost:8080/jbib-rest/api/libros?autor=Martin%20Fowler
```

Resultado:

Resultado Json de la petición: GET http://localhost:8080/jbib-rest/api/libros?autor=Martin%20Fowler

```
[
  {
   "id": 1
   "self": "http://localhost:8080/jbib-rest/api/libros/1",
```

```
    "isbn": "0321127420",
    "titulo": "Patterns Of Enterprise Application Architecture",
    "autor": "Martin Fowler",
    "image": "http://localhost:8080/jbib-rest/media/0321127420-small.png",
  },
  {
    "id":2
    "self": "http://localhost:8080/jbib-rest/api/libros/2",
    "isbn": "0132350882",
     "titulo": "Clean code",
     "autor": "Robert C. Martin",
     "image": "http://localhost:8080/jbib-rest/media/0132350882-small.png"
   }
]
```

Cada ítem de libro incluye la url de la carátula del libro, en versión "small". Todas las carátulas estarán accesibles a través de la URL base "http://localhost:8080/jbib-rest/media/". El nombre del fichero que contiene la carátula en versión pequeña se obtiene añadiendo el sufijo "-small" al nombre del fichero del atributo `portadaURI` de la clase `Libro de la capa de negocio.

- `GET /libros/{id}`: Devuelve el detalle de un libro, contiene, además de la información anterior, el número de ejemplares totales y número de ejemplares disponibles.

Ejemplo:

Solicitud del detalle del libro con id 1

```
http://localhost:8080/jbib-rest/api/libros/1
```

Resultado:

Resultado Json de la petición: http://localhost:8080/jbib-rest/api/libros/1

```
{
    "id": 1,
    "self": "http://localhost:8080/jbib-rest/api/libros/1",
    "isbn": "0321127420",
    "titulo": "Patterns Of Enterprise Application Architecture",
    "autor": "Martin Fowler",
    "image": "http://localhost:8080/jbib-rest/media/img/0321127420.png",
    "ejemplares": 4,
    "disponibles": 2
}
```

Si el libro con id = {id} no es encontrado en la biblioteca (se obtiene un valor de libro `null`), entonces lanzaremos la excepción *RestException.LIBRO_NO_EXISTENTE*, con el mensaje "Libro no existente".

- `GET /libros/{id}/recomendaciones?num=`: Devuelve una lista de libros recomendados relacionados con un libro dado. Por defecto se muestra una recomendación, pero este número puede variar en función del parámetro de consulta `num`

Solicitud de recomendaciones del libro con id 2

```
http://localhost:8080/jbib-rest/api/libros/2/recomendaciones?num=2
```

Resultado:

Resultado Json de la petición: http://localhost:8080/jbib-rest/api/libros/2/recomendaciones?num=2

```
[
 {
   "id": 3
   "self": "http://localhost:8080/jbib-rest/api/libros/3",
   "isbn": "0321146530",
   "titulo": "Test Driven Development",
   "autor": "Kent Beck",
   "imagen": "http://expertojava.ua.es/media/0321146530-small.jpg"
 },
 {
   "id": 1
   "self": "http://localhost:8080/jbib-rest/api/libros/1",
   "isbn": "0321127420",
   "titulo": "Patterns Of Enterprise Application Architecture",
   "autor": "Martin Fowler",
   "imagen": "http://expertojava.ua.es/media/0321127420-small.jpg"
 }
]
```

Si no hay recomendaciones para el libro referenciado en la petición, se mostrará una lista vacía

Si el libro con id = {id} no es encontrado en la biblioteca (se obtiene un valor de libro `null`), entonces lanzaremos la excepción `404 Not Found`, con el mensaje "Libro no existente"

Recurso **Usuario**

Veamos con más detalle cada una de las operaciones sobre este recurso.

- `GET /usuarios/{id}`: Devuelve información detallada de un usuario, dado su login. Se trata de una operación restringida al usuario. El usuario debe estar registrado en el sistema, por lo que se requiere enviar la autorización en la cabecera. Utilizaremos seguridad Basic.

Ejemplo:

```
GET http://localhost:8080/jbib-rest/api/usuarios/antonio.perez
Authorization: Basic YW50b25pby5wZXJlejphbnRvbmlv
```

Resultado:

Resultado Json de la petición: GET http://localhost:8080/jbib-rest/api/usuarios/antonio.perez

```
{
  "id": 2,
  "login": "antonio.perez",
```

```
  "eMail": null,
  "nombre": "Antonio",
  "apellido1": "Perez",
  "apellido2": "Sierra",
  "multa": {
    "hasta": "2015-12-12"
  },
  "num_prestamos": 1,
  "tipo_usuario": "Alumno",
  "estado": "MULTADO"
}
```

Si el usuario con login = {usuario} no está registrado como usuario de la biblioteca y con el mismo login, lanzaremos la excepción *RestException.USUARIO_NO_ES_EL_LOGUEADO*, con el mensaje ""El usuario no es el logueado".

Si no se encuentra la información de dicho usuario, lanzaremos la excepción *RestException.USUARIO_NO_EXISTENTE*, con el mensaje "Usuario no existente".

- `GET /usuarios/{id}/prestamos`: Devuelve la lista de libros que tiene prestados el usuario. Para cada libro se indica la URI con la que acceder a dicho recurso. Se trata de una operación restringida al usuario. El usuario debe estar registrado en el sistema, por lo que se requiere enviar la autorización en la cabecera. Utilizaremos seguridad Basic.

Ejemplo:

```
GET http://localhost:8080/jbib-rest/api/usuarios/1/prestamos
Authorization: Basic dmljZW50ZS5jYXNhbWF5b3I6dmljZW50ZQ==
```

Resultado:

Resultado Json de la petición: GET http://localhost:8080/jbib-rest/api/usuarios/1/prestamos

```
[
  {
    "id_prestamo": 3,
    "fecha_prestamo": "2014-11-01,00:00",
    "fecha_devolucion": "2014-11-30,00:00",
    "libro": {
      "id": 1
      "self": "http://localhost:8080/jbib-rest/api/libros/1",
      "titulo": "Patterns Of Enterprise Application Architecture",
      "ejemplar_id": 2
    }
  },
  {
    "id_prestamo": 4,
    "fecha_prestamo": "2014-11-01,00:00",
    "fecha_devolucion": "2014-11-30,00:00",
    "libro": {
      "id": 3
      "self": "http://localhost:8080/jbib-rest/api/libros/3",
      "titulo": "Clean Code",
```

```
      "ejemplar_id": 3
    }
  }
]
```

Si el usuario con identificador = {id} no está registrado como usuario de la biblioteca y con el mismo login, lanzaremos la excepción *RestException.USUARIO_NO_ES_EL_LOGUEADO*, con el mensaje "El usuario no es el logueado".

Si no se encuentra la información de dicho usuario, lanzaremos la excepción *RestException.USUARIO_NO_EXISTENTE*, con el mensaje "Usuario no existente".

- `POST /usuarios/{id}/prestamos`: Se trata de una operación restringida al usuario. Se envía en el cuerpo de la petición el identificador del libro que se quiere tomar prestado. El usuario debe estar registrado en el sistema, por lo que se requiere enviar la autorización en la cabecera. Utilizaremos seguridad Basic.

Ejemplo:

```
POST http://localhost:8080/jbib-rest/api/usuarios/3/prestamos
Authorization: Basic YW5hYmVsLmdhcmNpYTphbmFiZWw=

Cuerpo del mensaje:
{
    "id": 1
}
```

Resultado:

Resultado Json de la petición: POST http://localhost:8080/jbib-rest/api/usuarios/3/prestamos

```
 {
  "id_prestamo": 5,
  "fecha_prestamo": "2015-12-17,19:00",
  "fecha_devolucion": "2015-12-24,19:00",
  "libro": {
    "id": "1",
    "resource_uri": "http://localhost:8080/jbib-rest/api/libros/1",
    "ejemplar_id": 1
}
```

Devolvemos en la cabecera **Location** la URI del nuevo recurso creado. Para este ejemplo tendrá el valor: `http://localhost:8080/jbib-rest/api/usuarios/3/prestamos/5`

Si el usuario con identificador = {id} no está registrado como usuario de la biblioteca y con el mismo login, lanzaremos la excepción *RestException.USUARIO_NO_ES_EL_LOGUEADO*, con el mensaje ""El usuario no es el logueado".

Si no se encuentra la información de dicho usuario, lanzaremos la excepción *RestException.USUARIO_NO_EXISTENTE*, con el mensaje "Usuario no existente".

- `POST /usuarios/{id}/devoluciones`: Restringida al usuario. Se envía por POST el identificador del ejemplar y se realiza la devolución.

  En este caso, y dado que estamos realizando una implementación parcial y preliminar, hemos optado por utilizar el método POST, aunque no vamos a devolver en la cabecera location (de momento) la URI del nuevo recurso creado.

  En el cuerpo de la respuesta devolveremos un objeto Json sencillo informando del resultado de la devolución.

Ejemplo:

```
POST http://localhost:8080/jbib-rest/api/usuarios/3/devoluciones
Authorization: Basic YW5hYmVsLmdhcmNpYTphbmFiZWw=

Cuerpo del mensaje:
{
    "id": 2
}
```

Resultado:

Resultado Json de la petición: POST http://localhost:8080/jbib-rest/api/usuarios/anabel.garcia/devoluciones

```
{
 "resultado": "DEVOLUCION_CORRECTA"
 }
```

Si nos hemos retrasado en devolver el libro, el resultado de la devolución mostrará el mensaje: "DEVOLUCION_FUERA_DE_PLAZO"

Representaciones JSON

El API REST trabajará con representaciones JSON de los objetos. Utilizaremos JAXB para mapear JSON en objetos y definiremos las clases necesarias en el paquete `org.expertojava.jbibrest.rest.dto`. (**dto** = **d**ata **t**ransfer **o**bject).

Recomendamos usar clases específicas para cada caso de uso, nombrándolas con el nombre utilizado en el modelo como prefijo, y un sufijo relacionado con el caso de uso o pantalla de la aplicación cliente en la que se van a mostrar los datos.

Por ejemplo: `LibroDetalle` para los datos devueltos por `GET /libros/{isbn}` o `LibroItem` para los elementos de la colección devuelta por `GET /libros`.

Para que los objetos de tipo `Date` se serialicen con el formato *YYYY-MM-dd* tenemos que utilizar la anotación `com.fasterxml.jackson.annotation.JsonFormat` con el atributo correspondiente de tipo *Date*, de la siguiente forma:

```
@JsonFormat(shape=JsonFormat.Shape.STRING, pattern="yyyy-MM-dd,HH:00",
 timezone="CET")
```

```
public Date fecha;
```

Para poder utilizar la anotación `JsonFormat` debemos incluir en el *pom.xml* la siguiente dependencia:

Dependencia para poder serializar el tipo Date a "yyy-MM-dd"

```
<!--Librería para serializar el tipo Date a "yyy-MM-dd" -->
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-annotations</artifactId>
    <version>2.6.4</version>
    <scope>provided</scope>
</dependency>
```

Seguridad

Utilizaremos la autenticación HTTP BASIC en cada petición restringida al usuario, enviando una cabecera `Authorization` con la cadena `usuario:contraseña` codificada en base64.

Ejemplo: `Authorization: Basic bG9naW46cGFzc3dvcmQ=`.

Tendrás que realizar los cambios necesarios en el fichero */src/main/webapp/WEB-INF/web.xml* para configurar la autentificación y autorización de los usuarios.

Recuerda que también tendrás que añadir los login y password de los usuarios en el *ApplicationRealm* de Wildfly, con el grupo "usuario" (con el comando *add-user.sh*).

Por ejemplo, y suponiendo que hemos realizado las peticiones que hemos mostrado para los recursos rest, tendríamos que haber incluido los siguientes usuarios en el ApplicationRealm de Wildfly (todos ellos pertenecientes al grupo "usuario"):

- login: vicente.casamayor, password: vicente, grupo: usuario
- login: antonio.perez, password: antonio, grupo: usuario
- login: anabel.garcia, password: anabel, grupo: usuario

Excepciones

Las excepciones de la capa REST serán de tipo `RestException` (por ejemplo). Resumimos las excepciones generadas desde el recurso libro y desde el recurso usuario (tened en cuenta que NO necesariamente tenéis que haber implementado "exactamente" todas ellas, se trata de un ejemplo orientativo):

- RestException.USUARIO_NO_EXISTENTE
- RestException.USUARIO_NO_ES_EL_LOGUEADO
- RestException.LIBRO_NO_EXISTENTE
- RestException.BUSQUEDA_POR_TITULO_Y_AUTOR_NO_IMPLEMENTADA

Las excepciones de la capa Servicio son de tipo BibliotecaException. Resumimos las excepciones de dicha capa:

- BibliotecaException.USUARIO_NO_EXISTENTE
- BibliotecaException.NO_HAY_EJEMPLARES_DISPONIBLES
- BibliotecaException.EJEMPLAR_NO_EXISTENTE
- BibliotecaException.EJEMPLAR_NO_DISPONIBLE
- BibliotecaException.USUARIO_MOROSO
- BibliotecaException.USUARIO_MULTADO

Implementaremos un **mapeador de excepciones** para capturar las excepciones tanto de la capa rest como de la capa de servicio, de forma que devolvamos la respuesta HTTP adecuada. Puedes crear una clase que encapsule el código de estado, y el mensaje, tal y como se ha implementado en el proyecto s5-tienda. Dicha clase se podrá serializar a formato Json con anotaciones JAXB.

El mapeado de excepciones se podrá hacer de la siguiente forma:

Para las excepciones de la capa **rest**:

- Respuesta **404 Not found**, cuando no exista el recurso
- Respuesta **501 Not implemented**, si no se ha implementado dicho servicio
- Respuesta **401 Unauthorized**, cuando el usuario que realiza la petición no es el mismo que se ha logueado en el sistema
- Respuesta 500 Internal server error, en otro caso

Para las excepciones de la capa de **servicio**:

- Respuesta **403 Forbiden**, cuando no haya disponibilidad de ejemplares o el usuario sea moroso o esté multado
- Respuesta **404 Not found**, cuando no exista una entidad
- Respuesta **500 Internal server error**, en otro caso

4.3. Pruebas

Implementaremos algunos tests para probar los servicios. En este caso los implementaremos como "tests de integración", y se ejecutarán durante la fase "integration-test" de Maven. Ya hemos visto cómo hacer ésto en las sesiones de rest, aunque utilizando el plugin surefire (que está pensado para utilizarse con tests unitarios, en la fase test de Maven).

En este proyecto utilizaremos un nuevo plugin, el plugin *surefire*, que está pensado para trabajar con tests de integración (tests que requieren disponer del empaquetado con TODAS las "unidades" de código de nuestra aplicación, o al menos un subconjunto de ellas).

El plugin *surefire* está configurado para "reconocer" los **tests de integración**, que serán todas aquellas clases (con anotaciones JUnit) cuyo nombre siga uno de estos patrones: ***/**

IT*.java, /IT.java, y *//*ITCase.java. Por lo tanto tendremos que llamar a nuestras clases de pruebas, por ejemplo, como "LibroTestsIT" y "UsuarioTestsIT". De esta forma la clase `LibroTestsIT.java` contendrá la implementación de los tests de integración con las pruebas sobre el recurso *LibroResource*, y la clase `UsuarioTestsIT.java` implementará los tests de integración sobre el recurso *UsuarioResource*.

¿Por qué necesitamos "diferenciar" nuestros tests en diferentes tipos? pues porque, además de que se ejecutan en instantes de tiempo diferentes (los tests de la capa de persistencia se ejecutan antes de los de la capa de servicio, y éstos antes que los de la capa rest), también se ejecutan en "entornos" diferentes (los tests de la capa de servicio, por ejemplo, se ejecutan DENTRO del contenedor correspondiente, en Wildfly, mientras que los tests rest NO se ejecutan en el contenedor, sino en una máquina virtual java separada). Por lo tanto, es lógico que haya diferentes *plugins* para encargarse de la ejecución de diferentes tipos de tests.

Ya hemos visto, por ejemplo, que no necesitamos utilizar Arquillian para ejecutar los tests rest (aunque podría hacerse utilizando ciertas anotaciones de Arquillian). Los test que habéis implementado hasta ahora se ejecutan con "mvn test" (podemos hacerlo porque Arquillian se encarga de realizar las pruebas en el contenedor). Ahora ejecutaremos los tests con "verify", y tendremos que incluir el plugin *failsafe* en nuestro *pom.xml* y configurarlo para ejecutar nuestros tests REST.

Configuración de los plugins surefire, failsafe y Wilfdfly.

```
<!--Añadimos dos propiedades: skipTests, y skipITs -->
<properties>
  ...
  <skipTests>false</skipTests>
  <skipITs>false</skipITs>
</properties>

<plugins>
  ...
  <!-- configuramos el plugin surefire. Por defecto
     se ejecutan los tests unitarios en la fase "test" -->
  <plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-surefire-plugin</artifactId>
    <version>2.19</version>
    <configuration>
        <skipTests>${skipTests}</skipTests>
    </configuration>
  </plugin>

  <plugin>
    <groupId>org.wildfly.plugins</groupId>
    <artifactId>wildfly-maven-plugin</artifactId>
    <version>1.0.2.Final</version>
    <configuration>
      <hostname>localhost</hostname>
      <port>9990</port>
    </configuration>
    <!-- forzamos el despliegue después de empaquetar
            y antes de ejecutar los tests REST -->
    <executions>
      <execution>
        <id>wildfly-deploy</id>
```

```
        <phase>pre-integration-test</phase>
        <goals>
           <goal>deploy</goal>
        </goals>
      </execution>
    </executions>
  </plugin>

  <!-- ejecución de la goal failsafe:test
       durante la fase integration-test -->
  <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-failsafe-plugin</artifactId>
      <version>2.19</version>
      <configuration>
          <skipITs>${skipITs}</skipITs>
          <skipTests>false</skipTests>
      </configuration>
      <executions>
          <execution>
              <goals>
                  <goal>integration-test</goal>
                  <goal>verify</goal>
              </goals>
          </execution>
      </executions>
  </plugin>
</plugins>
```

Con esta configuración:

- si ejecutamos `mvn test` se ejecutarán todos los tests de las capas inferiores a rest.
- si ejecutamos `mvn verify` se ejecutarán TODOS los tests
- si queremos ejecutar solamente los tests rest, podemos hacerlo con el comando:

```
mvn verify -DskipTests=true -DskipITs=false"
```

Por otro lado, para implementar los tests REST necesitaremos incluir las dependencias necesarias para trabajar con el API Json, para serializar/deserializar tipos java a Json, así como para utilizar el API cliente, y *matchers* Hamcrest para realizar aserciones sobre tipos Json.

Dependencias para implementar los tests REST

```
<!--Jaxrs Api cliente -->
<dependency>
    <groupId>org.jboss.resteasy</groupId>
    <artifactId>resteasy-client</artifactId>
    <version>3.0.13.Final</version>
    <scope>test</scope>
</dependency>

<!--Librerías para serializar/deserializar json -->
<dependency>
```

```
    <groupId>org.jboss.resteasy</groupId>
    <artifactId>resteasy-jackson-provider</artifactId>
    <version>3.0.13.Final</version>
    <scope>test</scope>
</dependency>

<!--Jaxrs API json -->
<dependency>
    <groupId>org.jboss.resteasy</groupId>
    <artifactId>resteasy-json-p-provider</artifactId>
    <version>3.0.13.Final</version>
    <scope>test</scope>
</dependency>

<!--Hamcrest Json -->
<dependency>
    <groupId>uk.co.datumedge</groupId>
    <artifactId>hamcrest-json</artifactId>
    <version>0.2</version>
    <scope>test</scope>
</dependency>
```

Mostramos a continuación un ejemplo de tests (todos nuestros tests estarán en el **paquete org.expertojava.jbibrest.rest** (en la carpeta **/src/test**).

Ejemplo de test que realiza una consulta de un usuario NO registrado

```
package org.expertojava.jbibrest.rest;
...

public class UsuarioTestsIT {
...

@Test
public void consultarUsuarioNORegistrado() throws Exception {
  inicializamos_BD_con_datos("/dbunit/dataset3-rest.xml");

  String login_esperado = "vicente.casamayor";
  String mail_esperado = null;

  JsonObject respuesta_esperada =
          Json.createObjectBuilder()
                .add("status", "Unauthorized")
                .add("code", 401)
                .add("message", "El usuario no es el logueado")
                .build();

  Response respuesta = this.client
      .target("http://localhost:8080/jbib-rest/api/usuarios/
roberto.garcia")
      .request(MediaType.APPLICATION_JSON)
      .header("Authorization", "Basic
 dmljZW50ZS5jYXNhbWF5b3I6dmljZW50ZQ==")
      .get();

  String respuesta_real=
 respuesta.readEntity(JsonObject.class).toString();
```

```
   //Comprobamos que recibimos el mensaje de error correcto
   Assert.assertThat(respuesta_real,
             sameJSONAs(respuesta_esperada.toString())
                     .allowingExtraUnexpectedFields()
                     );
}
}
```

Indicamos algunos de los test que podéis implementar,

ejemplos de tests para el recurso **libro**:

- obtener el listado de todos los libros
- obtener el listado de los libros de un autor determinado
- obtener el listado de los libros con un título determinado
- obtener el listado de los libros por autor y título
- obtener los detalles de un libro
- obtener las recomendaciones de un libro (sin indicar número)
- obtener las reocomendaciones de un libro (indicando un número)
- obtener las reocomendaciones de un libro que devuelva una lista vacía

ejemplos de tests para el recurso **usuario**:

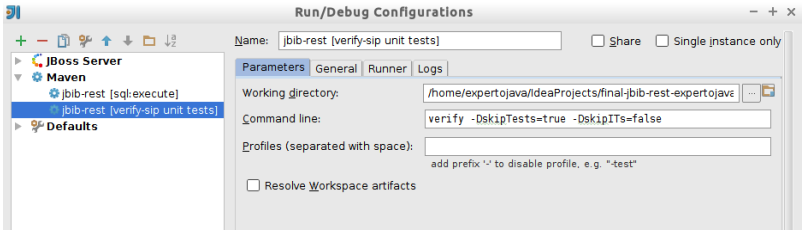
- obtener la información detallada de un usuario (sin proporcionar sus credenciales en la cabecera), de un usuario previamente registrado
- obtener la información detallada de un usuario (proporcionando sus credenciales en la cabecera), de un usuario previamente registrado
- obtener la información detallada de un usuario registrado (proporcionando las credenciales de otro usuario registrado)
- solicitar un préstamo de un libro por parte de un usuario registrado cuyo estado sea moroso
- solicitar un préstamo de un libro por parte de un usuario registrado, del que no existan ejemplares
- solicitar un préstamo de un libro por parte de un usuario registrado, cuyos ejemplares no estén disponibles
- solicitar un préstamo de un libro por parte de un usuario registrado que se lleve a cabo con éxito
- realizar una devolución de un libro que se ha prestado previamente

Para ejecutar los tests implementados desde el IDE, y por comodidad, podemos crear un perfil de ejecución, desde la ventana *Maven Projects* con botón derecho sobre la fase de Maven **verify**, de forma que se ejecute el comando:

```
mvn verify -DskipTests=true -DskipITs=false
```

esto hará que el proceso de construcción y despliegue sea más rápido, ya que nos "saltaremos" la ejecución de cualquier test de las capas inferiores a REST. Así nos centraremos únicamente en la implentación de nuestros tests REST, con independencia de que tengamos parciamente operativos los tests de las capas inferiores. Ojo! sólo evitamos que se "ejecuten" los test de capas inferiores. Podríamos habernos "saltado" también la compilación de los mismos, aunque no lo hemos hecho.

La siguiente figura muestra el configuración del perfil de ejecución mencionado, de forma que ejecutemos el comando maven anterior, simplemente seleccionando dicho perfil desde la ventana *Maven Projects*:



Finalmente, nos puede resultar útil utilizar el plugin "sql" para borrar los datos de las tablas y volver a inicializarla con datos, durante el proceso de implementación de los tests, o si por ejemplo, queremos hacer pruebas previas con postman. En este caso tendremos que añadir el siguiente plugin en el pom.xml:

Plugin sql. Utilizado para borrar e inicializar los datos de la BD

```
<!-- Plugin para inicializar la BD con comandos sql-->
<!-- Podemos crear un elemento de configuración en la ventana Maven
 Projects
     de la siguiente forma:
     - desde jbib-rest - Plugins - sql - sql:execute
     - seleccionar con botón derecho "Create 'jbib-rest...
     - editamos el perfil de configuración añadiendo en
       "Command" line el comando "sql:execute@init-database"
     Para ejecutarlo, se hace
     desde jbib-rest - Run Configurations - jbib-rest [sql:execute]
     Hay que cambiar las settings del proyecto para inidicar que la ruta
 de
     maven es: /usr/local/apache-maven-3.3.3
     (si no no funciona, ya que la versión de maven que incorpora
     IntelliJ es la 3.0.5 y tiene que ser superior a 3.1
      -->
<plugin>
    <groupId>org.codehaus.mojo</groupId>
    <artifactId>sql-maven-plugin</artifactId>
    <version>1.5</version>

    <dependencies>
        <!-- JDBC driver -->
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <version>5.1.33</version>
        </dependency>
    </dependencies>
```

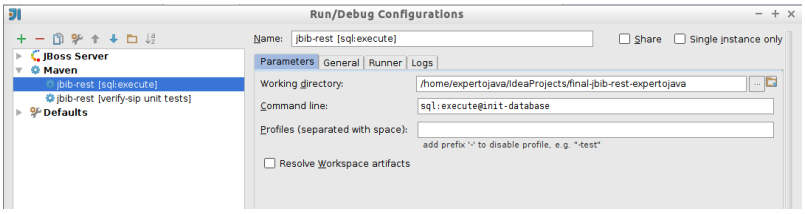
```
    <!-- common configuration shared by all executions -->
    <configuration>
        <driver>com.mysql.jdbc.Driver</driver>
        <url>jdbc:mysql://localhost:3306/biblioteca</url>
        <username>root</username>
        <password>expertojava</password>
    </configuration>

    <executions>
        <execution>
            <id>init-database</id>
            <goals>
                <goal>execute</goal>
            </goals>
            <configuration>
                <srcFiles>
                    <srcFile>src/test/resources/sql/biblioteca.sql</
srcFile>
                </srcFiles>
            </configuration>
        </execution>
    </executions>
</plugin>
```

El **fichero sql** contendrá, por ejemplo:

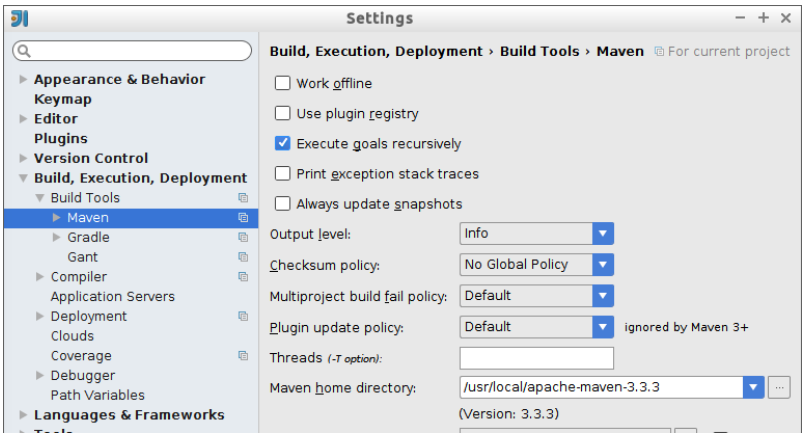
```
USE biblioteca;
DELETE FROM Recomendacion;
DELETE FROM Libro;
DELETE FROM Ejemplar;
DELETE FROM Prestamo;
DELETE FROM Usuario;

INSERT INTO Libro(id, titulo, autor,isbn, numPaginas, portadaURI)
VALUES ('1','Patterns Of Enterprise Application Architecture', 'Martin
 Fowler',
 '0321127420', '533','0321127420.jpg');
INSERT INTO Libro(id, titulo, autor,isbn, numPaginas, portadaURI)
VALUES ('2','Clean Code', 'Robert C. Martin',
 '0132350882', '288','0132350882.jpg');
INSERT INTO Libro(id, titulo, autor,isbn, numPaginas, portadaURI)
VALUES ('3','Test Driven Development', 'Kent Beck',
 '0321146530', '192','0321146530.jpg');

INSERT INTO Recomendacion(id, libro_id,libroRelacionado_id) VALUES
 ('1','2', '3');
INSERT INTO Recomendacion(id, libro_id,libroRelacionado_id) VALUES
 ('2','2', '1');
INSERT INTO Recomendacion(id, libro_id,libroRelacionado_id) VALUES
 ('3','3', '2');

INSERT INTO Ejemplar(id, codigoEjemplar, fechaAdquisicion, libroId)
VALUES ('1', '001', '2014-10-01', '1');
INSERT INTO Ejemplar(id, codigoEjemplar, fechaAdquisicion, libroId)
VALUES ('2', '002', '2014-10-01', '1');
```

```
INSERT INTO Ejemplar(id, codigoEjemplar, fechaAdquisicion, libroId)
VALUES ('3', '001', '2014-11-01', '2');
INSERT INTO Ejemplar(id, codigoEjemplar, fechaAdquisicion, libroId)
VALUES ('4', '001', '2014-11-21', '3');

INSERT INTO Usuario(id,tipo,login, nombre, apellido1, apellido2)
VALUES
 ('1', 'PROFESOR', 'vicente.casamayor', 'Vicente', 'Casamayor', 'Garcia');
INSERT INTO Usuario(id,tipo,login, nombre, apellido1, apellido2)
VALUES ('2', 'ALUMNO', 'antonio.perez', 'Antonio', 'Perez', 'Sierra');
INSERT INTO Usuario(id,tipo,login, nombre, apellido1, apellido2)
VALUES ('3', 'ALUMNO', 'anabel.garcia', 'Anabel', 'Garcia', 'Sierra');

INSERT INTO Prestamo(id,ejemplar_id,usuario_id,fecha,deberiaDevolverseEl)
VALUES ('2', '4', '2', '2014-12-01', '2014-12-05');
INSERT INTO Prestamo(id,ejemplar_id,usuario_id,fecha,deberiaDevolverseEl)
VALUES ('3', '2', '1', '2014-11-01', '2014-11-30');
INSERT INTO Prestamo(id,ejemplar_id,usuario_id,fecha,deberiaDevolverseEl)
VALUES ('4', '3', '1', '2014-11-01', '2014-11-30');
```

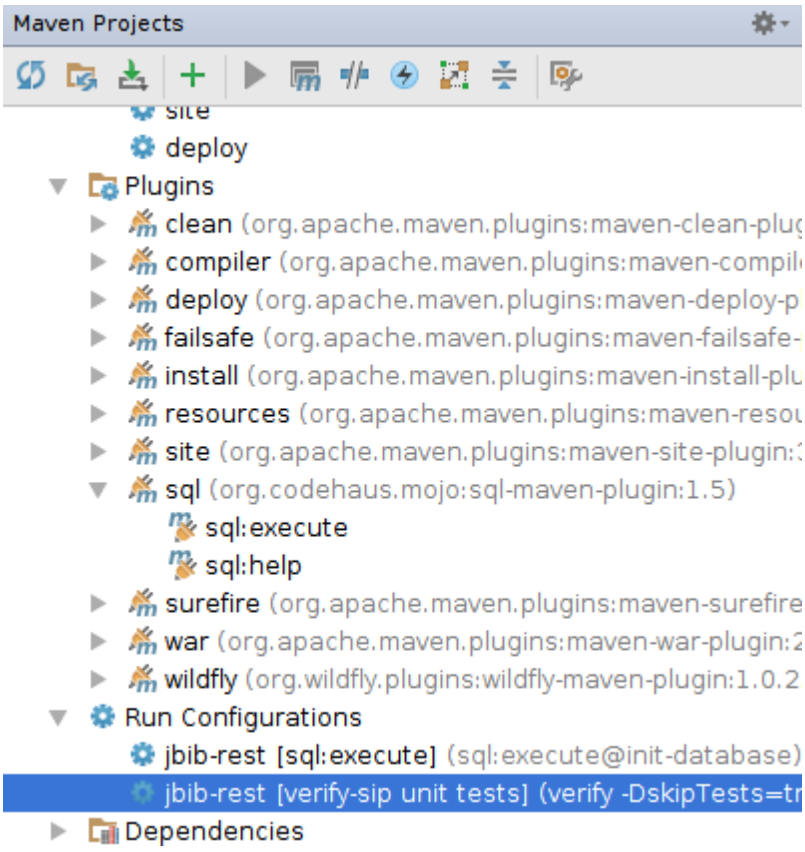
Igual que antes, podemos crear el siguiente perfil de configuración desde ventana *Maven Projects*. Así podremos "recrear" los datos de la base de datos en cualquier momento seleccionando dicho perfil:



Para poder ejecutar el nuevo perfil creado, tenemos que asegurarnos de que la versión de Maven que se ejecute desde el IDE es una versión superior a la 3.1. Esto podremos hacerlo desde la ventana "Settings":



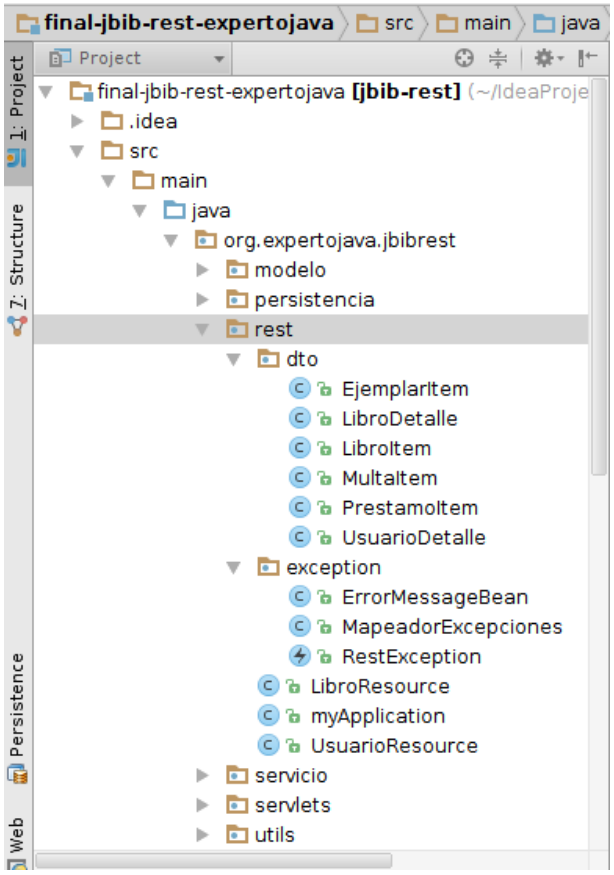
A continuación mostramos el aspecto de nuestra ventana *Maven Projects* después de añadir los dos perfiles nuevos de ejecución anteriores:



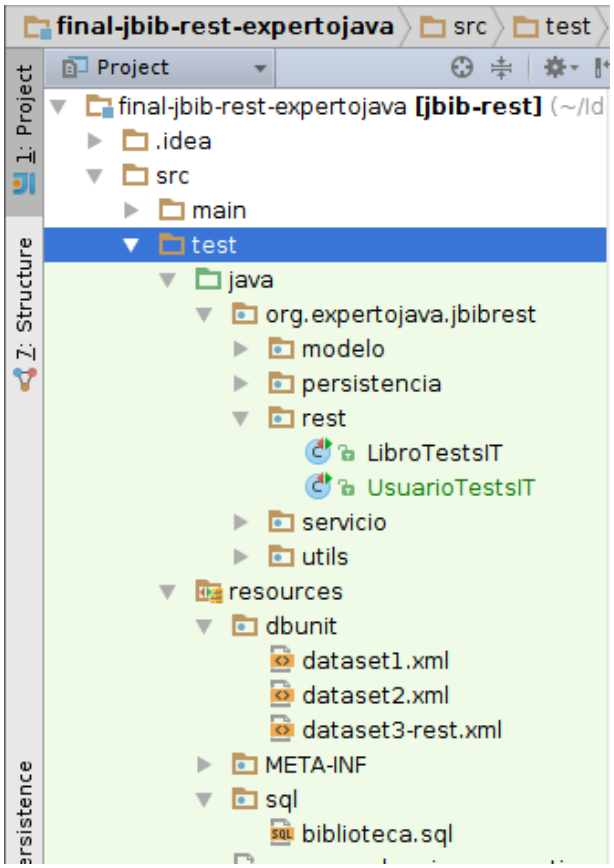
4.4. Entrega

Como resumen del trabajo a realizar, mostramos una vista del contenido del paquete **org.expertojava.jbib.rest**, en donde se situarán las nuevas clases implementadas.

La vista de la carpeta **src/** es la siguiente:



La vista de la carpeta **test/** es la siguiente:



La fecha de entrega del proyecto será el jueves 28 de enero.

Debes dar premiso de lectura en el repositorio `jbib-rest-expertojava` al usuario **entregas-expertojava** y confirmar la entrega en Moodle.

5. (1,5 puntos) Despliegue en plataforma en la nube

En esta sesión vamos a utilizar los conocimientos que hemos adquirido durante las sesiones de PaaS para desplegar la parte del servidor en la nube, utilizando OpenShift.

Como resumen realizaremos las siguientes tareas:

- Revisión del proyecto pom.xml del proyecto.
- Revisión del entorno local de pruebas.
- Configuración y despliegue de la capa de servidor en OpenShift.
- Utilizar Integración Continua mediante Shippable a partir de nuestro repositorio en Bitbucket.
- Construir una imagen de Docker y publicarla en Docker Hub.

5.1. Revisión del proyecto pom.xml

La definición actual del proyecto pom.xml tiene algunos aspectos que debemos modificar, sobre todo pensando en las distintas fases por las que avanzará la aplicación antes de su publicación en OpenShift y la construcción de la imagen Docker. Estos cambio suponen.

- Revisar que el proyecto sólo incluya aquellas librerías que el servidor de aplicaciones no proporcione. Salvo causa justificada ya sabemos que es recomendable utilizar el conjunto de librerías del servidor pues entre otras cosas están validadas para trabajar en conjunto.
- Parametrizar la versión del servidor de aplicaciones y de las herramientas importantes como Arquillian. Recordad que en local estamos utilizando WildFly 8.2.1 pero en OpenShift se puede trabajar con WildFly 8, 9 o 10 y esto facilita el poder validar la aplicación en un entorno u otro.
- Perfiles. El pom.xml original no define ningún perfil. En el escenario en el que nos vamos a mover definiremos tres perfiles distintos:
  # *arquillian-wildfly-remote* este perfil es el que hemos utilizado hasta ahora implícitamente para la compilación y pruebas. Para su correcto funcionamiento requiere una instancia en ejecución del servidor de aplicaciones.
  # *arquillian-wildfly-managed* este perfil permite iniciar automáticamente una instancia de WildFly siempre y cuando tengamos definido la variable de entorno *JBOSS_HOME* apuntando a la carpeta de WildFly */usr/local/wildfly/*. Este perfil nos permitiría compilar y lanzar las pruebas unitarias pero nos impide el ejecutar las pruebas integradas ya que éstas están definidas para ejecutarse contra la aplicación desplegada en un servidor de aplicaciones.
  # *openshift* como ya sabemos es el perfil que utilizará OpenShift para compilar el proyecto: *mvn clean package -Popenshift -DskipTests*

El objetivo final es definir unos procesos de compilación y pruebas que nos sirvan tanto para trabajar en local como para utilizar integración y despliegue continuos.

El proyecto quedaría estructurado de la siguiente forma:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://
maven.apache.org/xsd/maven-4.0.0.xsd">
```

```
    <modelVersion>4.0.0</modelVersion>
    <groupId>org.expertojava</groupId>
    <artifactId>jbib-rest</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>war</packaging>
    <name>jbib-rest</name>
    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <version.wildfly>8.2.1.Final</version.wildfly>
        <version.arquillian>1.1.10.Final</version.arquillian>
        <version.arquillian-persistence>1.0.0.Alpha7</version.arquillian-
persistence>
    </properties>
    <dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>org.jboss.arquillian</groupId>
                <artifactId>arquillian-bom</artifactId>
                <version>${version.arquillian}</version>
                <scope>import</scope>
                <type>pom</type>
            </dependency>
        </dependencies>
    </dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>javax</groupId>
            <artifactId>javaee-web-api</artifactId>
            <version>7.0</version>
            <scope>provided</scope>
        </dependency>

        <!-- rest -->

        <dependency>
            <groupId>com.fasterxml.jackson.core</groupId>
            <artifactId>jackson-annotations</artifactId>
            <version>2.6.4</version>
            <scope>provided</scope>
        </dependency>

        <!-- Logging -->

        <dependency>
            <groupId>commons-logging</groupId>
            <artifactId>commons-logging</artifactId>
            <version>1.2</version>
            <scope>provided</scope>
        </dependency>

        <!-- Tests -->

        <dependency>
            <groupId>log4j</groupId>
            <artifactId>log4j</artifactId>
            <version>1.2.17</version>
            <scope>test</scope>
        </dependency>
```

```
<dependency>
 <groupId>org.slf4j</groupId>
 <artifactId>slf4j-simple</artifactId>
 <version>1.7.13</version>
           <scope>test</scope>
</dependency>

        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.11</version>
            <scope>test</scope>
        </dependency>

        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-api</artifactId>
            <version>1.7.7</version>
            <scope>test</scope>
        </dependency>

        <!-- JPA -->

        <dependency>
            <groupId>org.dbunit</groupId>
            <artifactId>dbunit</artifactId>
            <version>2.5.0</version>
            <scope>test</scope>
        </dependency>

        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <version>5.1.33</version>
            <scope>test</scope>
        </dependency>

        <dependency>
            <groupId>org.hibernate.javax.persistence</groupId>
            <artifactId>hibernate-jpa-2.1-api</artifactId>
            <version>1.0.0.Final</version>
            <scope>test</scope>
        </dependency>

        <dependency>
            <groupId>org.hibernate</groupId>
            <artifactId>hibernate-entitymanager</artifactId>
            <version>4.3.7.Final</version>
            <scope>test</scope>
        </dependency>

        <!-- Hibernate validator -->

        <dependency>
            <groupId>org.hibernate</groupId>
            <artifactId>hibernate-validator</artifactId>
            <version>5.1.3.Final</version>
```

```
        <scope>test</scope>
    </dependency>

    <dependency>
        <groupId>javax.el</groupId>
        <artifactId>javax.el-api</artifactId>
        <version>2.2.4</version>
        <scope>test</scope>
    </dependency>

    <dependency>
        <groupId>org.glassfish.web</groupId>
        <artifactId>el-impl</artifactId>
        <version>2.2</version>
        <scope>test</scope>
    </dependency>

    <!-- Arquillian -->

    <dependency>
        <groupId>org.jboss.arquillian.junit</groupId>
        <artifactId>arquillian-junit-container</artifactId>
        <scope>test</scope>
    </dependency>

    <!-- Arquillian persistence -->

    <dependency>
        <groupId>org.jboss.arquillian.extension</groupId>
        <artifactId>arquillian-persistence-api</artifactId>
        <version>${version.arquillian-persistence}</version>
        <scope>test</scope>
    </dependency>

    <dependency>
        <groupId>org.jboss.arquillian.extension</groupId>
        <artifactId>arquillian-persistence-core</artifactId>
        <version>${version.arquillian-persistence}</version>
        <scope>test</scope>
    </dependency>

    <dependency>
        <groupId>org.jboss.arquillian.extension</groupId>
        <artifactId>arquillian-persistence-dbunit</artifactId>
        <version>${version.arquillian-persistence}</version>
        <scope>test</scope>
    </dependency>

    <dependency>
        <groupId>org.jboss.arquillian.extension</groupId>
        <artifactId>arquillian-persistence-spi</artifactId>
        <version>${version.arquillian-persistence}</version>
        <scope>test</scope>
    </dependency>

    <!--Hamcrest Json -->
    <dependency>
        <groupId>uk.co.datumedge</groupId>
```

```
            <artifactId>hamcrest-json</artifactId>
            <version>0.2</version>
            <scope>test</scope>
        </dependency>

    </dependencies>

    <profiles>
        <profile>
            <id>arquillian-wildfly-remote</id>
            <dependencies>
                <dependency>
                    <groupId>org.wildfly</groupId>
                    <artifactId>wildfly-arquillian-container-remote</
artifactId>
                    <version>${version.wildfly}</version>
                    <scope>test</scope>
                </dependency>

         <!--Jaxrs Api cliente -->

         <dependency>
             <groupId>org.jboss.resteasy</groupId>
             <artifactId>resteasy-client</artifactId>
             <version>3.0.13.Final</version>
             <scope>test</scope>
         </dependency>

         <!--Librerías para serializar/deserializar json -->
         <dependency>
             <groupId>org.jboss.resteasy</groupId>
             <artifactId>resteasy-jackson-provider</artifactId>
             <version>3.0.13.Final</version>
             <scope>test</scope>
         </dependency>

         <!--Jaxrs API json -->
         <dependency>
             <groupId>org.jboss.resteasy</groupId>
             <artifactId>resteasy-json-p-provider</artifactId>
             <version>3.0.13.Final</version>
             <scope>test</scope>
         </dependency>

            </dependencies>

            <build>
                <finalName>${project.name}</finalName>
                <plugins>
                    <plugin>
                        <groupId>org.apache.maven.plugins</groupId>
                        <artifactId>maven-compiler-plugin</artifactId>
                        <version>3.1</version>
                        <configuration>
                            <source>1.7</source>
                            <target>1.7</target>
                        </configuration>
                    </plugin>
```

```
      <plugin>
                        <groupId>org.apache.maven.plugins</groupId>
                        <artifactId>maven-war-plugin</artifactId>
                        <version>2.3</version>
                        <configuration>
                            <failOnMissingWebXml>false</
failOnMissingWebXml>
                        </configuration>
                    </plugin>
  <plugin>
                <groupId>org.wildfly.plugins</groupId>
                <artifactId>wildfly-maven-plugin</artifactId>
                <version>1.0.2.Final</version>
                <configuration>
                    <hostname>localhost</hostname>
                    <port>9990</port>
                </configuration>
                <!-- forzamos el despliegue después de empaquetar
                     y antes de ejecutar los tests REST -->
                <executions>
                    <execution>
                        <id>wildfly-deploy</id>
                        <phase>pre-integration-test</phase>
                        <goals>
                            <goal>deploy</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
                    <plugin>
                        <groupId>org.apache.maven.plugins</groupId>
                        <artifactId>maven-surefire-plugin</artifactId>
                        <version>2.19</version>
                        <configuration>
                            <redirectTestOutputToFile>true</
redirectTestOutputToFile>
                            <reportsDirectory>shippable/testresults</
reportsDirectory>
       <skipTests>${skipTests}</skipTests>
                        </configuration>
                        <dependencies>
                            <dependency>
                                <groupId>org.apache.maven.surefire</
groupId>
                                <artifactId>surefire-junit4</artifactId>
                                <version>2.7.2</version>
                            </dependency>
                        </dependencies>
                    </plugin>
  <plugin>
                 <groupId>org.codehaus.mojo</groupId>
                        <artifactId>cobertura-maven-plugin</artifactId>
                        <version>2.7</version>
                        <configuration>
                            <format>xml</format>
                            <aggregate>true</aggregate>
                            <outputDirectory>shippable/codecoverage</
outputDirectory>
```

```
                        </configuration>
                    </plugin>
        <!-- ejecución de la goal failsafe:test
                     durante la fase integration-test -->
                <plugin>
                    <groupId>org.apache.maven.plugins</groupId>
                    <artifactId>maven-failsafe-plugin</artifactId>
                    <version>2.19</version>
                    <configuration>
                        <skipITs>${skipITs}</skipITs>
                        <skipTests>false</skipTests>
                    </configuration>
                    <executions>
                        <execution>
                            <goals>
                                <goal>integration-test</goal>
                                <goal>verify</goal>
                            </goals>
                        </execution>
                    </executions>
                </plugin>
                   </plugins>
                </build>
            </profile>

            <profile>
                <id>arquillian-wildfly-managed</id>
                <dependencies>
                    <dependency>
                        <groupId>org.wildfly</groupId>
                        <artifactId>wildfly-arquillian-container-managed</
artifactId>
                        <version>${version.wildfly}</version>
                        <scope>test</scope>
                    </dependency>

                </dependencies>
                <build>
                    <finalName>${project.name}</finalName>
                    <plugins>
                        <plugin>
                            <groupId>org.apache.maven.plugins</groupId>
                            <artifactId>maven-compiler-plugin</artifactId>
                            <version>3.1</version>
                            <configuration>
                                <source>1.7</source>
                                <target>1.7</target>
                            </configuration>
                        </plugin>
                        <plugin>
                            <groupId>org.codehaus.mojo</groupId>
                            <artifactId>cobertura-maven-plugin</artifactId>
                            <version>2.6</version>
                            <configuration>
                                <format>xml</format>
                                <maxmem>256m</maxmem>
                                <aggregate>true</aggregate>
```

```
                            <outputDirectory>shippable/codecoverage</
outputDirectory>
                        </configuration>
                    </plugin>
                    <plugin>
                        <groupId>org.apache.maven.plugins</groupId>
                        <artifactId>maven-surefire-plugin</artifactId>
                        <version>2.19</version>
                        <configuration>
                            <redirectTestOutputToFile>true</
redirectTestOutputToFile>
                            <reportsDirectory>shippable/testresults</
reportsDirectory>
                        </configuration>

                        <dependencies>
                            <dependency>
                                <groupId>org.apache.maven.surefire</
groupId>
                                <artifactId>surefire-junit4</artifactId>
                                <version>2.7.2</version>
                            </dependency>
                        </dependencies>
                    </plugin>
                    <plugin>
                        <groupId>org.apache.maven.plugins</groupId>
                        <artifactId>maven-war-plugin</artifactId>
                        <version>2.3</version>
                        <configuration>
                            <failOnMissingWebXml>false</
failOnMissingWebXml>
                        </configuration>
                    </plugin>
                </plugins>
            </build>
        </profile>
    </profiles>
</project>
```

Pruebas integradas REST

Estas pruebas se situan exclusivamente en el contenedor remoto ya que requieren de un despliegue sobre una instancia en ejecución.

Tareas

- Reemplazar el pom.xml antiguo por el propuesto.
- Añadir el perfil openshift para que se pueda desplegar correctamente en PaaS.
- Definir el perfil openshift como perfil por defecto. Esto nos permite que por defecto no lancemos las pruebas integradas y nos permite compilar de forma rápida tanto en local como en OpenShift (seguiremos teniendo que especificar que no se ejecuten los test mediante el parámetro -DskipTests o bien configurarlo en el propio profile).
- Definir un MANIFEST.MF que referencie a los módulos *org.apache.commons.logging y org.apache.log4j* que forman parte del paquete de librerías de WildFly y no es necesario incluirlos en el WAR.

Con estos cambios la aplicación contendrá únicamente las clases del proyecto ocupando 74kb aproximadamente gracias a las librerías del propio servidor. Además, el pom admitirá una serie de parámetros como el perfil o la versión de WildFly con la que queremos trabajar:

mvn [goal]… -P[perfil] -Dversion.wildfly[versión]

5.2. Revisión del entorno local de pruebas

Shippable es una herramienta de integración continua que facilita una serie de servicios para realizar las pruebas y permite utilizar imágenes de Docker personalizadas como máquina de pruebas si lo que nos ofrece por defecto no se ajusta a nuestras necesidades.

La imagen de pruebas incorpora una instancia de mysql pero nos propociona un usuario que no dispone de todos los permisos. Concretamente podemos crear bases de datos nuevas pero no definir nuevos usuarios, con lo que nos obliga a utilizar un usuario "shippable" en las pruebas que queramos realizar. Para no complicarnos demasiado lo que vamos a hacer es adaptar nuestro proyecto al entorno de pruebas que propiorciona Shippable dando permisos en local al usuario shippable:

```
grant all on biblioteca to "shippable@%" identified by "";
```

El '%' nos permite que podamos utilizar este usuario tanto en conexiones locales como remotas. Alternativamente podéis utilizar la herramienta Mysql Workbench y de forma visual crear el usuario shippable y asignarle todos los permisos sobre el esquema biblioteca.

Tareas

- Exportar el contenido actual de la base de datos biblioteca a un fichero SQL mediante el comando:

```
mysqldump -u root -p biblioteca > fichero.sql
```

El *fichero.sql* lo necesitaremos para inicializar la base de datos de producción (OpenShift).

- Dar permisos al usuario shippable (sin contraseña) sobre nuestra base de datos local.
- Modificar los casos de prueba.

  # Modificar la configuración de acceso a base de datos de los Test para que utilicen el usuario shippable que acabamos de crear:

```
.getConnection(
 "jdbc:mysql://localhost/biblioteca",
    "shippable", "");

....

    this.databaseTester = new
 JdbcDatabaseTester("com.mysql.jdbc.Driver",
    "jdbc:mysql://localhost:3306/biblioteca", "shippable", "")
```

Crear un Datasource de nombre *biblioteca-ds.xml* en */src/test/resources*:

```
<?xml version="1.0" encoding="UTF-8"?>

<datasources xmlns="http://www.jboss.org/ironjacamar/schema"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://www.jboss.org/ironjacamar/
schema http://docs.jboss.org/ironjacamar/schema/datasources_1_0.xsd">

    <xa-datasource jndi-name="java:/datasources/BibliotecaDS" pool-
name="MySQLTest" enabled="true" use-ccm="true">
        <xa-datasource-property name="URL">
            jdbc:mysql://localhost/biblioteca
        </xa-datasource-property>
        <xa-datasource-
class>com.mysql.jdbc.jdbc2.optional.MysqlXADataSource</xa-datasource-
class>
        <driver>mysql-connector-
java-5.1.33.jar_com.mysql.jdbc.Driver_5_1</driver>
        <security>
            <user-name>shippable</user-name>
        </security>
    </xa-datasource>
</datasources>
```

5.3. Configuración y despliegue de la capa de servidor en OpenShift

En esta parte crearemos una aplicación escalable con soporte de BD MySQL y añadiremos el código fuente de la capa REST.

Tareas

- Crear una aplicación escalable, de nombre *jbibrest* con los cartridges de WildFly 8 y MySQL. (Si por lentitud os tardase demasiado podéis configurarla como no escalable, es una buena práctica pero no es requerido por el ejercicio).
- Eliminar el código fuente de ejemplo (pom.xml y carpeta src).

```
git rm -r src/ pom.xml
git commit -m "borrado plantilla"
```

- Inicializar manualmente el contenido de la BD MySQL de OpenShift a partir del fichero de backup que creamos en el primer apartado.
- Añadir un repositorio Git remoto (el de bitbucket):

```
git remote add bitbucket https://[usuario]@bitbucket.org/[usuario]/
[repositorio].git
```

- Hacer un "pull" del repositorio de Bitbucket para traernos los fuentes de nuestro proyecto:

```
git pull bitbucket master
```

- Definir en el fichero *standalone.xml* de la configuración, el datasource XA *BibliotecaDS*. Fijaos que OpenShift por defecto crea una BD de WildFly con el mismo nombre que la aplicación (jbibrest) por tanto utilizad esta BD en el datasource. Recordad que debéis trabajar con las variables de entorno en lugar de con valores fijos.
- Hacer un push al repositorio de Bitbucket para almacenar el proyecto actualizado con los ficheros de configuración de OpenShift.

```
git add .
git commit -m "integración con OpenShift"
git push bitbucket master
```

- Hacer un push al repositorio de OpenShift y asegurarnos de que la aplicación está funcionando correctamente en http://jbibrest-[usuario].rhcloud.com.

```
git push origin master
```

- Por último, dar de alta dos usuarios en el ApplicationRealm de WildFly:

Usuario	Password	Rol
vicente.casamayor	vicente	usuario
anabel.garcia	anabel	usuario

5.4. Utilizar Integración Continua mediante Shippable a partir de nuestro repositorio en Bitbucket.

Dada la lentitud de los gear gratuitos de OpenShift es preferible utilizar un proveedor de CI externo, como pueda ser Shippable. Shippable tiene varias ventajas sobre otras opciones.

- Nos proporciona una instancia de CI de forma gratuita.
- Integración con OpenShift, GitHub y BitBucket.
- Uso muy sencillo, mediante un script YAML.
- Uso de imágenes de Docker como base para realizar las pruebas. Se configura el entorno de pruebas desde cero en cada prueba con los elementos que necesitamos.
- Admite tareas de integración continua PreCI y PostCI a partir de otras imágenes de Docker. Podemos construir una imagen previamente y utilizarla para ejecutar las pruebas o bien construir una imagen con los binarios resultado de la compilación y pruebas.
- Acceso simple mediante Oauth utilizando vuestras credenciales de Bitbucket.

La idea es que ante cualquier cambio que subáis a vuestro repositorio de BitBucket, se active una instancia de Shippable, recoja estos cambios, haga las pruebas que hayamos definido y si todo es correcto, lo despliegue en el gear de OpenShift.

Tareas

- Crear un script de integración continua donde se detallan los pasos para configurar el entorno y realizar las pruebas. El fichero se llamará *shippable.yml* y se situará al mismo nivel que el fichero *pom.xml*.

El contenido del fichero será el siguiente:

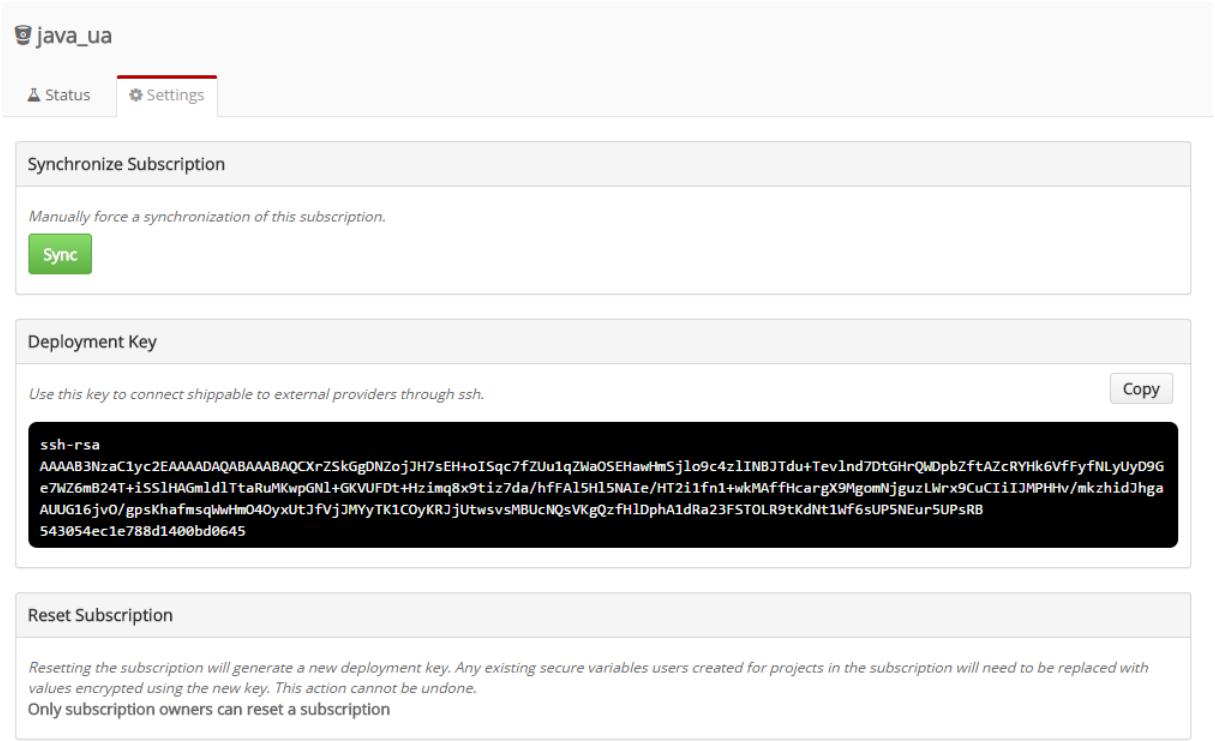
```
language: java

jdk:
   - openjdk7

env:
  global:
    - JBOSS_HOME=/tmp/wildfly-8.2.1.Final
    - JBOSS_SERVER_LOCATION=http://download.jboss.org/wildfly/8.2.1.Final/
wildfly-8.2.1.Final.tar.gz
    - OPENSHIFT_REPO=ssh://567f9c8f0c1e6651f4000122@jbibrest-
jlzamora.rhcloud.com/~/git/jbibrest.git
    - MYSQL_DRIVER=http://central.maven.org/maven2/mysql/mysql-connector-
java/5.1.33/mysql-connector-java-5.1.33.jar
    - MYSQL_DRIVER_DEST=/tmp/wildfly-8.2.1.Final/standalone/deployments/
mysql-connector-java-5.1.33.jar

before_install:
  - if [ ! -e $JBOSS_HOME ]; then curl -s $JBOSS_SERVER_LOCATION | tar zx
 -C /tmp; fi
  - curl -s $MYSQL_DRIVER >$MYSQL_DRIVER_DEST
  - cp src/test/resources/biblioteca-ds.xml $JBOSS_HOME/standalone/
deployments
  - ls /tmp/wildfly-8.2.1.Final/standalone/deployments/
  - mysql -e "create database biblioteca;"
  - sh $JBOSS_HOME/bin/add-user.sh --silent=true -a vicente.casamayor
 vicente
  - sh $JBOSS_HOME/bin/add-user.sh  --silent=true -a anabel.garcia anabel
  - echo "vicente.casamayor=usuario">> $JBOSS_HOME/standalone/
configuration/application-roles.properties
  - echo "anabel.garcia=usuario">> $JBOSS_HOME/standalone/configuration/
application-roles.properties
  - sh $JBOSS_HOME/bin/standalone.sh &
  - git remote -v | grep ^openshift || git remote add openshift
 $OPENSHIFT_REPO

before_script:
  - mkdir -p shippable/testresults
  - mkdir -p shippable/codecoverage
  - mkdir -p shippable/buildoutput

script:
  - cp deployments/ROOT.war  ./shippable/buildoutput/jbib-rest.war
  - mvn clean verify -Parquillian-wildfly-remote -
Dwildfly.version=8.2.1.Final
  - mvn cobertura:cobertura -Dtest=modelo/* -Parquillian-wildfly-remote

after_success:
```

```
- git push -f openshift $BRANCH:master
```

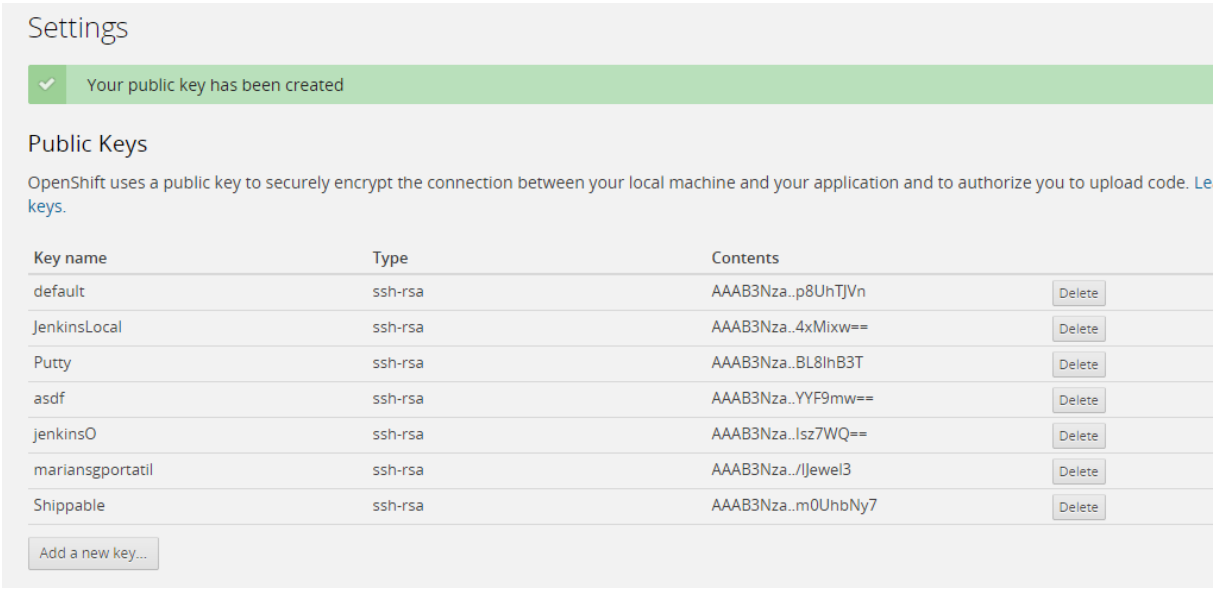
Este script descarga y descomprime la misma versión de WildFly que se utilizará en OpenShift, prepara el entorno para lanzar las pruebas y si las pruebas concluyen con éxito hace un *push* del código al repositorio Git de OpenShift.

Debeis modificar el script con la ruta SSH a vuestro gear concreto

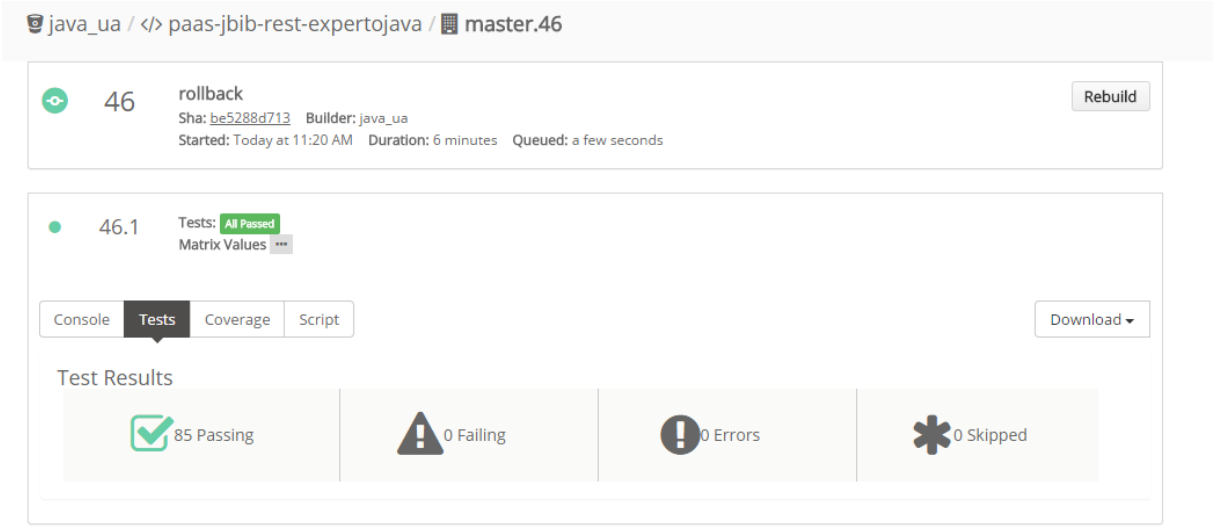
- Acceder a http://www.shippable.com autenticándonos con el usuario/password de Bitbucket. Esto nos permitirá listar nuestros proyectos.
- En el panel de la derecha saldrá nuestro usuario de BitBucket y podremos seleccionar un repositorio para habilitar integración continua. Seleccionar el repositorio del proyecto de la capa REST.
- Obtener la "Developer Key" de nuestra sesión en Shippable. Esta clave se debe registrar en OpenShift para que quede constancia de que damos permisos a Shippable para subir código a los repositorios Git gestionados por OpenShift. Dentro de Shippable, pulsaremos en el icono de la rueda, situado al lado de nuestro usuario y se mostrará la key:

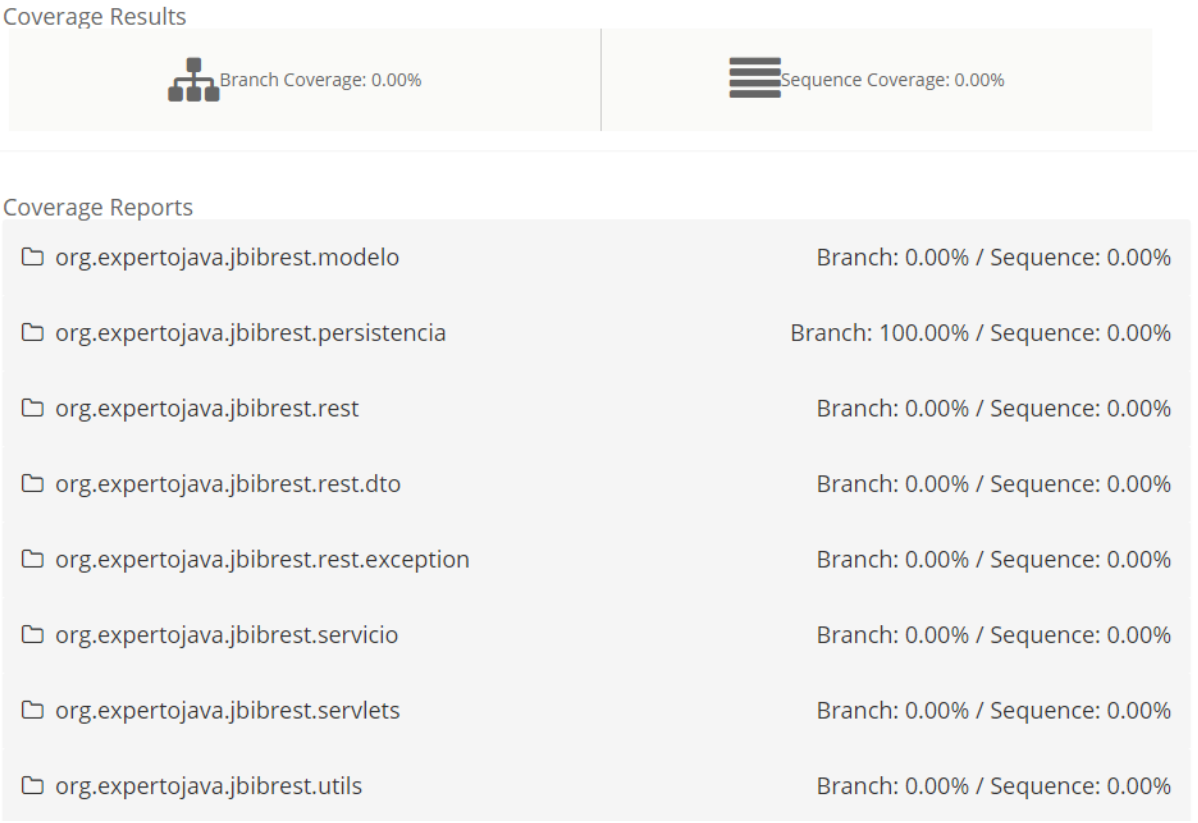


- En Settings de OpenShift, daremos de alta esta clave como *public key*. El nombre asociado puede ser el que queráis, por ejemplo *Shippable*



Hacer un *push* únicamente al repositorio de Bitbucket. Si lo hemos hecho todo bien, se iniciará una tarea de compilación en Shippable y podremos seguir paso a paso todo el proceso. Si falla algo, revisad la ejecución de los pasos en busca del error concreto que se haya producido.





Si todo es correcto, para trabajar en futuras sesiones de integración bastaría con clonarnos el repositorio de Bitbucket a local y gracias a Shippable podremos desplegar nuestros cambios en OpenShift sin necesidad de tener nada configurado, únicamente subiendo los cambios a Bitbucket.

5.5. Construir una imagen de Docker y publicarla en Docker Hub

Si las pruebas han ido bien y la aplicación se ha desplegado en OpenShift, queremos añadir un paso mas que nos permita generar una imagen de Docker con nuestro backend Rest que podamos utilizar para nuestras pruebas de Fixing, en conjunto con una base de datos local.

Técnicamente es posible y es mejor solución el definir también la base de datos en otro contenedor y desplegarlos de forma conjunta pero nos limitaremos a los conocimientos de Docker que hemos adquirido en la primera sesión.

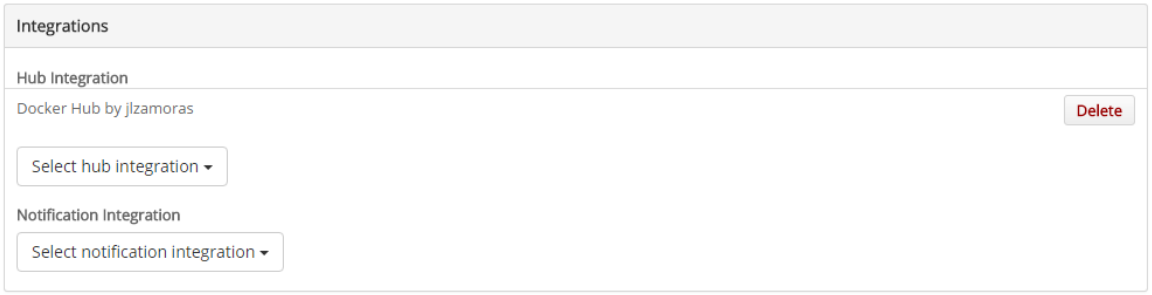
Tareas

- Configurar la base de datos local Mysql para que admita conexiones remotas. Como vamos a trabajar con contenedores, estos sólamente se pueden comunicar con el resto de servicios mediante comunicación TCP/IP. Para ello modificaremos el fichero my.cnf:

```
/etc/mysql/my.cnf # cambiar el parámetro bind adress por de 127.0.0.1 a
 0.0.0.0
# a continuación reiniciar la BD

$ sudo /etc/init.d/mysql restart
```

- El los settings del proyecto jbib-rest dentro de Shippable, dar de alta una nueva integración. Las integraciones son enlaces con repositorios de imágenes y entre ellos seleccionaremos

"Docker" (Docker Hub). Será necesario indicar nuestro usuario, password y datos de contacto:

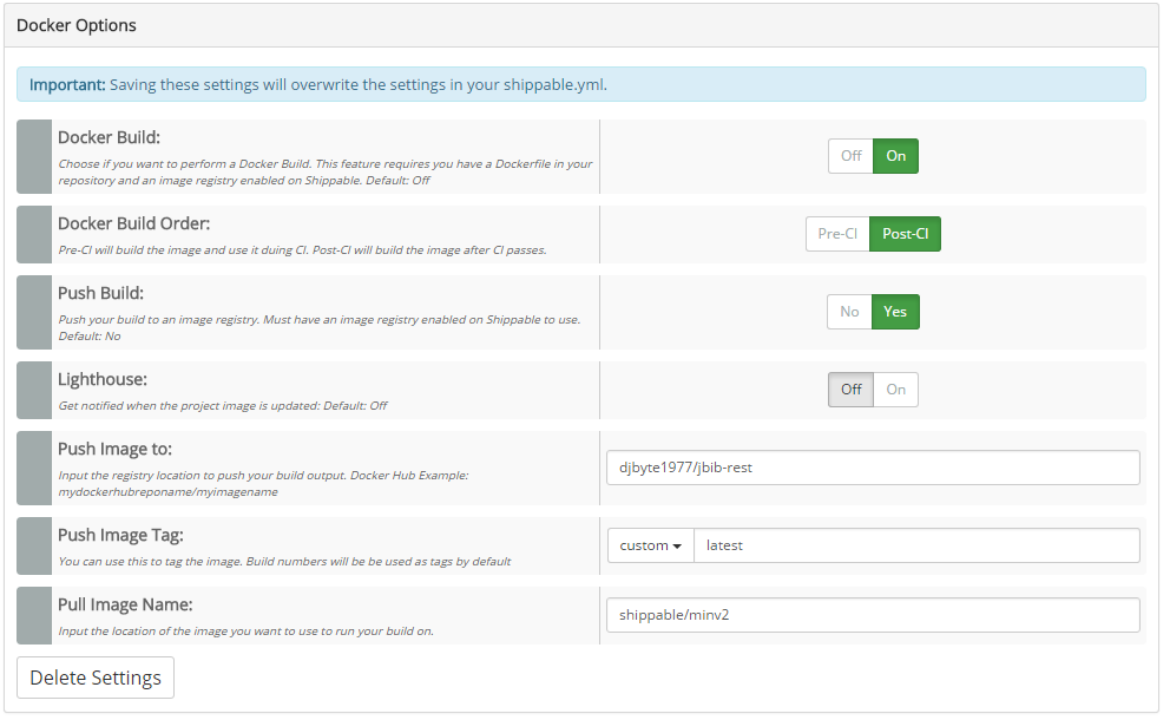


- Crear un catasource de nombre biblioteca-docker-ds.xml y a partir del que ya tenéis (biblioteca-ds) y añadirlo al repositorio. Dentro de este datasouce debéis eliminar la referencia a la maquina "localhost" dentro de la cadena de conexión de base de datos y en su lugar apuntar a una máquina llamada "docker".
- Añadir un fichero Dockerfile al proyecto: Tenéis que basaros en el que vimos en la primera sesión incorporando las siguientes instrucciones:

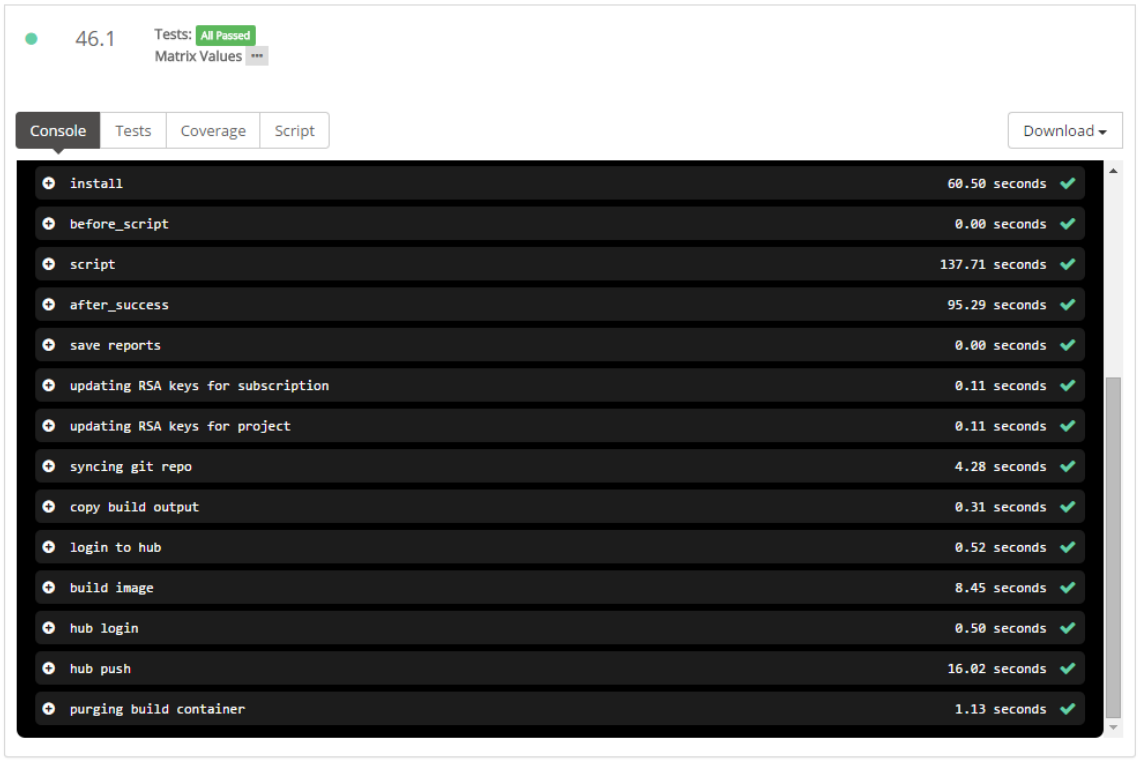
```
RUN /opt/jboss/wildfly/bin/add-user.sh --silent -a vicente.casamayor
 vicente
RUN /opt/jboss/wildfly/bin/add-user.sh --silent -a anabel.garcia anabel
RUN curl http://central.maven.org/maven2/mysql/mysql-connector-
java/5.1.33/mysql-connector-java-5.1.33.jar > /opt/jboss/wildfly/
standalone/deployments/mysql-connector-java-5.1.33.jar
COPY src/test/resources/biblioteca-docker-ds.xml /opt/jboss/wildfly/
standalone/deployments/
COPY buildoutput/jbib-rest.war /opt/jboss/wildfly/standalone/
deployments/
```

En el fichero shippable.yml especificábamos una carpeta donde dejar los binarios compilados y en el Dockerfile recuperaremos el WAR y lo integraremos en la nueva imagen.

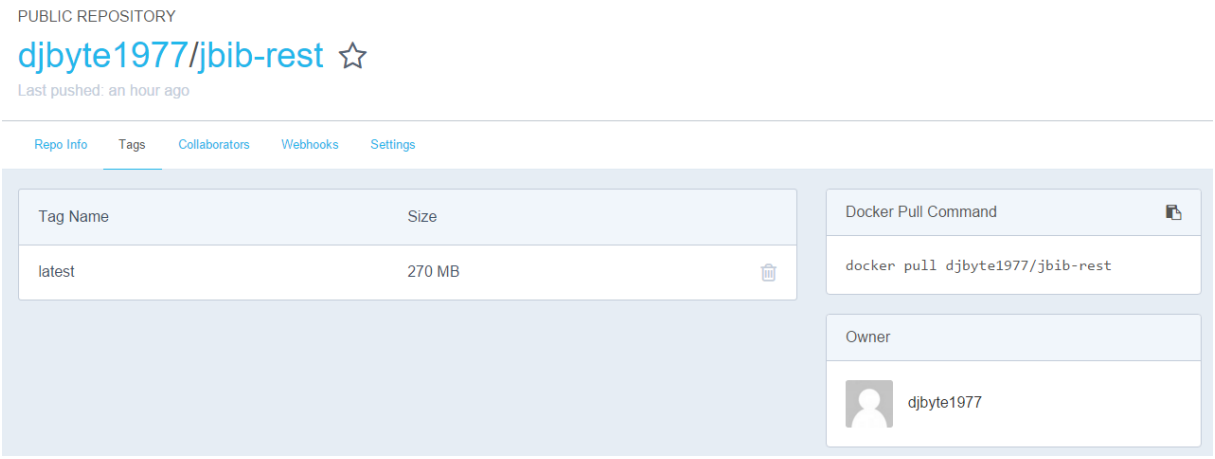
- Habilitar la opción de generar una imagen de docker una vez se haya completado la tarea de integración continua. Esto lo podéis hacer en la opción settings de Shippable, dentro del proyecto . Configurar los siguientes parámetros susituyendo el nombre de usuario por el vuestro:



- Ejecutar de nuevo la build. Si todo funciona bien se completarán los siguientes pasos.



En vuestro repositorio de Docker Hub tiene que parecer una nueva imagen con la etiqueta "latest", que es la que hemos especificado en las propiedades del proyecto.



Para poder probar la imagen en local bastaría con ejecutar:

```
$ ifconfig docker0  #Sólo 1 vez, para averiguar la DIRECCION_IP del Host
 de Docker
$ docker run -it --add-host=docker:DIRECCION_IP   -p 8080:8080 -p
 9990:9990 djbyte1977/jbib-rest
```

Si habéis completado todos los pasos, el contenedor se podrá conectar a la BD local y la aplicación funcionará correctamente.

5.6. Entrega

La fecha de entrega del proyecto será el jueves 4 de febrero.

Añade la etiqueta `sesion-4` al respositorio después de hacer el último commit y sube los cambios al repositorio `jbibrest-proyint-expertojava` en Bitbucket:

```
$ git tag -a sesion-4 -m "Sesión 4"
$ git push origin --tags
```

Recuerda dar premiso de lectura/escritura al usuario **entregas-expertojava** y confirmar la entrega en Moodle. Una de las pruebas será modificar el código fuente y comprobar que los cambios se despliegan correctamente en OpenShift.

Adicionalmente revisaremos que la aplicación esté desplegada correctamente en OpenShift accediendo a la URL convenida:

http://jbibrest-[usuario].rhcloud.com.

Y también probaremos vuestra imagen publicada en Docker Hub. Debéis indicarnos vuestro usuario, bien en la entrega o bien en un fichero *Readme.md* o *leeme.txt* del proyecto.

Para verificar que la parte de integración continua funciona correctamente os pediremos un fichero de log de Shippable. Este se puede obtener a partir de la opción Script, que aparece en la build junto con el resto de información resultante de la ejecución.

5.7. Referencias

Shippable & OpenShift http://docs.shippable.com/en/latest/continuous_deployment.html#continuous-deployment-to-red-hat-openshift <<<

6. Cliente AngularJS

En esta sesión vamos a hacer uso de los servicios REST implementados y desplegados en OpenShift, y vamos a crear una interfaz realizada en HTML5 + Bootstrap + AngularJS para consumir dichos servicios a través de una aplicación web SPA.

Dado que estamos trabajando con clientes ricos y aplicaciones desacopladas, nuestra aplicación web se desarrollará en un proyecto independiente del proyecto java, y también se ejecutará en otro servidor.

Las ventajas de esta arquitectura son múltiples. Al tratarse nuestra aplicación web de un consumidor de servicios, sería muy fácil crear una aplicación móvil con Apache Cordova[37], o una aplicación de escritorio con Electron[38]. Simplemente habría que introducir el código del ejercicio en la carpeta *webapps* de cada plataforma.

Mediante el uso de Bootstrap[39], estamos garantizando un diseño responsive que se adapta de manera adecuada a todo tipo de dispositivos. Aunque la solución idónea para dispositivos móviles pasa por Ionic Framework[40]: un framework basado en AngularJS y con ui-router como gestor de rutas que se ha convertido en el framework de referencia a la hora de desarrollar aplicaciones móviles híbridas.

Aunque la aplicación es *responsive*, la maquetación llevada a cabo no se adapta del todo bien a dispositivos móviles.

Pese a que muchas aplicaciones se puedan adaptar correctamente, en mi opinión los dispositivos móviles suelen utilizar otros patrones y el desarrollo de la interfaz no debería ser adaptable, sino específico para éstos. Es por ello que considere a ionic framework algo imprescindible para las interfaces de aplicaciones móviles, pudiendo reaprovechar lógica de negocio existente en servicios, filtros y algunas directivas.

6.1. Fork del repositorio

Empezaremos realizando un *fork* del repositorio https://bitbucket.org/java_ua/jbib-angular-expertojava. Éste tiene una base de código sobre la que empezaremos.

6.2. Instalación de las dependencias del proyecto

Fijémonos en los ficheros `.bowerrc` y `package.json`. En éstos se definen las dependencias de nuestro proyecto, de igual manera que maven lo hace con los paquetes java que podamos necesitar. El hecho de usar un gestor de dependencias hace que el tamaño de nuestro repositorio sea mucho más liviano, que podamos ver las versiones de nuestras dependencias de una manera sencilla (el fichero indica la versión), y que todos los desarrolladores puedan trabajar con las mismas versiones de las librerías de las que depende un proyecto.

Para instalarnos las dependencias, ejecutaremos, desde la línea de comandos y poniéndonos en la raíz de nuestro proyecto:

[37] https://cordova.apache.org/
[38] http://electron.atom.io/
[39] http://getbootstrap.com/
[40] http://ionicframework.com/

```
$ npm install
```

Con esto habremos instalado las librerías que necesitamos para automatizar tareas. Ahora, para instalar las dependencias de nuestra aplicación web, ejecutaremos el comando:

```
$ bower install
```

Ya tenemos instaladas todas las dependencias. El gestor también nos habrá instalado un pequeño servidor web, que utilizaremos para probar nuestra aplicación. Podemos lanzarlo mediante el comando:

```
$ node node_modules/webserver/webserver.js
```

Si todo va bien, deberíamos obtener la salida:

```
WEBROOT.
Listening @ 8003
```

Así, ya tenemos un servidor corriendo y escuchando en el puerto 8003, y otro servidor en OpenShift con nuestra API. También podría ser un Wildfly corriendo de manera local en nuestra máquina, pero con la primera opción tenemos más presente el desacoplamiento. El ejercicio se corregirá probándolo contra vuestra API de OpenShift y toda la configuración deberá dejarse preparada de esta manera.

6.3. CORS en clientes web

Para que nuestra aplicación pueda realizar peticiones AJAX contra un servidor remoto, deberemos crear un filtro para que la aplicación soporte CORS (Cross-Origin Resource Sharing) que nos permitirá saltarnos el *sandbox* de las peticiones AJAX, que en un principio sólo permiten realizarse dentro del mismo dominio. También, vamos a tener que realizar alguna modificación más sobre nuestro código Java.

Esto se debe a que muchas peticiones requieren una comunicación extra entre el cliente y el servidor antes de realizar una petición `GET`, `POST`, `PUT` o `DELETE`. A esto se le conoce como *preflight request*, y es una llamada al servidor, sin *payload*, y que se envía con una cabecera `OPTIONS`.

Es por ello que debemos preparar nuestro código Java para que acepte este tipo de llamadas. Esto podríamos resolverlo de dos maneras:

- Creando una función para cada llamada con la anotación `@OPTIONS` y devuelva un código de respuesta `200 OK`.
- Creando un filtro que devuelva una respuesta `200 OK` en cada petición con cabecera `OPTIONS` directamente desde el filtro, sin llegar a pasar por la lógica de negocio.

Para el ejercicio, se ha optado por la primera opción, al considerarla más restrictiva.

Un ejemplo concreto, para la clase LibroResource, sería el siguiente:

```
@OPTIONS
@Consumes({"application/json"})
@Produces({"application/json"})
@Path("{id}/prestamos")
public Response realizarPrestamoOptions(@PathParam("id") Long id) {
    return Response.ok().build();
}
```

Antes de realizar un préstamo, se enviará una petición sin *payload* al servidor y con la cabecera `OPTIONS`. Si la respuesta es correcta, será entonces cuando se envíe la petición `POST`.

Para aliviar la tarea de implementar este nuevo código, se adjunta el filtro CORS, así versiones modificadas de las clases de recursos con estas anotaciones añadidas:

- CORSFilter.java[41]
- UsuarioResource.java[42]
- LibroResource.java[43]

Si tu solución difiere de la que se ha adjuntado, deberás adaptar el código para que funcione correctamente.

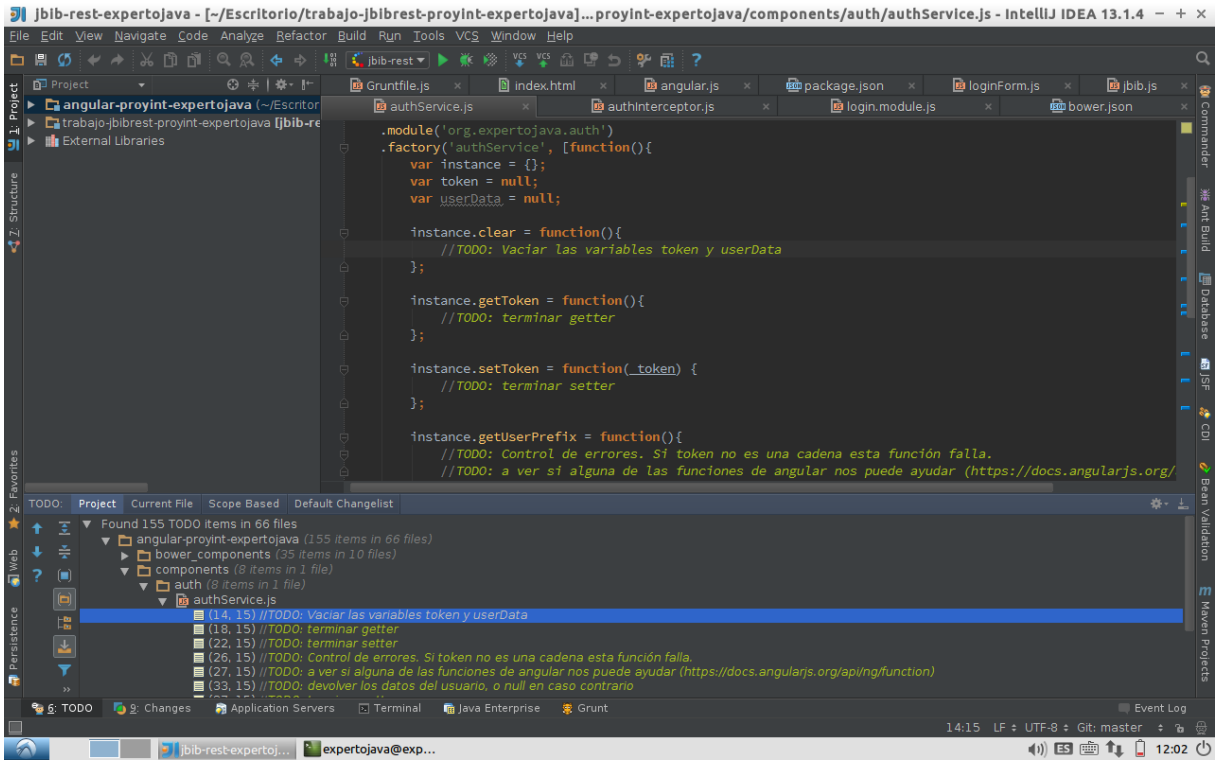
6.4. Aplicación web

Como hemos dicho, la aplicación web estará desarrollada en AngularJS, y hará uso de los servicios REST implementados.

El código de la aplicación (javascript y HTML) está incompleto, y tendremos que terminarlo para que ésta funcione.

Se ha puesto un `TODO` allá donde haga falta realizar algo. IntelliJ Idea tiene una vista de TO-DOs, donde podemos verlos todos juntos para que no se nos pase ninguno:

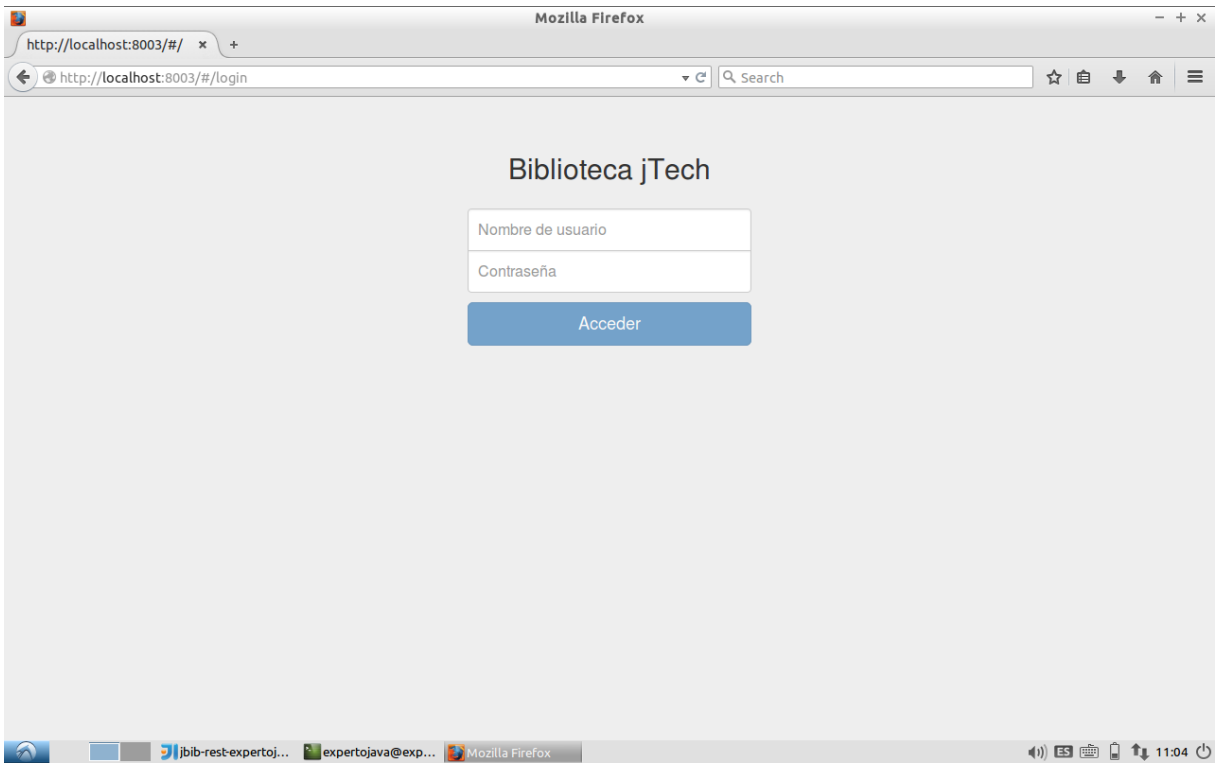
41 resources/CORSFilter.java
42 resources/UsuarioResource.java
43 resources/LibroResource.java



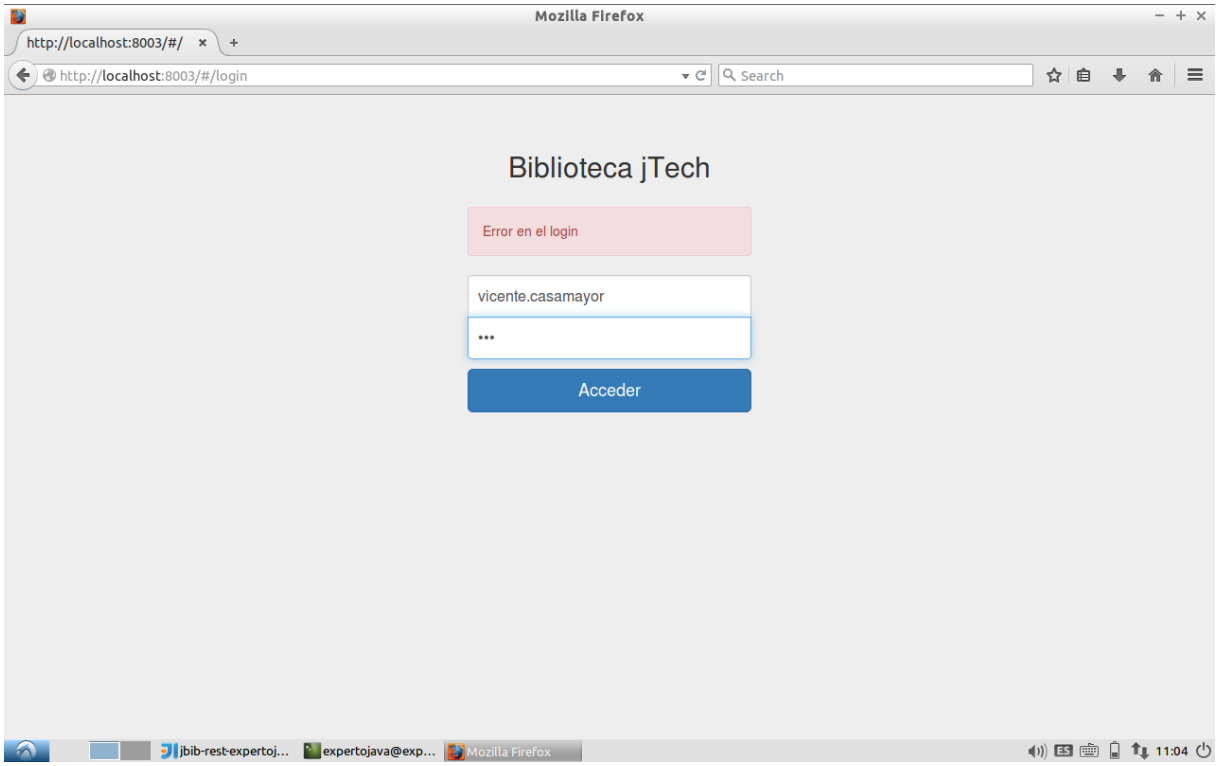
La funcionalidad se corresponderá con las acciones asocidadas a un usuario, y pasamos a ver en los siguientes pantallazos:

Login

Será el punto de entrada a la aplicación, y desde la que el usuario introducirá sus credenciales de acceso:



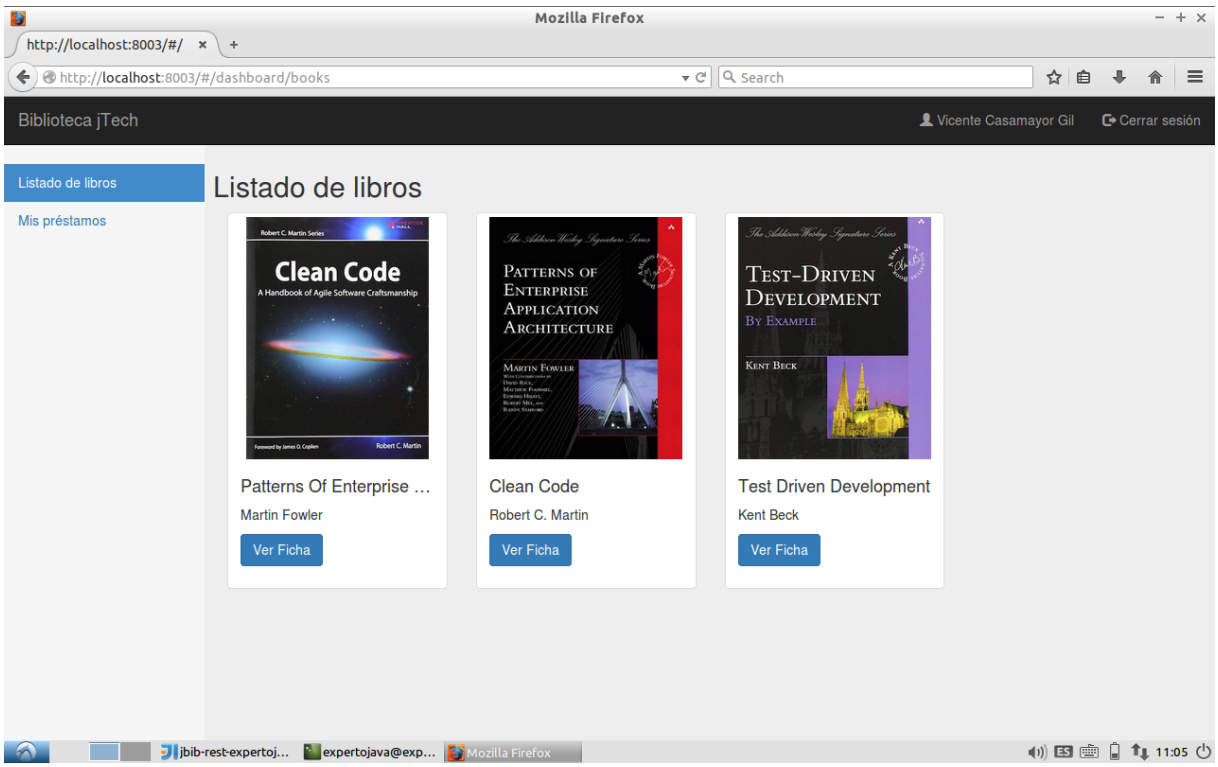
Si las credenciales introducidas son incorrectas, se indicará al usuario mediante un mensaje:



El sistema de autenticación

Listado de libros

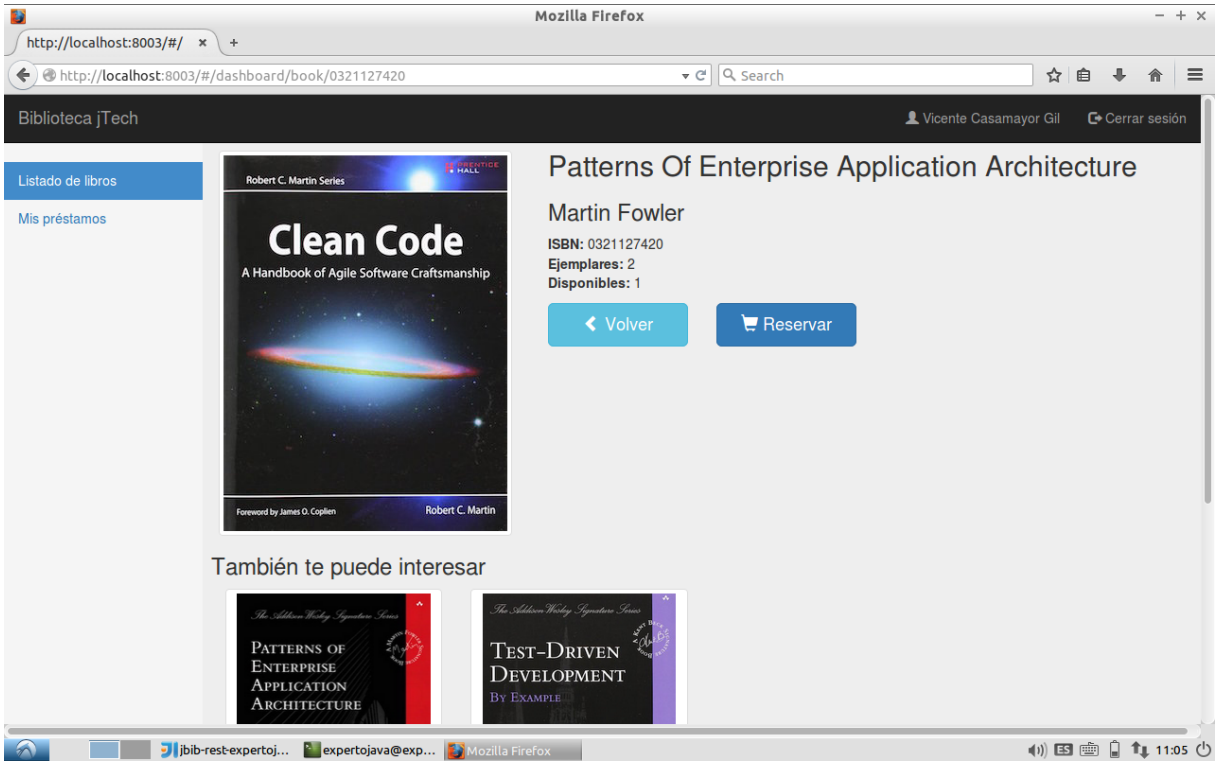
Inmediatamente después de acceder a la aplicación, veremos el listado de libros:



Cada libro es una directiva que tendremos que terminar de implementar. Además, como el serivcio REST nos devuelve la miniatura de la imagen, vamos a implementar un filtro que modifique la cadena para que devuelva la imagen a tamaño completo. No queremos modificar el servicio, sino lo que éste nos devuelve y por eso lo vamos a hacer desde el lado del cliente.

Detalle de un libro

Muestra el detalle de un libro: título, autor, isbn, número de ejemplares total y número de ejemplares disponibles. Además, habrá un botón *Volver* que hará un `history.back()` del navegador y un botón *Reservar*, que invocará al servicio de reservas.



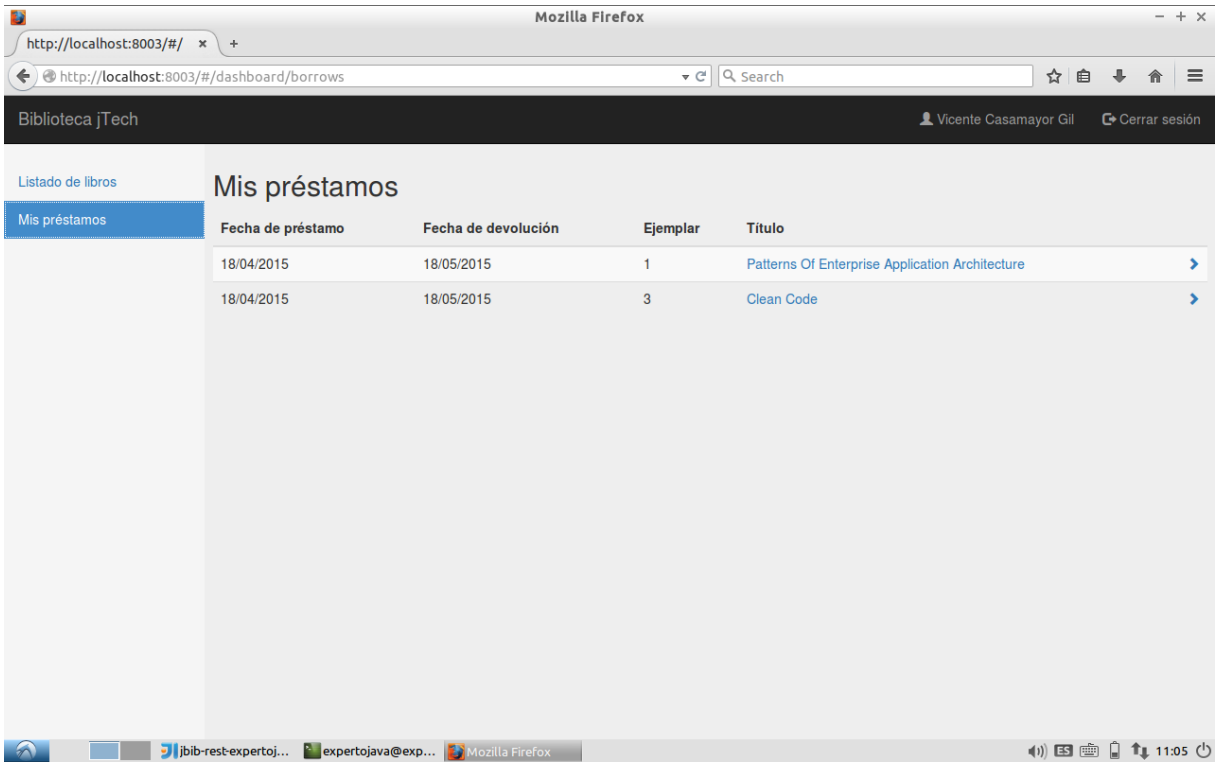
Si se puede realizar la reserva de manera correcta, se mostrará una alerta en verde. Si por el contrario no se puede realizar la reserva, se mostrará un mensaje en rojo y el motivo (por ejemplo: usuario no válido porque tiene una multa). El fichero UsuarioResource ya captura excepciones y manda un mensaje de error junto con una cabecera 403 (`Forbidden`) en caso de no poder realizar una reserva.

El detalle del libro es una directiva. Ésta muestra en rojo el número de ejemplares disponibles cuando es cero. Además, el botón de reservas deberá quedar deshabilitado si el número de ejemplares disponibles es cero. Si se está haciendo una reserva, el botón de reservar también se deshabilitará para evitar el doble click, y mostrará un *spinner*. La clase CSS de la rueda girando está preparada y se llama `glyphicon-cog`.

A continuación del detalle del libro tenemos la lista de libros relacionados. Deberá ser un máximo de 8, y reaprovecharemos la misma directiva empleada en el listado principal de libros.

Listado de préstamos

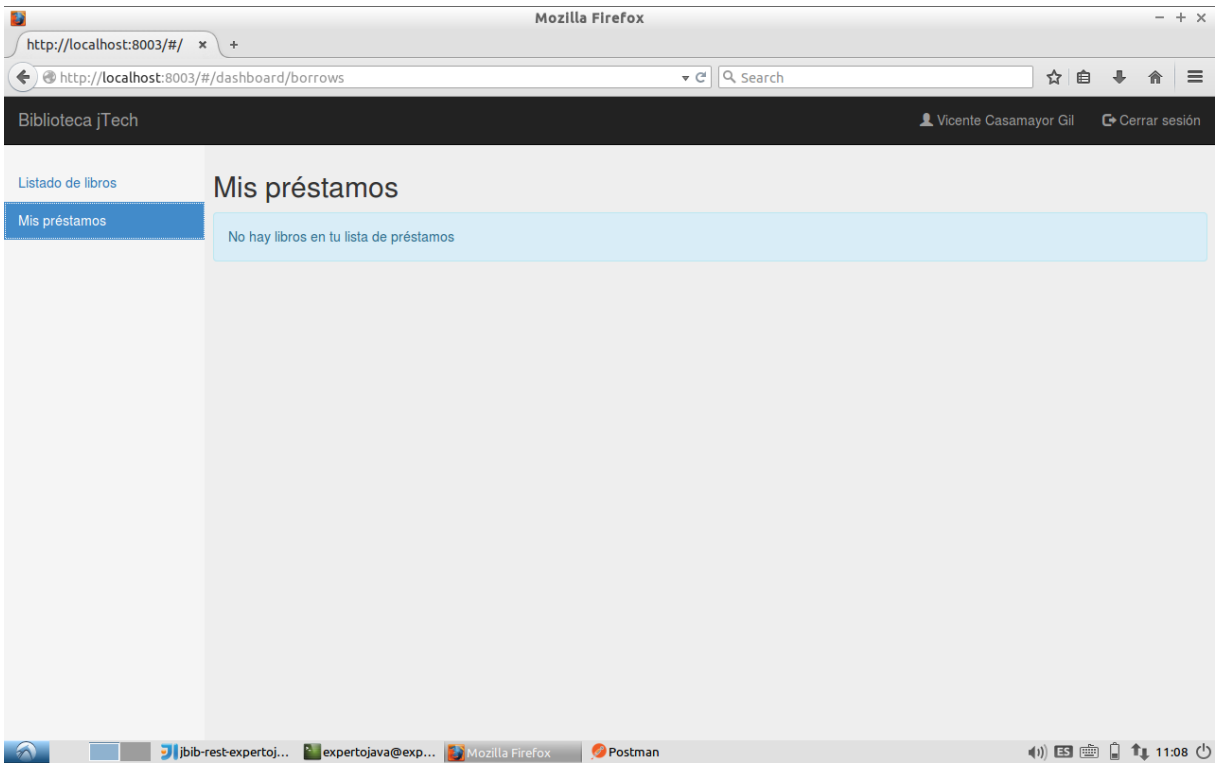
Mostrará todos los préstamos del usuario:



Aunque están marcados en azul sólo los títulos de los libros, haciendo click en cualquier lugar de la fila seremos llevados al detalle del libro.

Si la fecha de devolución de un préstamo ha vencido, el texto tendrá la clase `text-alert` para mostrarse en rojo.

En caso de no tener ningún préstamo, en lugar de mostrar la tabla, se enseñará un mensaje informativo:



Habrá que intentar que la vista no haga *flickering*, mostrando una cosa un milisegundo y luego mostrando otra.

Devoluciones.

La funcionalidad de devolución queda fuera del alcance este proyecto. En caso de necesitar hacer una devolución lo haremos a través de la aplicación *Postman*, o cualquier otro cliente REST. También podríamos acceder directamente a la base de datos y lanzar la query.

6.5. Control de acceso

Se ha implementado un sencillo control de acceso en la aplicación. Lo podemos ver en el fichero `jbib.js`, en el método `run` de la aplicación. Básicamente consiste en que si no estamos en los `states` de nombre `login` o `logout` se mirará si el usuario está logado. Si no lo está, redirigirá a la pantalla de login.

6.6. Comunicación con el servidor

Toda la comunicación con el servidor se hará con el módulo `ngResource`[44], que hemos visto que facilita la integración con servicios REST.

Deberemos terminar la implementación de los siguientes servicios, que son los que favorecerán dicha comunicación:

- booksResource.js
- borrowService.js
- relatedBooksResource.js

Observad en el código que `borrowsService` se ha concebido de manera ligeramente distinta: mientras los otros dos devuelven el propio recurso, éste implementa una factoría como las que hemos visto en clase.

6.7. Estructura del proyecto

Se ha tenido en cuenta algunas de las *best practices* a la hora de elaborar la estructura del proyecto. Cada funcionalidad está en una carpeta distinta (préstamos, libros, acceso, dashboard). Dentro de cada una de ellas, habrá una carpeta para servicios, directivas, controladores, filtros y vistas.

Las funcionalidades de login y gestión de acceso se han creado como módulos independientes, con la idea de poder ser reutilizados en otros proyectos.

```
.
├── books
│   ├── controllers
│   │   ├── bookController.js
│   │   └── booksController.js
│   ├── directives
│   │   ├── book
│   │   │   ├── book.html
│   │   │   └── book.js
│   │   └── book-detail
│   │       ├── book-detail.html
```

[44] https://docs.angularjs.org/api/ngResource

```
│   │       └── bookDetail.js
│   ├── filters
│   │   └── largeImg.js
│   ├── services
│   │   ├── booksResource.js
│   │   ├── borrowService.js
│   │   └── relatedBooksResource.js
│   └── views
│       ├── book.html
│       └── books.html
├── borrows
│   ├── controllers
│   │   └── borrowsController.js
│   └── views
│       └── borrows.html
├── bower_components
├── bower.json
├── components
│   ├── auth
│   │   ├── authInterceptor.js
│   │   ├── auth.module.js
│   │   └── authService.js
│   └── images
│       └── nophoto.png
├── dashboard
│   ├── directives
│   │   ├── navbar
│   │   │   ├── navbar.html
│   │   │   └── navbar.js
│   │   └── sidebar
│   │       ├── sidebar.html
│   │       └── sidebar.js
│   └── views
│       └── dashboard.html
├── dist
├── Gruntfile.js
├── index.html
├── jbib.css
├── jbib.js
├── login
│   ├── directives
│   │   ├── login-form.html
│   │   └── loginForm.js
│   ├── login.module.js
│   └── views
│       └── login.html
├── node_modules
├── package.json
├── tree.txt
└── tsd.json
```

6.8. Automatización

Se han creado unas reglas de *linting*[45] y minificación. En las etapas iniciales del desarrollo será complicado de usar, pero a medida que vayamos avanzando será necesario. La prueba de la aplicación se hará sobre el código minificado, y éste no se minificará si no pasa el *linting*.

Lanzado el siguiente comando desde la raíz del proyecto:

```
$ grunt
```

Se ejecutará un *listener* que intentará pasar las reglas de JSHint y minificar el código cada vez que realizamos una modificación en el mismo. Así, sólo tendremos que refrescar el navegador cuando estas tareas se hayan realizado.

Si sólo queremos minificar el código sin que se ejecute ningún *listener*, haremos uso del comando:

```
$ grunt dist
```

La carpeta `dist` está en el .gitignore y no deberá subirse al repositorio. El código de distribución se generará en el momento de la corrección.

6.9. Corrección

Aplica el tag `FINAL` al *commit* que quieras que se te corrija.

[45] http://jshint.com/