

# Proyecto Java EE

## Índice

1 Caso de estudio.....	3
1.1 Introducción a Maven.....	3
1.2 Caso de estudio.....	24
1.3 Implementación.....	35
1.4 Resumen.....	53
2 Capa de datos y negocios con JPA.....	55
2.1 Creación de entidades de dominio con JPA.....	55
2.2 Capa de lógica de negocio.....	74
2.3 Resumen.....	85
3 Capa web con servlets y JSPs.....	88
3.1 Introducción.....	88
3.2 Proyecto jbib-web.....	88
3.3 Configuración de la fuente de datos.....	89
3.4 Configuración de la seguridad.....	91
3.5 Componentes web.....	93
3.6 Resumen.....	98
4 Servicios RESTful con Jersey.....	100
4.1 Introducción.....	100
4.2 Proyecto jbib-rest.....	100
4.3 Configuración del acceso a datos.....	101
4.4 Configuración de Jersey.....	103
4.5 Configuración de la seguridad.....	103
4.6 Recursos REST.....	104
4.7 Resumen.....	109
5 Sprint Web.....	111
5.1 Introducción.....	111

5.2 Funcionalidades a Desarrollar.....	111
5.3 A Entregar.....	113
6 Componentes de presentación.....	115
6.1 Introducción.....	115
6.2 Login del bibliotecario.....	115
6.3 Página principal - Listado de libros.....	116
6.4 Menú del usuario.....	117
6.5 Borrado de libros.....	118
6.6 Creación de libros.....	120
6.7 Edición de libros.....	121
6.8 Validaciones.....	122
6.9 Control de acceso.....	123
6.10 Ayuda: Esqueleto de proyecto.....	123
7 Integración componentes Java EE: EJB y JMS.....	125
7.1 GlassFish y Componentes EJB.....	125
7.2 JMS y MDBs.....	129
8 Integración con Servicios Web.....	132
8.1 Introducción.....	132
8.2 Creación de servicios web.....	132
8.3 Acceso a servicios web externos.....	135
8.4 Cliente para el servicio Web.....	138
8.5 Resumen.....	138
9 Sprint Enterprise.....	140
9.1 Introducción.....	140
9.2 Funcionalidades a Desarrollar.....	140
9.3 A Entregar.....	141

## 1. Caso de estudio

### 1.1. Introducción a Maven

Maven es una herramienta Java de gestión del proceso de desarrollo 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 la versión más utilizada en la actualidad de Maven: 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 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, las interfaces de usuario incluyen sus distintos comandos en menús de opciones para que puedan seleccionarse de forma gráfica. 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). De esta forma es posible publicar el proyecto en un *repositorio* y ponerlo a disposición de la comunidad para que otros a su vez puedan usarlo como librería.

#### 1.1.1. Instalando 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: <http://maven.apache.org/download.html> y descomprimirla en algún directorio del sistema. En el caso de la MV, lo hemos instalado en /opt/apache-maven-3.0.3.

Maven es una aplicación Java, y utiliza la variable `JAVA_HOME` para encontrar el path del JDK. También es necesario añadir el directorio `bin` de Maven al `PATH` del sistema. Se pueden definir en el fichero de configuración `.bashrc` de un usuario o en el fichero del sistema `/etc/bashrc` para todos los usuarios. En nuestro caso hemos modificado el único usuario de la MV especialista. El código que hemos añadido ha sido este:

```
export JAVA_HOME=/opt/jdk1.6.0_27
export MAVEN_HOME=/opt/apache-maven-3.0.3
export PATH=${PATH}: ${MAVEN_HOME}/bin
```

### 1.1.2. Dependencias de librerías en proyectos Java

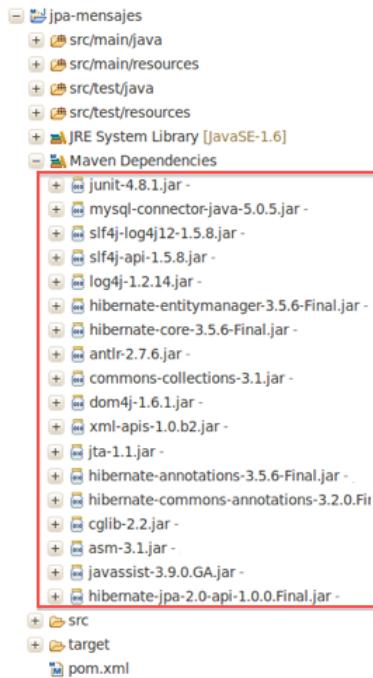
Una característica del desarrollo de proyectos Java es la gran cantidad de librerías (ficheros JAR) necesarios para compilar y ejecutar un proyecto. Todas las librerías que se importan deben estar físicamente tanto en la máquina en la que se compila el proyecto como en la que posteriormente se ejecuta.

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 en el fichero POM 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 librerías cuando ejecutamos el comando `mvn install`. La siguiente imagen muestra las librerías descargadas en un proyecto Eclipse en el que se utiliza Maven:



### 1.1.3. 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 librerías 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, librerías, ficheros de configuración y ficheros de distintos formatos que maneja el servidor web (HTML, JPG, etc.)

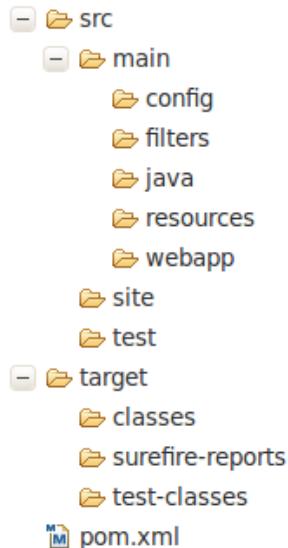
- **Fichero EAR:** aplicación enterprise que se despliega en un servidor de aplicaciones. Contiene librerías, componentes EJB y distintas aplicaciones web (ficheros WAR).

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, subversión o Git), lanzar tests o desplegar la aplicación en algún servidor de aplicaciones.

Podría pensarse que los entornos de desarrollo (Eclipse o Netbeans) 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.

#### 1.1.4. Estructura de un proyecto Maven

La estructura de directorios de un proyecto Maven genérico es la que aparece en la siguiente figura.



Si observamos el primer nivel, encontramos en la raíz del proyecto el fichero `pom.xml`, el directorio `src` y el directorio `target`. El fichero `pom.xml` es el fichero principal de Maven en el que se describen todas las características del proyecto: nombre, versión, tipo de artefacto generado, librerías de las que depende, etc. Lo estudiaremos más adelante. El directorio `src` contiene todo el código fuente original del proyecto. Y el directorio

target contiene el resultado de la construcción del proyecto con Maven.

Entrando en más profundidad, nos encontramos los siguientes directorios:

- src/main/java: el código fuente de las clases Java del proyecto
- src/main/resources: ficheros de recursos que necesita la aplicación
- src/main/filters: filtros de recursos, en forma de ficheros de propiedades, que pueden usarse para definir variables que se utilizan en tiempo de ejecución
- src/main/config: ficheros de configuración
- src/main/webapp: el directorio de aplicación web de un proyecto WAR
- src/test/java: código fuente de las pruebas de unidad de las clases que tiene la misma estructura que la estructura del código fuente en el directorio main/java
- src/tests/resources: recursos utilizados para los tests que no serán desplegados
- src/tests/filters: filtros de recursos utilizados para los tests
- src/site: ficheros usados para generar el sitio web Maven del proyecto

Los ficheros de recursos son ficheros leídos desde la aplicación. Tal y como hemos visto en el módulo JHD, estos recursos deben estar contenidos en la raíz del JAR para poder acceder a ellos utilizando el método `getResourceAsStream` de la clase `Class`:

```
InputStream in =  
    getClass().getResourceAsStream("/datos.txt");
```

Maven se encarga de colocar los recursos en la raíz del JAR al empaquetar el proyecto.

### 1.1.5. 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 por el grupo de desarrollo del proyecto. Viendo algunos de ellos podemos entender también más características de Maven.

Vamos a utilizar como ejemplo el primer proyecto de integración jbib-modelo que construiremos más adelante. Veamos su fichero `pom.xml`. Al comienzo nos encontramos jbib-modelos con la cabecera XML y la definición del proyecto:

```
<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/maven-v4_0_0.xsd">  
<modelVersion>4.0.0</modelVersion>  
<groupId>es.ua.jtech.proyint</groupId>  
<artifactId>jbib-modelo</artifactId>  
<packaging>jar</packaging>  
<version>0.0.1-SNAPSHOT</version>  
<name>jbib-modelo</name>  
<url>http://web.ua.es/expertojee</url>
```

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.

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:

```
es.ua.jtech.proyint:jbib-modelo:jar:0.0.1-SNAPSHOT
```

Por último, hay que definir el nombre lógico del proyecto (*name*) y una URL asociada al mismo *url*.

A continuación definimos algunas propiedades del proyecto, que se utilizarán en los distintos procesos de Maven. En nuestro caso 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>
```

A continuación viene la definición de las dependencias del proyecto: librerías de las que dependen el proyecto. En nuestro caso:

- JUnit: junit:junit:4.8.1:jar
- Log4j: log4j:log4j:1.2.14:jar
- Commons Logging: commons-logging:commons-logging:1.1.1:jar

```
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.8.1</version>
    <type>jar</type>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>log4j</groupId>
    <artifactId>log4j</artifactId>
    <version>1.2.14</version>
    <type>jar</type>
    <scope>compile</scope>
  </dependency>
  <dependency>
    <groupId>commons-logging</groupId>
    <artifactId>commons-logging</artifactId>
    <version>1.1.1</version>
    <type>jar</type>
    <scope>compile</scope>
  </dependency>
</dependencies>
```

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

En nuestro caso, definimos el nivel de compilación de Java, necesario para que no haya

conflicto al importar el proyecto en Eclipse. Es interesante hacer notar que el plugin se identifica de una forma similar al proyecto, utilizando el identificador del grupo, el de proyecto y su número de versión.

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

Es posible definir herencia entre los ficheros POM utilizando el identificador `parent`. Es útil para definir elementos comunes y evitar repetirlos en todos los POM. Por ejemplo, podríamos definir un POM en el que se declaren todas las librerías que se usan habitualmente por nuestros proyectos e incluir este POM en todos los proyectos utilizando herencia.

```
<project>
  <parent>
    <groupId></groupId>
    <artifactId></artifactId>
    <version></version>
  </parent>
  ...
</project>
```

Maven define un *super POM* que por defecto es el padre de todos los POM. Allí se definen elementos comunes como la localización de los repositorios o la estructura de directorios por defecto de Maven. Se puede encontrar este super POM en el fichero llamado `pom-4.0.0.xml` en el JAR `maven-2.2.1-uber.jar` en el directorio `lib` de Maven.

Maven resuelve todas las relaciones de herencia entre POMs y genera internamente un *POM efectivo (effective POM)* en el que combinan todos los POMs que afectan a un determinado proyecto. Este POM efectivo es el que se utiliza para realizar la construcción del proyecto. Es posible consultar este POM efectivo con el comando:

```
mvn help:effective-pom
```

También puede consultarse en la pestaña correspondiente del editor POM del plugin de Eclipse.

### 1.1.6. Repositorios

Los proyectos software están relacionados. Los proyectos necesitan 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 obliga a declarar explícitamente estas dependencias en el fichero POM del proyecto y se encarga de gestionar estos y otros problemas:

- 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`.
- **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
- **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
```

### 1.1.7. 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.

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".

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 *deploy* y se despliega el artefacto en el servidor remoto, Maven sustituye el palabra SNAPSHOT por la fecha actual.

### **1.1.8. 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 esta 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 repositorio 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.

### 1.1.9. 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`. 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.

**Nota:**

Para un listado completo de todas las opciones se puede consultar la página de Apache Maven [Introduction to the Build Lifecycle](#)

### 1.1.10. Ejecutando 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 es.ua.jtech.jbib.model.UsuarioTest
Tests run: 5, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.082 sec
Running es.ua.jtech.jbib.model.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 es.ua.jtech.jbib.model.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:

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

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

### 1.1.11. Plugins y goals

Todo el trabajo que realiza Maven es realizado por módulos independientes que son

también descargados del repositorio global. Estos módulos reciben el nombre de *plugins*. Cada plugin tiene un conjunto de *goals* que podemos lanzar desde línea de comando. La sintaxis de una ejecución de un *goal* de un *plugin* es:

```
$ mvn <plugin>:goal -Dparam1=valor1 -Dparam2=valor2 ...
```

Las fases del ciclo de vida vistas en el apartado anterior también se procesan mediante el mecanismo de plugins. Por ejemplo, la llamada a `mvn test` realmente genera una llamada al objetivo `test` del plugin `surefire`:

```
$ mvn surefire:test
```

Otro ejemplo es el plugin Jar que define el objetivo `jar` para realizar la fase `package`:

```
$ mvn jar:jar
```

Existen múltiples plugins de Maven que pueden utilizarse para automatizar distintas tareas complementarias necesarias para la construcción del proyecto. Un ejemplo es el [plugin SQL](#) de Codehaus que permite lanzar comandos SQL utilizando su objetivo `execute`:

```
$ mvn sql:execute
```

La configuración del plugin se define en el fichero POM. La siguiente configuración lanza en la fase de construcción `process-test-resource` dos ficheros de comandos SQL; el primero inicializa la base de datos y el segundo la llena con datos sobre los que se van a realizar las pruebas.

```
<plugin>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>sql-maven-plugin</artifactId>
  <version>1.4</version>

  <dependencies>
    <!-- specify the dependent JDBC driver here -->
    <dependency>
      <groupId>mysql</groupId>
      <artifactId>mysql-connector-java</artifactId>
      <version>5.0.8</version>
    </dependency>
  </dependencies>

  <!-- common configuration shared by all executions -->
  <configuration>
    <driver>com.mysql.jdbc.Driver</driver>
    <url>jdbc:mysql://localhost:3306/</url>
    <username>root</username>
    <password>expertojava</password>
  </configuration>

  <executions>
    <execution>
      <id>create-db</id>
      <phase>process-test-resources</phase>
      <goals>
        <goal>execute</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

```
<srcFiles>
    <srcFile>src/main/sql/biblioteca.sql</srcFile>
</srcFiles>
</configuration>
</execution>

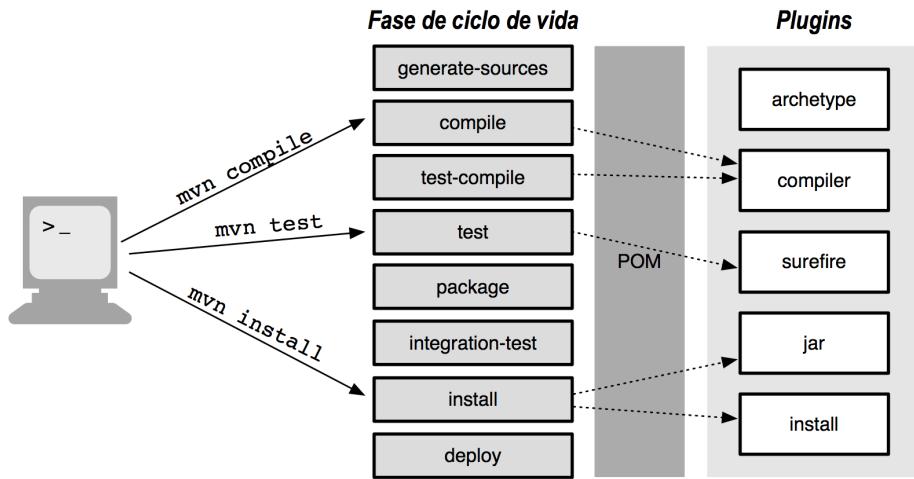
<execution>
    <id>create-data</id>
    <phase>process-test-resources</phase>
    <goals>
        <goal>execute</goal>
    </goals>
    <configuration>
        <srcFiles>
            <srcFile>src/test/sql/datos.sql</srcFile>
        </srcFiles>
    </configuration>
    </execution>
</executions>
</plugin>
```

Vemos que en el apartado dependency se definen las dependencias del plugin. En este caso el JAR mysql-connector-java-5.0.8 necesario para la conexión a la base de datos. En el apartado executions es donde se definen los ficheros SQL que se ejecutan y la fase del ciclo de vida en la que se lanza.

Cuando se llame a la fase process-test-resoures de Maven (o a cualquiera posterior) se ejecutaran los ficheros SQL:

```
$ mvn process-test-resources
[INFO] Scanning for projects...
...
[INFO] [sql:execute {execution: create-db}]
[INFO] Executing file: /tmp/biblioteca.2081627011sql
[INFO] 24 of 24 SQL statements executed successfully
[INFO] [sql:execute {execution: create-data}]
[INFO] Executing file: /tmp/datos.601247424sql
[INFO] 27 of 27 SQL statements executed successfully
[INFO]
-----
[INFO] BUILD SUCCESSFUL
[INFO]
-----
[INFO] Total time: 3 seconds
[INFO] Finished at: Wed Oct 06 06:16:02 CEST 2010
[INFO] Final Memory: 8M/19M
[INFO]
```

La siguiente figura muestra un resumen de los conceptos vistos hasta ahora de fases de build y plugins:

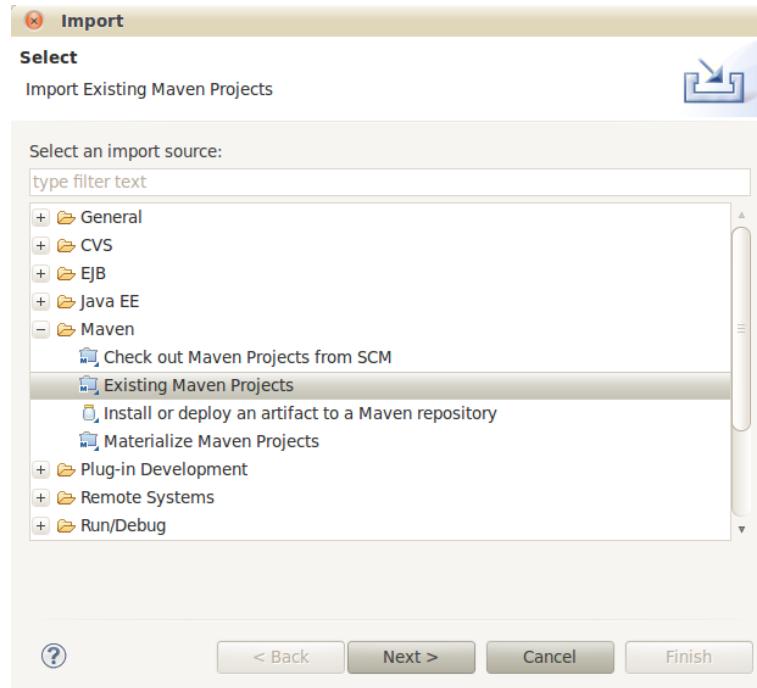


### 1.1.12. Usando Maven en Eclipse

El plugin de Eclipse [m2eclipse](#) permite la utilización de Maven desde el entorno de programación. Ha sido desarrollado por la compañía Sonatype y se distribuye de forma gratuita. El plugin permite importar proyectos Maven en Eclipse, editar y explorar el fichero POM del proyecto, explorar repositorios de Maven o lanzar builds de Maven desde el propio entorno.

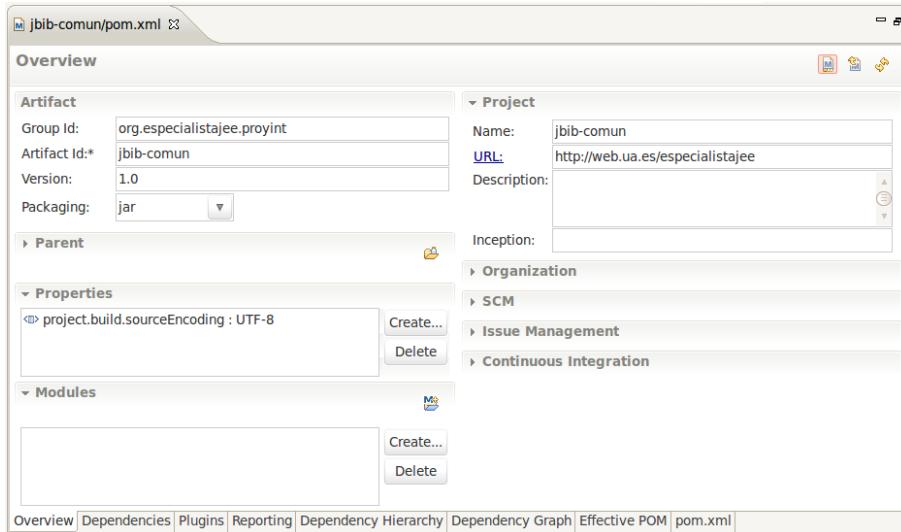
El documento [Sonatype m2eclipse](#) es un excelente manual de uso del plugin. Os remitimos a él para conocer en profundidad todas sus características. Aquí sólo haremos un rápido repaso.

El plugin permite importar un proyecto Maven en Eclipse y trabajar con él manteniendo su estructura de directorios. El plugin genera automáticamente los ficheros de configuración de Eclipse (los ficheros `.project` y `.classpath`) para que utilicen esa estructura de directorios y para que se actualicen las dependencias entre los proyectos y las librerías JAR.

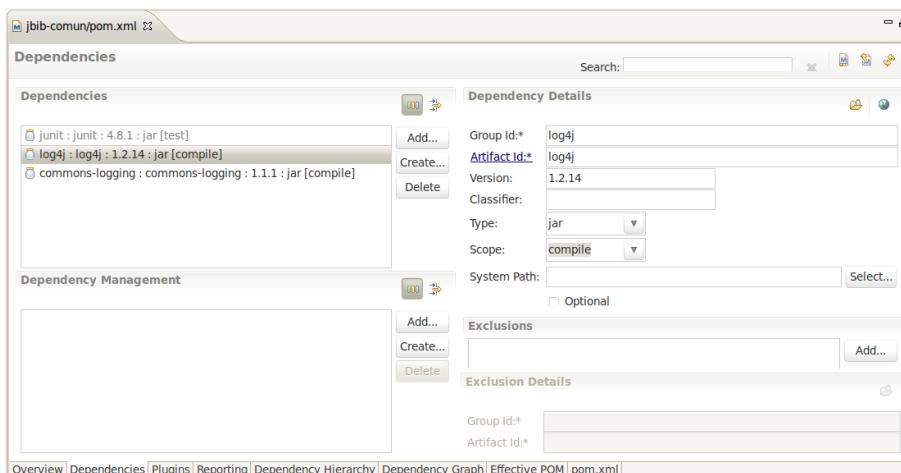


Una vez importado el proyecto el desarrollador puede seguir trabajando en él como lo haría con cualquier proyecto Eclipse. Puede lanzar añadir código, ejecutarlo, lanzar los tests o compartirlo por CVS. En el caso en que fuera un proyecto web, podrá también desplegarlo en el servidor instalado en Eclipse.

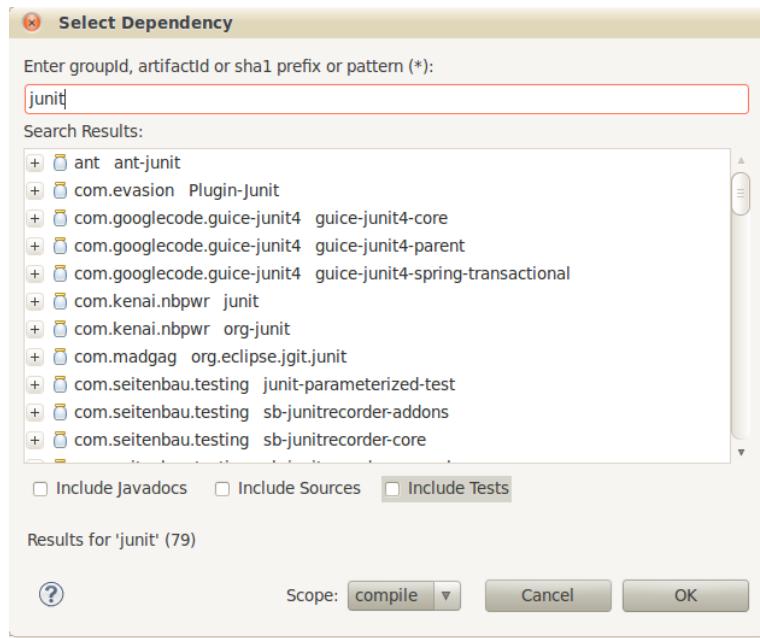
El plugin m2eclipse proporciona un editor especializado para trabajar con los ficheros POM. En la parte inferior de la imagen se pueden observar pestañas para acceder a los distintas elementos configurados en el POM: general, dependencias, plugins, informes, jerarquía de dependencias, grafo de dependencias, POM efectivo y código XML.



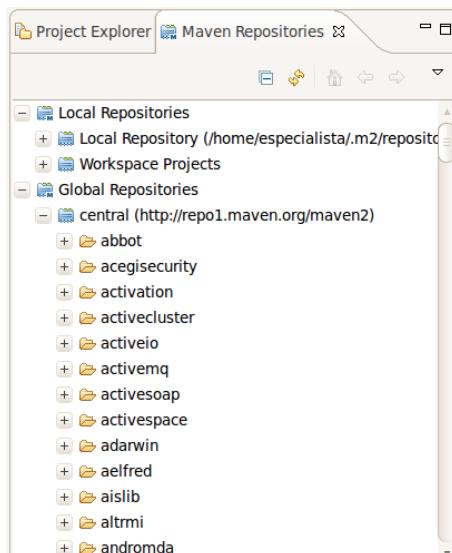
Por ejemplo, la pestaña de dependencias del POM muestra el siguiente aspecto.



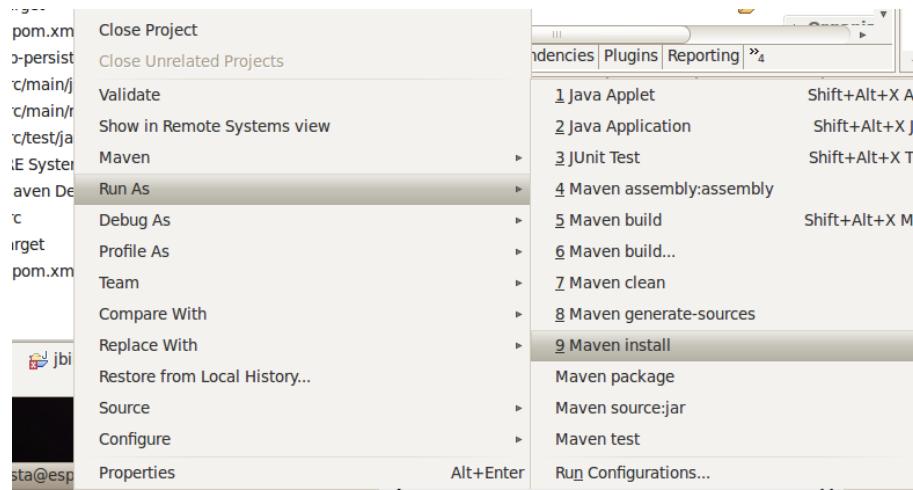
Desde esa pestaña es posible editar las dependencias del POM y añadir nuevas. Una ventana de diálogo permite explorar todos los artefactos del repositorio de Maven.



Es posible activar una vista especial que permite hojear los repositorios. Se activa en la opción *Window > Show View > Other...* y buscando *Maven* y seleccionando *Maven Repositories*.



Es posible también lanzar builds predeterminados de Maven y configurar esas ejecuciones.



### 1.1.13. Usando Maven con Mercurial

Para crear un repositorio Mercurial a partir de un proyecto Maven, hay que hacer lo habitual: inicializar el repositorio mercurial 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 que crea Maven a partir de los ficheros fuente originales deben ser ignorados.

El siguiente fichero `.hgignore` contiene las reglas que determinan los ficheros ignorados. Ignoramos los ficheros `.jar` porque Maven los obtiene automáticamente.

#### Fichero `.hgignore`

```
syntax: glob

# java deployment assets
*.class
*.jar

# mvn build trees
target/*
cargo/*
bin/*

# hg merge reject files and originals arising for reverts
*.rej
*.orig

# Mac idiosyncrasies
.DS_Store

# Eclipse local workspace configuration
.metadata/*

# Eclipse configuration files are created from Maven
.classpath
.project
.settings
```

### 1.1.14. Para saber más

---

Las siguientes referencias son de mucho interés para aprender más sobre Maven o como referencia para consultar dudas:

- [Maven in 5 minutes](#): Introducción rápida a Maven en la que se muestra cómo construir, empaquetar y ejecutar una sencilla aplicación.
- [Maven by Example](#): Libro de introducción a Maven realizado por Sonatype, la compañía creada por Jason Van Zyl el desarrollador principal de Maven. Disponible también en PDF en [esta dirección](#).
- [Maven, the complete reference](#): Otro libro muy recomendable de Sonatype. Disponible también en PDF en [esta dirección](#).
- [Java Power Tools](#): Excelente libro sobre 30 herramientas que ayudan en el desarrollo de software Java. El capítulo 1 está dedicado a Ant y el 2 a Maven. Disponible en [proquestcombo.safaribooksonline.com](#) (portal de Safari Books disponible para la UA) en [esta URL](#).
- [Maven](#): Página principal del proyecto Maven

## 1.2. Caso de estudio

---

### 1.2.1. 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.

### 1.2.2. Ingeniería de Requisitos

---

El Instituto de Educación Secundaria "jUA" 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 llegado a recopilar las siguientes funcionalidades a informatizar.

- Un usuario **administrador** puede realizar acciones relacionadas con la configuración de la aplicación: dar de alta nuevos usuarios, modificar ciertos parámetros, etc.
- La aplicación será utilizada por **bibliotecarios** y **usuarios** de la biblioteca.
- 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.
- Tanto bibliotecarios como usuarios podrán realizar con la aplicación **acciones sobre los libros y ejemplares**: consultar su disponibilidad, reservarlos, prestarlos, etc. Las acciones estarán limitadas al rol.
- Un bibliotecario se encargará de la **gestión de préstamos, libros y ejemplares**. La aplicación le permitirá registrar los préstamos y devoluciones de ejemplares que realiza en el mostrador de la biblioteca, así como consultar el estado de un determinado libro, ejemplar o usuario. Podrá también dar de alta libros, ejemplares, modificar su información y darlos de baja.
- Los **profesores** y **alumnos** van a poder realizar las siguientes acciones con la aplicación:
  - **Pedir prestado un ejemplar** disponible (que el bibliotecario le entregará cuando se pase por el mostrador)
  - **Reservar un libro** que tiene todos los ejemplares prestados
  - Consultar el **estado de los libros** y sus ejemplares: un libro puede estar prestado (en sala, en casa o en el departamento) o disponible.

- Además podrán hacer las siguientes **acciones físicas**:
  - **Recoger** un préstamo
  - **Devolver** un préstamo
- Para **recoger el ejemplar pedido** tanto los profesores como los alumnos deberán personarse en la biblioteca, coger el libro y solicitar su préstamo al bibliotecario. El bibliotecario utilizará la aplicación para buscar el ejemplar que ha pedido prestado y se lo dará.
- La **fecha de devolución del ejemplar** dependerá de si el usuario es alumno o profesor y empezará a contar a partir del momento en que el libro se toma prestado con la aplicación. El número máximo de libros que puede tener en préstamo un usuario dependerá también de si es profesor o alumno.
- Los usuarios pueden **reservar** libros con la aplicación cuando todos los ejemplares estén prestados. En el momento en que uno de los ejemplares se devuelva y quede disponible, la aplicación consultará la lista de reservas del libro y se lo prestará automáticamente al usuario que ha hecho la reserva más antigua. Le enviará una notificación para que sepa lo tiene prestado y que tiene que pasar a recogerlo por la biblioteca.
- El usuario puede **cancelar una reserva** en cualquier momento. Cuando la reserva se cancela o se realiza un préstamo asociado a ella, pasa a un histórico de reservas.
- Existe un **número máximo de préstamos y reservas** (operaciones) que puede hacer un usuario. La suma del número de préstamos y reservas de un usuario no puede sobrepasar este máximo. El máximo depende de si el usuario es un profesor o un alumno.
- El estado por defecto de un usuario es **activo**. Cuando el usuario se retrasa en la devolución de un préstamo pasa a estado **moroso**. En ese estado no puede pedir prestado ni reservar ningún otro libro. Cuando devuelve el libro se le crea una multa y pasa a estado **multado**.
- De la **multa** nos interesa saber la fecha de inicio y de finalización de la misma. La finalización de la multa dependerá del tipo de usuario. Nos han comunicado que quieren mantener un histórico de las multas que ha tenido un usuario. Cuando pasa la fecha de finalización, el estado del usuario vuelve a activo.

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.

#### 1.2.2.1. 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 y password*
- *Nombre y apellidos*

- *Correo electrónico*
- *Estado de un usuario*: activo, moroso o multado
- Datos referentes a su dirección, como son *calle, número, piso, ciudad y código postal*.
- Si el usuario es alumno, necesitaremos guardar quién es su *tutor*
- Si el usuario es profesor, necesitaremos su *departamento*

Existirá un tipo especial de usuario en el sistema, el **bibliotecario**, que tiene acceso a todas las funciones de gestión de libros, ejemplares y préstamos. Queremos guardar:

- *Login y password*
- *NIF*: identificador del bibliotecario

Cuando un usuario se retrase en la devolución de un libro, se le creará una **multa**. Nos han comunicado que quieren mantener un histórico de las multas que ha tenido un usuario. De cada multa nos interesa saber:

- *Usuario* que tiene la multa
- *Fecha de inicio*
- *Fecha de finalización*
- *Estado de la multa*: ACTIVA o HISTÓRICA

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

- *ISBN*
- *Título y autor*
- *Número de páginas*
- *Fecha de alta del libro*
- *Número de ejemplares disponibles*: cambiará conforme se presten y devuelvan ejemplares

En los **ejemplares** individuales de cada libro guardaremos información propia del ejemplar y además información del préstamo en el que caso en que algún usuario lo haya tomado prestado:

- *Libro* del que es ejemplar
- *Identificador del ejemplar*: código identificador del ejemplar
- *Fecha de adquisición*: fecha en la que el ejemplar se adquirió
- *Usuario* que lo ha tomado prestado (vacío si el ejemplar está disponible)
- *Fecha de inicio del préstamo*: fecha en la que el usuario ha tomado prestado el ejemplar
- *Fecha de devolución del préstamo*: fecha en la que el usuario debe devolver el ejemplar

Respecto a los **prestamos históricos** vamos a guardar:

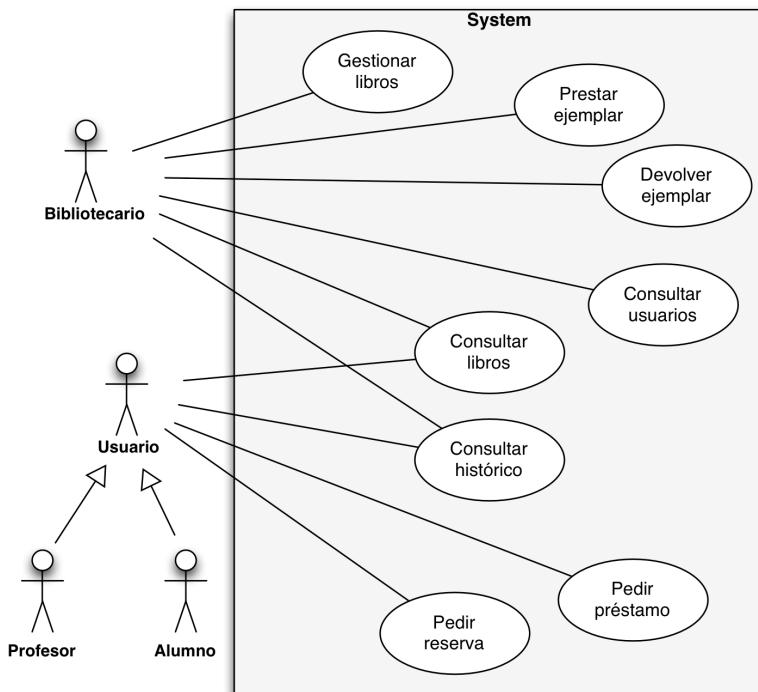
- *Fecha de inicio y finalización*
- *Usuario del préstamo*
- *Ejemplar del préstamo*

Por último, en las **reservas** guardaremos:

- *Usuario* que realiza la reserva
- *Libro que se reserva*
- *Fecha de reserva*: fecha en la que se realiza la reserva
- *Fecha de finalización de la reserva*: fecha en la que el usuario cancela la reserva o se activa un préstamo
- *Estado de la reserva*: ACTIVA o HISTÓRICA
- *Tipo de finalización de la reserva*: cancelada o préstamo

### 1.2.2.2. Casos de Uso

En un principio, el número de casos de uso es muy limitado. A lo largo del proyecto nos vamos a centrar en el desarrollo de los siguientes casos de uso generales:



- El **bibliotecario** podrá gestionar libros y ejemplares: consultándolos, modificándolos, dándolos de alta y eliminándolos. También podrá consultar y actualizar las operaciones relacionadas con los libros y los usuarios: prestar y devolver ejemplares. Relacionado con las operaciones, podrá consultar el histórico de préstamos y reservas. Y, por último, también podrá consultar los usuarios: sus préstamos, reservas y multas.
- El **usuario** podrá consultar los libros y ejemplares de la biblioteca, pedirlos prestados o reservarlos. Podrá también consultar sus propios libros y ejemplares prestados y reservados, así como su histórico de préstamos, reservas y multas.

Destacar que la **validación de usuarios** para todos los actores se considera una precondición que deben cumplir todos los casos de uso, y por lo tanto no se muestra en el diagrama. Entendemos también que la realización de una reserva por parte de un usuario

A continuación vamos a mostrar en mayor detalle los casos de uso separados por el tipo de usuario, de modo que quede más claro cuales son las operaciones que debe soportar la aplicación.

#### Un poco de UML...

Recordar que las relaciones entre casos de uso con estereotipo **<include>** representan que el caso de uso incluido se realiza siempre que se realiza el caso base. En cambio, el estereotipo **<extend>** representa que el caso de uso extendido puede realizarse cuando se realiza el caso de uso padre (o puede que no).

#### Bibliotecario

El tipo de usuario bibliotecario es el más complejo de nuestra aplicación. Como se puede observar, es el que va a poder realizar un mayor número de casos de uso:



Podemos observar tres consultas principales, a partir de las que se derivan posibles acciones adicionales, y una operación de "Alta de libros y ejemplares". El alta permite añadir nuevos libros y/o ejemplares.

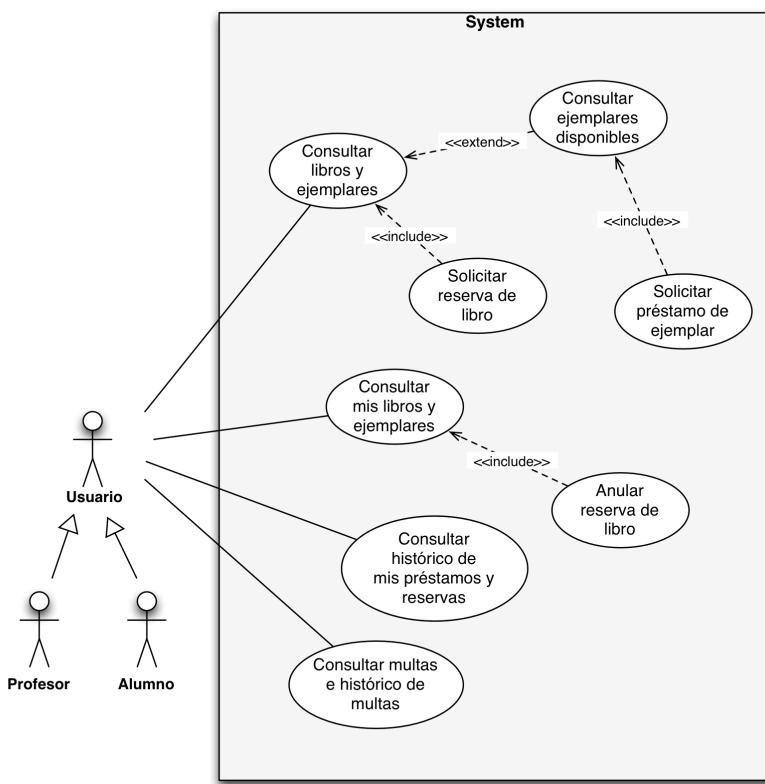
Las consultas son las siguientes:

- *Consultar libros y ejemplares*: el bibliotecario puede consultar tanto libros como ejemplares y, a partir del listado, seleccionar una serie de posibles acciones:
  - *Baja libros y ejemplares*: se puede dar de baja libros o ejemplares
  - *Modificación libros y ejemplares*: modificar los datos de los libros o ejemplares
  - *Consultar ejemplares prestados*: consultar qué ejemplares se encuentran prestados de un libro; se mostrará la información de los usuarios que han realizado el préstamo
  - *Consultar libros reservados*: consultar los libros con reservas; se mostrará la información de los usuarios que han realizado la reserva
  - *Consultar ejemplares disponibles*: consultar ejemplares disponibles de un libro
  - *Prestar ejemplar*: prestar en el mostrador un ejemplar a un usuario
  - *Devolver ejemplar*: devolver un ejemplar que se entrega en el mostrador

- *Consultar usuarios*: el bibliotecario puede consultar y seleccionar usuarios y hacer las siguientes acciones: *devolver un ejemplar* que el usuario entrega y tenía prestado, *entregar un préstamo* que el usuario ha realizado con la aplicación o consultar si el usuario está multado y su histórico de multas
- *Consultar histórico*: el bibliotecario puede consultar el histórico de préstamos y reservas realizadas por los usuarios

### Alumno o Profesor

En cuanto a un usuario cuyo tipo sea alumno o profesor, y por tanto, sea un usuario registrado en el sistema, las operaciones que puede realizar son las siguientes:



- *Consultar libros y ejemplares*: una vez buscados los libros y ejemplares, el usuario podrá tomar prestado un ejemplar disponible o reservar un libro cuyos ejemplares están prestados. Se deberá cumplir el cupo de préstamos y reservas. No podrán tomar prestado ni hacer reservas aquellos usuarios multados.
- *Consultar mis libros y ejemplares*: el usuario podrá consultar un listado de los libros y ejemplares que tiene prestados y reservados. Podrá también anular la reserva de un libro.
- *Consultar el histórico de mis préstamos y reservas*: el usuario podrá consultar y buscar en un listado del histórico de sus préstamos y reservas.

Más adelante, conforme cambien los requisitos del cliente (que siempre cambian), puede que el sistema permita renovar los préstamos a los usuarios registrados, que éstos puedan modificar su información de usuario, que los bibliotecarios obtengan informes sobre libros más prestados y/o reservados, etc...

### 1.2.2.3. Requisitos de Restricción (CRQ)

Respecto a las restricciones que se aplicarán tanto a los casos de uso como a los requisitos de información, y que concretan las reglas de negocio, hemos averiguado:

- Diferencias a la hora de realizar préstamos y reservas tanto por parte de un profesor como de un alumno:

	Número máximo de operaciones	Días de préstamo
Alumno	5	7
Profesor	8	30

Según esta información, el máximo de operaciones (reservas y préstamos) de un alumno es 6. Los libros prestados los tiene que devolver antes de 7 días.

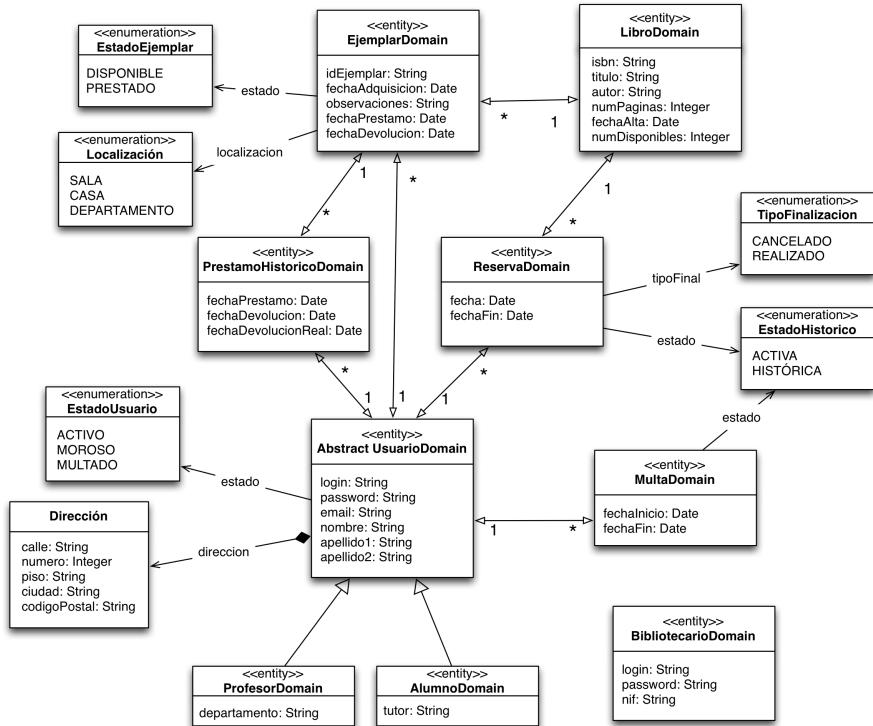
En el momento que un usuario tenga una demora en la devolución de un préstamo, se considerará al usuario moroso y se le impondrá una penalización del doble de días de desfase durante los cuales no podrá ni reservar ni realizar préstamos de libros.

### 1.2.3. Análisis y Diseño OO

A partir de esta captura de requisitos inicial, vamos a plantear los elementos que van a formar parte de la aplicación, comenzando por el modelo de clases.

#### 1.2.3.1. Modelo de Clases Conceptual

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.

¿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.

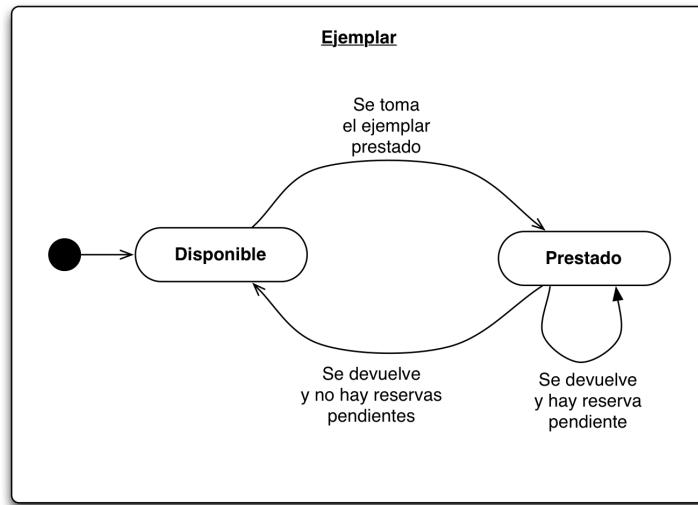
### **1.2.3.2. Diagramas de Estado**

A continuación podemos visualizar un par de diagramas que representan los diferentes estados que puede tomar tanto los ejemplares (por medio de las operaciones) como los usuarios (mediante las acciones que conllevan algunas operaciones).

## Estados de un ejemplar

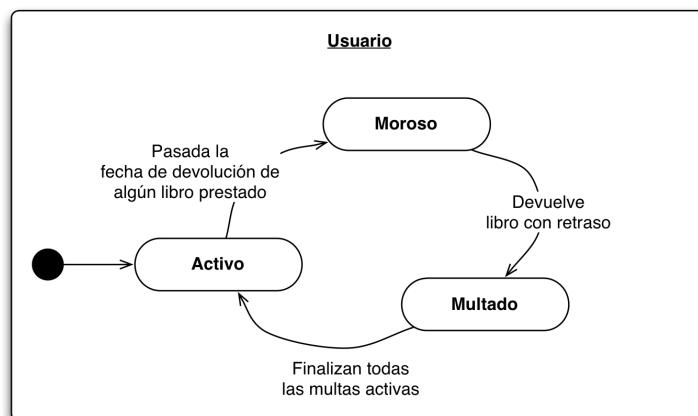
El estado de un ejemplar puede ser DISPONIBLE o PRESTADO. El cambio de un estado a otro cambia cuando se toma prestado y cuando se devuelve. Un caso especial es cuando el ejemplar se devuelve, pero el libro tiene una reserva. El libro continua prestado, ahora al usuario que ha hecho la reserva.

Lo representamos con el siguiente diagrama:



### **Estados de un usuario**

Del mismo modo, los estados de un usuario dependen de si el usuario realiza sus operaciones dentro de las reglas de negocio permitidas. Si la fecha de devolución de algún libro que tiene prestado un usuario vence sin que lo haya devuelto, el usuario pasa a ser MOROSO. Cuando el usuario devuelve el libro, pasa a ser MULTADO. Cuando el usuario ha cumplido todas las multas, vuelve a ser ACTIVO:





## 1.3. Implementación

El objetivo de la sesión de hoy es crear el proyecto `jbib-modelo`, módulo inicial en el que vamos a incluir todas las clases de dominio, los tipos enumerados y las clases auxiliares. Las clases de dominio implementan las entidades de la aplicación (objetos que vamos a hacer persistentes en la base de datos) y las podemos encontrar en el diagrama de clases presentado en el apartado anterior.

Vamos a construir el proyecto utilizando *Maven* como herramienta de gestión de *builds*. Al ser un proyecto formado por múltiples módulos, utilizaremos una estructura Maven de *multiproyecto*, en el que un directorio padre con un POM define las características comunes y contiene a distintos subdirectorios. Cada subdirectorio define un módulo y contiene su propio fichero POM que describe sus características específicas.

A lo largo de las siguientes sesiones del proyecto iremos construyendo nuevos módulos necesarios para la aplicación.

### 1.3.1. Paso a paso: Construcción del proyecto

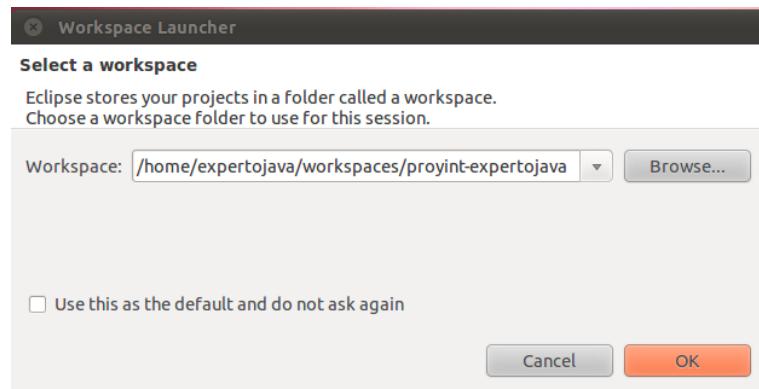
Vamos a crear el espacio de trabajo eclipse llamado `proyint-expertojava` dentro de `/home/expertojava/workspaces`. El espacio de trabajo contendrá el proyecto Maven con los distintos módulos que iremos programando a lo largo del curso. También crearemos un repositorio Mercurial en el que guardaremos el workspace y lo subiremos a Bitbucket.

1. Creamos el directorio `/home/expertojava/workspaces/proyint-expertojava` e inicializamos Mercurial:

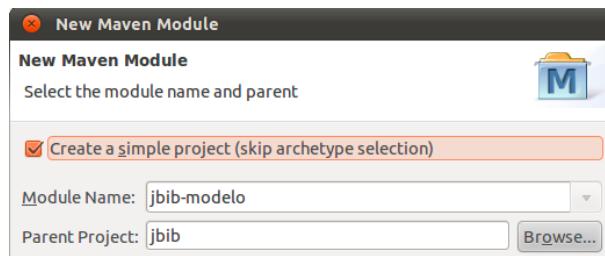
```
$ cd /home/expertojava/workspaces  
$ mkdir proyint-expertojava  
$ cd proyint-expertojava  
$ hg init .
```

2. Añadimos el fichero `.hgignore` para trabajar con Maven. Lo podemos copiar del repositorio Bitbucket [java ua/proyint-expertojava](#) (dejaremos allí la solución de esta sesión cuando se cumpla el plazo de entrega).

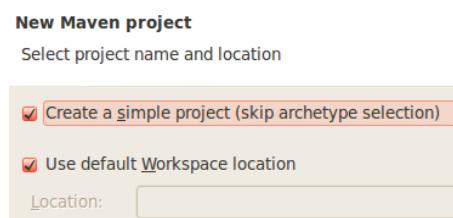
2. Una vez inicializado el repositorio Mercurial, abrimos un workspace de Eclipse en el directorio creado:



Vamos a utilizar el asistente de Eclipse para crear proyectos Maven. Pulsamos con el botón derecho en el panel de proyectos la opción *New > Project... > Maven > Maven Project*:

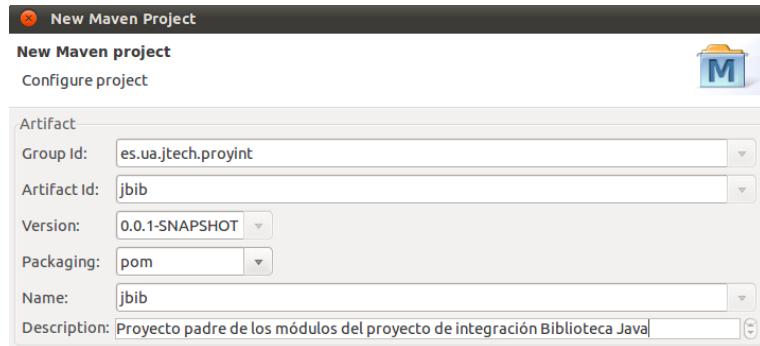


3. Creamos el proyecto padre de todos los módulos que iremos desarrollando más adelante. Seleccionamos la opción *skip archetype selection* para no basar el proyecto Maven en ningún arquetipo. Eclipse únicamente generará el proyecto con la estructura de directorios Maven adecuada y el POM mínimo con su descripción.

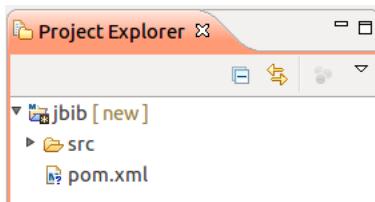


4. Introduce los datos del proyecto necesarios para crear el POM inicial de Maven con los siguientes parámetros:

- **GroupId:** es.ua.jtech.proyint
- **ArtifactId:** jbib
- **Version:** 0.0.1-SNAPSHOT
- **Packaging:** pom
- **Name:** jbib



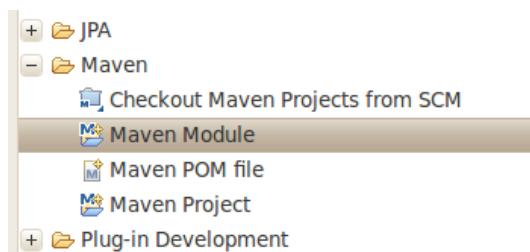
Eclipse creará un proyecto nuevo en el que existirá un directorio de fuentes (*src*) y el fichero POM que describe este proyecto padre. En el ícono del proyecto puedes ver una pequeña "M" que indica que se trata de un proyecto Maven y que el plugin Maven de Eclipse lo ha reconocido como tal.



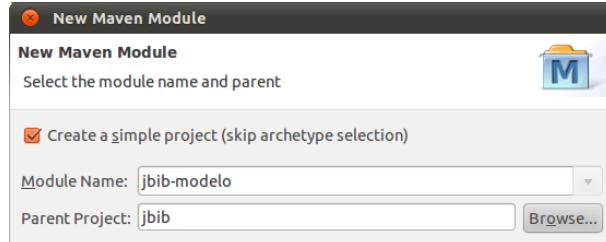
Podemos abrir el fichero POM para comprobar que el XML se ha creado correctamente:

```
<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>es.ua.jtech.proyint</groupId>
  <artifactId>jbib</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>pom</packaging>
  <name>jbib</name>
  <description>Proyecto padre de los módulos del proyecto de
    integración Biblioteca Java</description>
</project>
```

5. Procedemos a continuación a crear el módulo con el que vamos a trabajar en esta sesión: *jbib-modelo*. Lo hacemos también utilizando el asistente de Eclipse. Seleccionamos con el botón derecho la opción *New > Project... > Maven > Maven Module*:



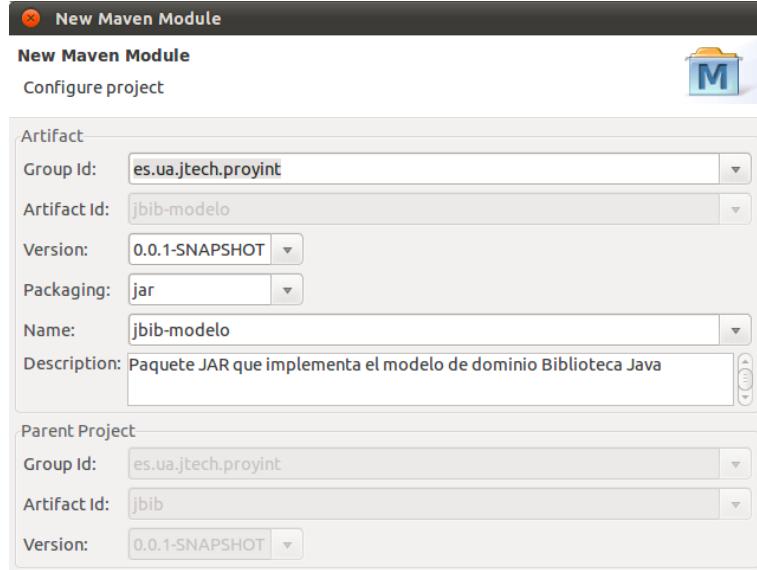
Marcamos también la opción de saltar la selección del arquetipo y escribimos el nombre del módulo y del proyecto padre:



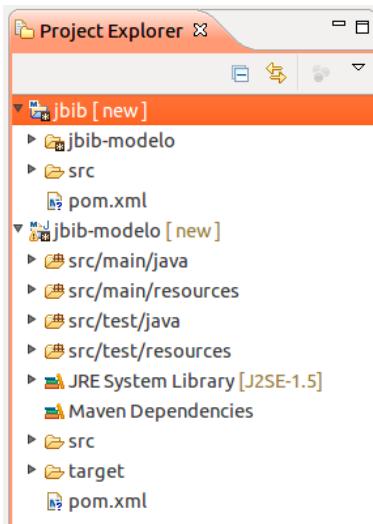
Introducimos a continuación los datos del módulo para crear su POM

- **GroupId:** es.ua.jtech.proyint
- **ArtifactId:** jbib-modelo
- **Version:** 0.0.1-SNAPSHOT
- **Packaging:** jar
- **Name:** jbib-modelo

Es importante el tipo de empaquetamiento. El módulo se va a empaquetar como un JAR conteniendo todas las clases del dominio y las clases de utilidad. Hay que definir esta opción. Ya veremos más adelante como Maven automatiza el proceso de compilación, prueba y empaquetamiento del JAR.



Podemos comprobar ahora que el proyecto se ha creado correctamente y que su estructura corresponde con la de un proyecto Maven:



Vemos que se ha creado un directorio de fuentes `src/main`, y otro directorio de tests `src/tests`. También vemos un directorio `target` donde Maven colocará los ficheros `.class` compilados y el JAR resultante del empaquetado del proyecto.

Si revisas los ficheros POM veremos que en el del proyecto padre se ha añadido el nombre del módulo recién creado. El POM del módulo contiene los datos introducidos:

```
<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>
  <parent>
    <artifactId>jbib</artifactId>
    <groupId>es.ua.jtech.proyint</groupId>
    <version>0.0.1-SNAPSHOT</version>
    <relativePath>..</relativePath>
  </parent>
  <artifactId>jbib-modelo</artifactId>
  <name>jbib-modelo</name>
  <description>Paquete JAR que implementa el modelo de dominio  
Biblioteca Java</description>
</project>
```

6. Hay que añadir algunos elementos a los ficheros POM para configurar correctamente los proyectos.

En primer lugar, podemos ver que el plug-in de Maven interpreta incorrectamente la versión de la máquina virtual Java del proyecto. En el proyecto aparece la J2SE-1.5, en lugar de la que está configurada por defecto en Eclipse, la 1.6. Para solucionar el aviso, indicamos en el POM del proyecto padre explícitamente que el proyecto va a trabajar con la versión 1.6 del compilador Java.

También indicamos que la codificación de los ficheros de código fuente que componen el proyecto es UTF-8 (la versión de Eclipse de la máquina virtual está configurado de esa

forma, se puede comprobar en *Windows > Preferences > General > Workspace > Text file encoding*).

Para ello añadimos el siguiente código al fichero POM del proyecto padre (*login-proyint-jbib/pom.xml*).

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

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

7. Para que el plugin de Maven actualice los proyectos, hay que pulsar con el botón derecho la opción *Maven > Update project configuration*. Hazlo en el proyecto hijo *jbib-metamodelo* y verás que en el explorador de proyectos se actualiza la versión de Java a la 1.6 y desaparece el aviso que existía.

8. Por último, añadimos en el proyecto padre las dependencias de algunas librerías que vamos a usar en todos los módulos del proyecto:

- Librería de pruebas **junit**: `junit-4.8.1`
- Librerías de gestión de logs **commons-logging** y **log4j**: `commons-logging-1.1.1` y `log4j-1.2.12`

```
...
</properties>

<dependencies>
  <dependency>
    <groupId>log4j</groupId>
    <artifactId>log4j</artifactId>
    <version>1.2.12</version>
    <type>jar</type>
    <scope>compile</scope>
  </dependency>

  <dependency>
    <groupId>commons-logging</groupId>
    <artifactId>commons-logging</artifactId>
    <version>1.1.1</version>
    <type>jar</type>
    <scope>compile</scope>
  </dependency>
```

```
</dependency>

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

<build>
...
```

El proyecto se actualizará automáticamente, y podremos ver las librerías importadas bajo las *Maven Dependencies*:



**Nota:**

También es posible añadir dependencias utilizando el asistente de Eclipse, que permite además buscar por nombre y versión.

Las librerías descargadas se guardan en el directorio `/home/expertojava/.m2` (repositorio local de Maven). Puedes explorar el directorio con el navegador de archivos para comprobar que los ficheros JAR se encuentran allí. En el repositorio existen más librerías porque las hemos descargado al montar la máquina virtual para que sea más rápida la construcción de los proyectos.

9. Desde la línea de comandos probamos a compilar con Maven el proyecto con el comando `mvn install` en el directorio del proyecto padre:

```
$ cd /home/expertojava/workspaces/proyint-expertojava/jbib
$ mvn install
```

Maven mostrará el resultado por la salida estándar, indicando el fichero JAR resultante y el directorio en el que se instala. Además de en el repositorio de Maven `.m2`, también lo deja en el directorio `target` del proyecto.

10. Añadimos todos los ficheros creados al repositorio Mercurial desde línea de comando o desde Eclipse. Desde línea de comando sería así:

```
$ cd /home/expertojava/workspaces/proyint-expertojava
$ hg add
$ hg commit -m "Añadido el proyecto padre jbib y el módulo jbib-modo
```

11. Por último, subimos el repositorio a Bitbucket. En Bitbucket creamos en la cuenta de estudiante el repositorio `proyint-expertojava`, y hacemos un push:

```
$ hg push https://bitbucket.org/login/proyint-expertojava
```

También creamos el fichero `.hg/hgrc` con la URL de push por defecto:

Fichero `.hg/hgrc`:

```
[paths]
default-push = ssh://bitbucket.org/login/proyint-expertojava
```

### 1.3.2. Primera iteración

Comentamos a continuación los pasos a seguir para desarrollar las clases de dominio del proyecto. Detallaremos una primera iteración en la que explicaremos cómo codificar una de las clases centrales (`LibroDomain`) y deberás hacer el ejercicio de desarrollar el resto.

#### 1.3.2.1. Primera clase de dominio

Las clases de dominio representan las entidades que van a hacerse persistentes y con las que va a trabajar 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. Serán también objetos que podremos utilizar como *Transfer Objects* (TOs) dentro del sistema, realizando funciones de meros contenedores y viajando por las capas de la aplicación.

Para asegurarnos que todos nuestros objetos de dominio tienen una estructura común, definimos una clase abstracta, que será la clase padre de todas las clases de dominio. La hacemos serializable para asegurar que los objetos pueden transmitirse entre capas físicas de la aplicación (por ejemplo, entre la capa de presentación y la capa de lógica de negocio) y definimos los métodos `equals()` y `hashCode()` para obligar a que las clases hijas implementen y redefinan la igualdad. Estos métodos son muy útiles en el caso en el que los objetos se guarden y queramos buscarlos en colecciones.

```
package es.ua.jtech.jbib.model;

import java.io.Serializable;

/**
 * Padre de todos los objetos de dominio del modelo. Los objetos del
 * dominio están relacionados entre si y se hacen persistentes
 * utilizando entidades JPA.
 * Pulsando F4 en Eclipse podemos ver la jerarquía. En la clase
 * definimos los métodos abstractos que deben implementar todas las
 * clases del dominio. Los objetos de dominio participan entre ellos
 * en relaciones uno-a-uno, uno-a-muchos y muchos-a-uno y están
 * relacionados mediante referencias.
 */
```

```
* Tendrán una clave única que permite identificarlos en la BD.  
* Una Direccion no es un objeto de dominio.  
*/  
  
public abstract class DomainObject implements Serializable {  
    private static final long serialVersionUID = 1L;  
    public abstract boolean equals(Object object);  
    public abstract int hashCode();  
}
```

Todas las entidades las vamos a definir dentro del paquete `es.ua.jtech.jbib.model`, y utilizaremos el sufijo `Domain` a modo de nomenclatura para indicar que un objeto de la aplicación es un objeto de dominio.

Cada objeto de dominio se compone de sus atributos, relaciones y de todos los getter/setter que encapsulan al objeto. Así pues, por ejemplo, la representación del *Domain Object* `LibroDomain` sería:

```
package es.ua.jtech.jbib.model;  
  
import java.util.Date;  
import java.util.HashSet;  
import java.util.Set;  
  
import org.apache.commons.logging.Log;  
import org.apache.commons.logging.LogFactory;  
  
public class LibroDomain extends DomainObject {  
  
    private static final long serialVersionUID = 1L;  
    private Log logger = LogFactory.getLog(LibroDomain.class);  
  
    private String isbn;  
    private String titulo;  
    private String autor;  
    private Integer numPaginas;  
    private Date fechaAlta;  
    private Integer numDisponibles;  
  
    private Set<EjemplarDomain> ejemplares =  
        new HashSet<EjemplarDomain>();  
    private Set<ReservaDomain> reservas =  
        new HashSet<ReservaDomain>();  
  
    public LibroDomain(String isbn) {  
        super();  
        this.isbn = isbn;  
        logger.debug("Creada una instancia de " +  
            LibroDomain.class.getName());  
    }  
  
    public String getIsbn() {  
        return isbn;  
    }  
  
    public void setIsbn(String isbn) {  
        this.isbn = isbn;  
    }  
  
    public String getTitulo() {  
        return titulo;  
    }
```

```

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

public String getAutor() {
    return autor;
}

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

public Integer getNumPaginas() {
    return numPaginas;
}

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

public Date getFechaAlta() {
    return fechaAlta;
}

public void setFechaAlta(Date fechaAlta) {
    this.fechaAlta = fechaAlta;
}

public Set<EjemplarDomain> getEjemplares() {
    return ejemplares;
}

public void setEjemplares(Set<EjemplarDomain> ejemplares) {
    this.ejemplares = ejemplares;
}

public Set<ReservaDomain> getReservas() {
    return reservas;
}

public void setReservas(Set<ReservaDomain> reservas) {
    this.reservas = reservas;
}

public IntegergetNumDisponibles() {
    return numDisponibles;
}

public void setNumDisponibles(Integer numDisponibles) {
    this.numDisponibles = numDisponibles;
}

@Override
public int hashCode() {
    final int prime = 31;
    int result = 1;
    result = prime * result + ((isbn == null) ? 0 : isbn.hashCode());
    return result;
}

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

```

```
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;
        LibroDomain other = (LibroDomain) obj;
        if (isbn == null) {
            if (other.isbn != null)
                return false;
        } else if (!isbn.equals(other.isbn))
            return false;
        return true;
    }
}
```

Es interesante hacer notar la forma en que están implementados los métodos `hashCode` y `equals`. Son métodos muy importantes porque son los que se utilizan para buscar en las colecciones. Los podemos generar con Eclipse usando la opción *Source > Generate hashCode() and equals()*...

Para elegir los campos a usar en la comparación debemos tener en cuenta que sean claves naturales de los objetos de dominio, tal y como se explica en el [esta nota](#) de jBoss e Hibernate. También es importante que garanticemos que los campos siempre existen, utilizándolos también en el constructor. El constructor lo podemos generar con la opción de Eclipse *Source > Generate Constructor using fields*....

En la próxima sesión, cuando realizemos el mapeado al modelo relacional, a la base de datos, añadiremos un identificador único en cada clase para garantizar la identidad de cada objeto y tener más flexibilidad. Pero es importante que esta identidad será sólo para trabajar con la base de datos. Desde el punto de vista de un objeto de dominio, la identidad debe estar basada en claves naturales propias del objeto.

### 1.3.2.2. Eliminando errores

Definimos las clases vacías y los tipos enumerados necesarias para eliminar todos los errores y para completar el módulo (consultar el diagrama de clases UML):

- Clase `EjemplarDomain`
- Clase `PrestamoHistoricoDomain`
- Clase abstracta `UsuarioDomain` y sus clases hijas `ProfesorDomain` y `AlumnoDomain`.
- Clase `ReservaDomain`
- Clase `MultaDomain`
- Clase `Direccion`
- Clase `BibliotecarioDomain`

### 1.3.2.3. Ficheros de recursos

Añadimos los ficheros de recursos necesarios para el funcionamiento de los logs.

Fichero `src/main/resources/commons-logging.properties`:

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

Fichero **src/main/resources/log4j.properties**:

```
# A#ade appender A1
log4j.rootLogger=DEBUG, A1

# A1 se redirige a la consola
log4j.appender.A1=org.apache.log4j.ConsoleAppender
# Coloca el nivel root del logger en INFO (muestra mensajes de INFO hacia arriba)
log4j.appender.A1.Threshold=INFO

# A1 utiliza PatternLayout
log4j.appender.A1.layout=org.apache.log4j.PatternLayout
log4j.appender.A1.layout.ConversionPattern=[%d{dd/MM/yyyy HH:mm:ss}] %p -
%m %n
```

Podemos también añadir un fichero de configuración de logs distinto en el directorio de test, en el que cambiamos el nivel de los tests a DEBUG:

Fichero **test/main/resources/log4j.properties**:

```
# A#ade appender A1
log4j.rootLogger=DEBUG, A1

# A1 se redirige a la consola
log4j.appender.A1=org.apache.log4j.ConsoleAppender
# Coloca el nivel root del logger en DEBUG (muestra mensajes de DEBUG hacia arriba)
log4j.appender.A1.Threshold=DEBUG

# A1 utiliza PatternLayout
log4j.appender.A1.layout=org.apache.log4j.PatternLayout
log4j.appender.A1.layout.ConversionPattern=[%d{dd/MM/yyyy HH:mm:ss}] %p -
%m %n
```

### 1.3.2.4. Primer test

Definimos el primer test con la clase `es.ua.jtech.jbib.model.LibroDomainTest` en el directorio `src/test/java`, con el que probamos el funcionamiento correcto de la igualdad en la clase `LibroDomain`:

```
[package es.ua.jtech.jbib.model;

import static org.junit.Assert.*;
import org.junit.Test;

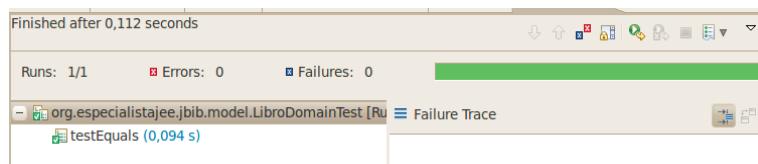
public class LibroDomainTest {

    /**
     * Dos libros son iguales cuando tienen el mismo ISBN
     */
    @Test
    public void testEquals() {
        LibroDomain libro1 = new LibroDomain("123456789");
        LibroDomain libro2 = new LibroDomain("123456789");
        assertTrue(libro1.equals(libro2));
        libro2.setIsbn(null);
        assertFalse(libro1.equals(libro2));
    }
}
```

```
}

/*
 * Prueba del funcionamiento de la pertenencia a
 * la colección de ejemplares
 */
@Test
public void testContainsEjemplares() {
    LibroDomain libro1 = new LibroDomain("123456789");
    EjemplarDomain ejemplar1 = new EjemplarDomain(libro1, "A");
    EjemplarDomain ejemplar2 = new EjemplarDomain(libro1, "B");
    EjemplarDomain ejemplar3 = new EjemplarDomain(libro1, "C");
    libro1.getEjemplares().add(ejemplar1);
    libro1.getEjemplares().add(ejemplar2);
    // Comprueba igualdad de referencia
    assertTrue(libro1.getEjemplares().contains(ejemplar1));
    assertFalse(libro1.getEjemplares().contains(ejemplar3));
    // Comprueba igualdad de valor
    assertTrue(libro1.getEjemplares().contains(
        new EjemplarDomain(libro1, "A")));
}
}
```

Ejecutamos el test con el botón derecho sobre la clase o el paquete: *Run As > JUnit Test* y debe aparecer en verde:



### 1.3.2.5. Construcción con Maven

Abrimos un terminal y ejecutamos los comandos Maven para construir el JAR que contiene el proyecto:

```
$ cd /home/expertojava/workspaces/proyint-expertojava
$ mvn clean
$ mvn install
```

El comando Maven `clean` borra todos los ficheros `.class` y `.jar` que puedan haber sido creados anteriormente. El comando `install` realiza secuencialmente las siguientes acciones:

- Compila todos los módulos del proyecto principal
- Ejecuta todos los tests
- Si los tests son correctos, empaqueta el subproyecto `bib-modelo` en el fichero JAR `jbib-modelo-0.0.1-SNAPSHOT.jar` que deja en el directorio `target` del proyecto
- Copia el fichero JAR en el repositorio local de Maven (directorio `.m2`) para dejarlo como una librería disponible para otros proyectos.

La última parte de la salida del comando debe ser esta:

```

T E S T S
-----
Running es.ua.jtech.jbib.model.LibroDomainTest
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.144 sec

Results :

Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
...

[INFO] Reactor Summary:
[INFO]
[INFO] jbib ..... SUCCESS [0.369s]
[INFO] jbib-modelo ..... SUCCESS [1.342s]
[INFO]

[INFO] BUILD SUCCESS
[INFO]

[INFO] Total time: 1.882s
[INFO] Finished at: Sat Oct 08 08:07:18 CEST 2012
[INFO] Final Memory: 4M/15M
[INFO]
-----
```

Todos los ficheros .class y JAR se guardan en el directorio `target` del proyecto. Puedes explorar el sistema de archivos para comprobarlo.

Esto lo podemos hacer también desde el asistente de Eclipse situándonos en el proyecto principal y pulsando con el botón derecho la opción *Run As > Maven install*. Puede ser que la salida de la consola sea ligeramente distinta, porque Eclipse usa una versión de Maven instalada en el propio entorno.

Para usar la misma versión que cuando la lanzamos desde línea de comandos, podemos seleccionar la opción *Window > Preferences > Maven > Installations* y añadir la ruta en la que se encuentra Maven en el sistema operativo: `/opt/apache-maven-3.0.3`.

### 1.3.2.6. Repositorio Bitbucket

Terminamos esta primera iteración haciendo un commit con los cambios y subiéndolos al repositorio Bitbucket.

Configura el repositorio para que el profesor del módulo (usuario: domingogallardo) y el administrador de java\_ue (usuario: java\_ue) tengan permiso de lectura.

### 1.3.2.7. Enumeraciones

En cuanto a las enumeraciones, Java (desde su versión 5.0) permite su creación mediante la clase `java.lang.Enum`. En nuestro caso, por ejemplo, la enumeración de `EstadoUsuario` quedaría del siguiente modo:

```

package es.ua.jtech.jbib.model;
public enum EstadoUsuario {
```

```
    ACTIVO, MOROSO, MULTADO  
}
```

Definimos al menos las enumeraciones (consulta el diagrama de clases):

- EstadoLibro
- EstadoHistorico
- EstadoUsuario
- Localizacion

### 1.3.3. Implementar y completar las clases de entidad

Para implementar nuestras clases, tendremos que codificar todas las clases y relaciones expuestas en el [Modelo de Clases Conceptual](#).

Hay que tener especial cuidado con las relaciones de herencia. Tenemos la clase `UsuarioDomain` y las clases hijas `ProfesorDomain` y `AlumnoDomain`. La clase padre es abstracta.

Algunas características comunes a todas las clases de entidad:

- Definimos el constructor y los métodos `hashCode` y `equals` usando campos que constituyan claves naturales.
- 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).

Definimos también la clase `Direccion` que será una clase no entidad y se utilizará en la entidad `UsuarioDomain`.

### 1.3.4. Gestión de las Excepciones

Todas las aplicaciones empresariales definen una política de gestión de las excepciones de aplicación.

En nuestro caso, conforme crezca el proyecto, iremos creando nuevas excepciones. Como punto de partida, y como buena norma de programación, vamos a definir una excepción genérica de tipo *unchecked* (`BibliotecaException`), que será la excepción padre de todas las excepciones de aplicación de la biblioteca. El hecho de que la excepción sea *unchecked* remarca el carácter de que estamos definiendo excepciones relacionadas 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.

Definimos las excepciones en el paquete global es.ua.jtech.jbib

```
package es.ua.jtech.jbib;

public class BibliotecaException extends RuntimeException {
    private static final long serialVersionUID = 1L;

    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*.

### 1.3.5. Implementación de las 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:

```
package es.ua.jtech.jbib;

// Imports

/**
 * Reglas de Negocio de la Biblioteca BR = Business Rules
 *
 * 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 " + BibliotecaBR.class);
    }

    public static BibliotecaBR getInstance() {
        return me;
    }

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

    /**
     * Valida que el número de operaciones realizadas por un determinado
     * tipo de usuario se inferior o igual al cupo definido
     *
     * @param usuario
     *         objeto UsuarioDomain
     * @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(UsuarioDomain usuario, int numOp)
        throws BibliotecaException {
        String msg;
        if (!(usuario instanceof AlumnoDomain)
            && !(usuario instanceof ProfesorDomain)) {
            msg = "Solo los alumnos y profesores pueden tener libros prestados";
            logger.error(msg);
            throw new BibliotecaException(msg);
        }
        if ((usuario instanceof AlumnoDomain && numOp >
            cupoOperacionesAlumno)
            || (usuario instanceof ProfesorDomain && numOp >
            cupoOperacionesProfesor)) {
            msg = "El cupo de operaciones posibles esta lleno";
            logger.error(msg);
            throw new BibliotecaException(msg);
        }
    }

    /**

```

```

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

```

Podemos observar que vamos a tener un método por cada regla de negocio, y que estos métodos lanzarán excepciones de aplicación en el caso de un comportamiento anómalo.

### 1.3.6. Tests

Vamos a definir dos tipos de tests, unos relacionados con las clases del modelo y otros con las reglas de negocio.

Los tests se deben colocar dentro de la carpeta test, en el mismo paquete que la clase a probar.

Ejemplo de tests de reglas de negocio:

```

package es.ua.jtech.jbib;

import static org.junit.Assert.*;
import org.junit.Test;

import es.ua.jtech.jbib.model.AlumnoDomain;
import es.ua.jtech.jbib.model.ProfesorDomain;

/**
 * Pruebas jUnit sobre las reglas de negocio de la biblioteca
 */
public class BibliotecaBRTTest {

    @Test
    public void testCalculaNumDiasPrestamoProfesor() {
        int diasProfesor = BibliotecaBR.getInstance()
            .calculaNumDiasPrestamo(
                new ProfesorDomain("pedro.garcia", "1234"));
        assertEquals(30, diasProfesor);
    }

    @Test

```

```
public void testCalculaNumDiasPrestamoAlumno() {
    // TODO
}

@Test
public void testCupoOperacionesProfesor() {
    // TODO
}

@Test
public void testCupoOperacionesAlumno() {
    // TODO
}

@Test
public void testCompruebaCupoOperacionesProfesorCorrecto() {
    try {
        ProfesorDomain profesor =
            new ProfesorDomain("pedro.garcia", "1234");
        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 testCompruebaCupoOperacionesProfesorIncorrecto()
    throws BibliotecaException {
    BibliotecaBR.getInstance()
        .compruebaCupoOperaciones(
            new ProfesorDomain("pedro.garcia", "1234"), 9);
}

@Test
public void testCompruebaCupoOperacionesAlumnoCorrecto() {
    // TODO
}

@Test(expected = BibliotecaException.class)
public void testCompruebaCupoOperacionesAlumnoIncorrecto()
    throws BibliotecaException {
    // TODO
}
```

En los tests de las clases del modelo debemos de comprobar:

- Relaciones de igualdad.
- Actualizaciones de las relaciones entre entidades. Por ejemplo, podemos crear una operación, un libro y un usuario, asociar la operación al libro y al usuario y comprobar que los métodos `get` devuelven correctamente las relaciones.

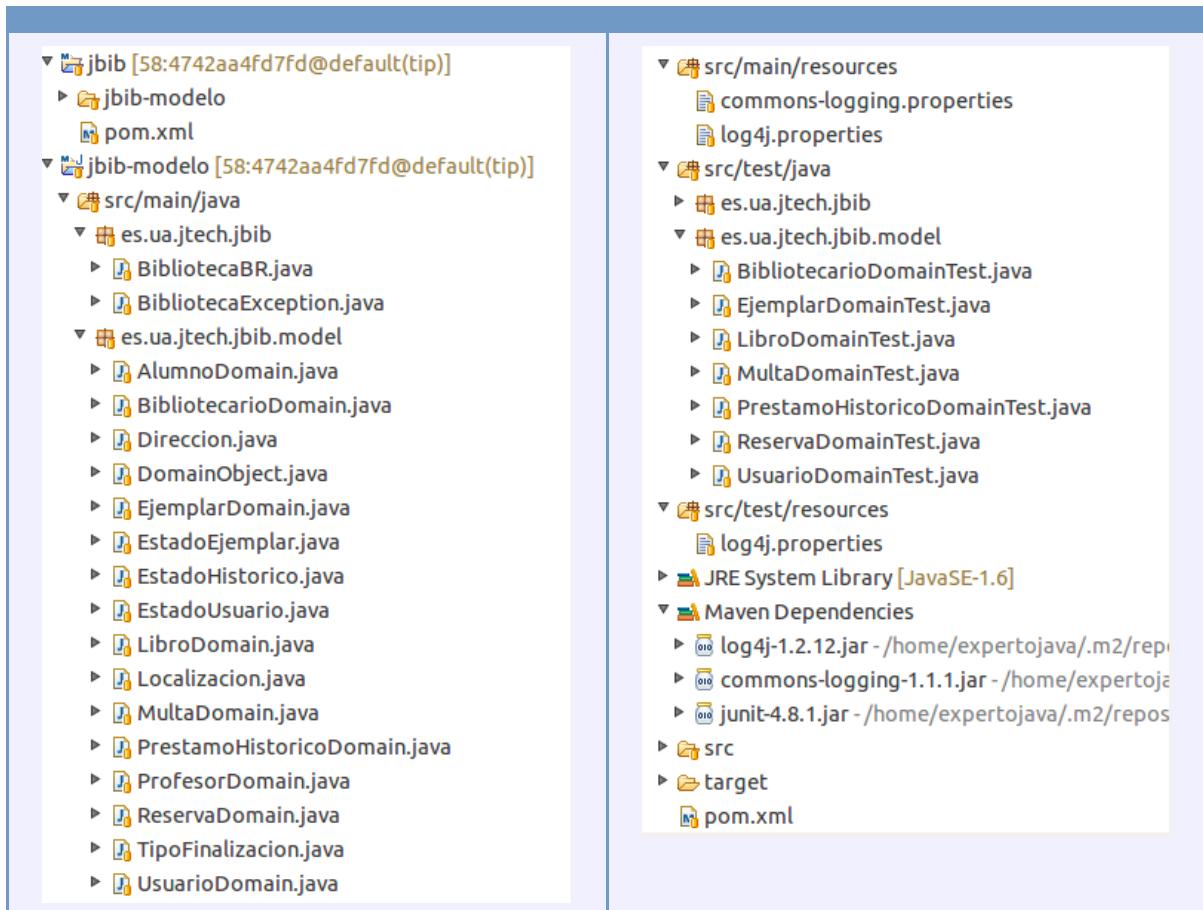
## 1.4. Resumen

En esta sesión vamos a preparar la base para el resto de sesiones. Por ellos, debemos crear un proyecto padre Maven, al que llamaremos **jbib**, que contendrá al módulo **jbib-modelo**

con el modelo de dominio de la aplicación:

1. modelo de objetos dentro del paquete `es.ua.jtech.jbib.model`, implementando cada entidad con los atributos representados en el diagrama UML, sus relaciones, y teniendo en cuenta la relación de herencia con la clase `DomainObject` y los constructores necesarios y los métodos de acceso.
2. clase `BibliotecaBR` con las reglas de negocio, implementada como un singleton, la cual debe pasar las pruebas JUnit aportadas. Para implementar esta clase, es necesaria la clase `BibliotecaException`.
3. tests de las clases de dominio y de las reglas de negocio que realicen algunas comprobaciones de los métodos `equals` y de las actualizaciones de las relaciones.

Las siguientes imágenes muestran todos los archivos que debe contener el proyecto Eclipse:



Debemos utilizar la etiqueta `entrega-proyint-modelo`. El plazo final de entrega será el **jueves 15 de noviembre**.

## 2. Capa de datos y negocios con JPA

### 2.1. Creación de entidades de dominio con JPA

El objetivo de esta primera parte de la sesión es añadir al proyecto jbib-modelo las clases y anotaciones necesarias para convertir las clases de dominio en entidades persistentes basadas en JPA. Adelantando algunos detalles, en esta sesión deberemos desarrollar los siguientes elementos:

- Añadir la configuración de JPA al proyecto jbib-modelo:
  - Dependencias de Hibernate en el fichero de configuración de Maven `pom.xml`
  - Fichero de configuración de la unidad de persistencia JPA `META-INF/persistence.xml` (del ámbito principal y del test)
- Añadir anotaciones en las clases de dominio para mapearlas en tablas
- Validar el esquema SQL resultante de la base de datos
- Desarrollar las clases DAO que proporcionan una capa de abstracción para la recuperación, modificación y borrado de las entidades
- Comprobar y ampliar tests que comprueben el funcionamiento de las entidades y de los DAO utilizando DbUnit

Vamos a ello paso a paso.

#### 2.1.1. Configuración de JPA

En primer lugar debes añadir en el fichero de Maven `pom.xml` del proyecto jbib-modelo las dependencias necesarias para trabajar con:

- JPA/Hibernate
- bases de datos MySQL y Apache Derby. Apache Derby es una base de datos que funciona en memoria y que vamos a utilizar para hacer más eficientes los tests.
- Plugin `hibernate3-maven` para poder generar el esquema de la base de datos por línea de comandos

Las dependencias son las siguientes:

```
<dependencies>
    <!-- MySql Connector -->
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>5.1.10</version>
        <scope>runtime</scope>
    </dependency>
    <!-- Hibernate -->
```

```

<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-entitymanager</artifactId>
    <version>3.6.10.Final</version>
</dependency>

<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-log4j12</artifactId>
    <version>1.6.1</version>
    <scope>runtime</scope>
</dependency>

</dependencies>

<build>
    <plugins>

        <!-- Plugin hibernate3-maven -->

        <plugin>
            <groupId>org.codehaus.mojo</groupId>
            <artifactId>hibernate3-maven-plugin</artifactId>
            <version>2.2</version>
            <configuration>
                <components>
                    <component>
                        <name>hbm2ddl</name>
                        <implementation>jpaconfiguration</implementation>
                    </component>
                </components>
                <componentProperties>
                    <persistenceunit>biblioteca</persistenceunit>
                    <outputfilename>schema.ddl</outputfilename>
                    <drop>false</drop>
                    <create>true</create>
                    <export>false</export>
                    <format>true</format>
                </componentProperties>
            </configuration>
        </plugin>
    </plugins>
</build>

```

Añadimos los ficheros de configuración de JPA en el ámbito de test de Maven. El ámbito de test lo usaremos para crear una base de datos vacía, poblarla con datos de prueba y comprobar que las clases funcionan correctamente.

Copia los ficheros commons-logging.properties y log4j.properties al ámbito de test (src/test/resources/) y añade las siguientes líneas al segundo, para activar los logs de Hibernate a nivel de INFO.

```

# Hibernate logging options (INFO only shows startup messages)
log4j.logger.org.hibernate=INFO

# Log JDBC bind parameter runtime arguments
#log4j.logger.org.hibernate.type=ALL

```

Continua creando el fichero de configuración de JPA en el directorio de recursos de tests src/test/resources/META-INF/persistence.xml con el siguiente contenido:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="1.0"
    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/persistence_1_0.xsd">

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

        <properties>
            <property name="hibernate.dialect"
                value="org.hibernate.dialect.MySQLInnoDBDialect" />
            <property name="hibernate.connection.driver_class"
                value="com.mysql.jdbc.Driver" />
            <property name="hibernate.connection.username" value="root" />
            <property name="hibernate.connection.password" value="expertojava" />
            <property name="hibernate.connection.url"
                value="jdbc:mysql://localhost:3306/biblioteca" />
            <property name="hibernate.hbm2ddl.auto" value="create" />
            <property name="hibernate.show_sql" value="true" />
        </properties>
    </persistence-unit>
</persistence>
```

Es importante hacer notar que el valor de `hibernate.hbm2ddl.auto` está puesto a `create`. Esto significa que el esquema de base de datos se creará de nuevo cada vez que lancemos los tests.

**Nota:**

En esta primera parte de la sesión vamos a crear el esquema inicial de base de datos a partir del modelo de clases. Despues refinaremos el modelo de clases, añadiendo anotaciones `@Table`, `@Column` y `@JoinColumn` para adaptarlo al esquema SQL único que proporcionaremos más adelante.

### 2.1.2. Test de la unidad de persistencia

Creamos el test que carga la unidad de persistencia vacía, en la que todavía no hemos añadido ninguna entidad. Lo hacemos en el directorio `src/test/java` y en el paquete `es.ua.jtech.jbib.persistence`:

```
package es.ua.jtech.jbib.persistence;

public class PersistenceUnitTest {

    private static Log logger = LogFactory
        .getLog(PersistenceUnitTest.class);
    private static EntityManagerFactory emf;

    @BeforeClass
    public static void initDatabaseTest() {
        try {
            logger.info("Building JPA EntityManagerFactory for unit tests");
            emf = Persistence.createEntityManagerFactory("biblioteca");
        } catch (Exception ex) {
            ex.printStackTrace();
            fail("Exception during JPA EntityManager instantiation.");
        }
    }
}
```

```

    }

    @AfterClass
    public static void closeEntityManagerFactory() {
        logger.info("Closing JPA EntityManagerFactory");
        if (emf != null)
            emf.close();
    }
}

```

La anotación `@BeforeClass` hace que el método `initDatabase()` se ejecute sólo una vez, antes de ejecutar todos los tests. En el método se obtiene la factoría de entity managers y se guarda en la variable estática `emf` a la que se accederá desde todos los tests para obtener entity managers.

La anotación `AfterClass` hace que el método se ejecute al terminar todos los tests y cierra la factoría de entity managers.

Lanza el test desde Eclipse y comprueba que todo funciona correctamente.

Abre un terminal y lanza todos los tests del proyecto usando Maven:

```
$ cd proyint-expertojava/jbib
$ mvn test
```

### 2.1.3. Anotaciones de las clases de dominio

Vamos ahora a crear las entidades anotando todas las clases de dominio y creando un esquema de base de datos inicial.

Empieza añadiendo todas las entidades en el fichero `src/test/resources/META-INF/persistence.xml`

```

<persistence-unit ...>
    <class>es.ua.jtech.jbib.model.EjemplarDomain</class>
    <class>es.ua.jtech.jbib.model.LibroDomain</class>
    <class>es.ua.jtech.jbib.model.ReservaDomain</class>
    <class>es.ua.jtech.jbib.model.UsuarioDomain</class>
    <class>es.ua.jtech.jbib.model.AlumnoDomain</class>
    <class>es.ua.jtech.jbib.model.ProfesorDomain</class>
    <class>es.ua.jtech.jbib.model.BibliotecarioDomain</class>
    <class>es.ua.jtech.jbib.model.PrestamoHistoricoDomain</class>
    <class>es.ua.jtech.jbib.model.MultaDomain</class>

    <properties>
        ...
    </properties>

```

Y añade también un test básico que prueba a cargar el *entity manager* en la clase `PersistenceUnitTest`:

```

@Test
public void createEntityManagerTest() {
    EntityManager em = emf.createEntityManager();
    assertNotNull(em);
    em.close();
}

```

Si lanzas ahora los tests con `mvn test` verás que salta un error debido a que las entidades no están definidas. Debes añadir ahora las anotaciones necesarias en las clases de dominio para mapear las entidades con las tablas de la base de datos. Los tipos de los campos deben corresponder con los definidos en las clases.

Para cada entidad añade un nuevo atributo llamado `id` de tipo `Long` que defina su clave primaria en la base de datos. Mapéalo con la anotación `@Id @GeneratedValue(strategy=GenerationType.AUTO)`.

Hay que añadir las anotaciones de JPA a las siguientes clases en el paquete `es.ua.jtech.jbib.model`:

- `EjemplarDomain`
- `LibroDomain`
- `MultaDomain`
- `UsuarioDomain` (abstracta) y su relación de herencia con `AlumnoDomain` y `ProfesorDomain`
- `BibliotecarioDomain`
- `PrestamoHistoricoDomain`

Como referencia, mostramos a continuación el esquema SQL completo de la base de datos resultante:

```
create database biblioteca;
use biblioteca;

create table bibliotecario (
    id bigint not null auto_increment,
    email varchar(255),
    login varchar(255) unique,
    nif varchar(255),
    password varchar(255),
    primary key (id)
) type=InnoDB;

create table ejemplar (
    id bigint not null auto_increment,
    fechaAdquisicion date,
    fechaDevolucion date,
    fechaPrestamo date,
    idEjemplar varchar(255),
    localizacion enum('SALA', 'CASA', 'DEPARTAMENTO') not null,
    observaciones varchar(255),
    libro_id bigint not null,
    usuario_id bigint,
    primary key (id)
) type=InnoDB;

create table historico (
    id bigint not null auto_increment,
    fechaDevolucion date,
    fechaDevolucionReal date,
    fechaPrestamo date,
    ejemplar_id bigint not null,
    usuario_id bigint not null,
    primary key (id)
) type=InnoDB;
```

```

create table libro (
    id bigint not null auto_increment,
    autor varchar(255),
    fechaAlta date,
    isbn varchar(255) unique,
    numDisponibles integer,
    numPaginas integer,
    titulo varchar(255),
    primary key (id)
) type=InnoDB;

create table multa (
    id bigint not null auto_increment,
    estado enum('ACTIVA','HISTORICA') not null,
    fechaFin date,
    fechaInicio date,
    usuario_id bigint not null,
    primary key (id)
) type=InnoDB;

create table reserva (
    id bigint not null auto_increment,
    estado enum('ACTIVA','HISTORICA') not null,
    fecha date,
    fechaFin date,
    tipoFinal enum('CANCELADO','FINALIZADO'),
    libro_id bigint not null,
    usuario_id bigint not null,
    primary key (id)
) type=InnoDB;

create table usuario (
    tipo varchar(31) not null,
    id bigint not null auto_increment,
    apellido1 varchar(255),
    apellido2 varchar(255),
    calle varchar(255),
    ciudad varchar(255),
    codigoPostal varchar(255),
    numero integer,
    piso varchar(255),
    email varchar(255) unique,
    estado enum('ACTIVO', 'MOROSO', 'MULTADO') not null,
    login varchar(255) unique,
    nombre varchar(255),
    password varchar(255),
    tutor varchar(255),
    departamento varchar(255),
    primary key (id)
) type=InnoDB;

alter table ejemplar
    add index FK95B940BAA013FCAC (usuario_id),
    add constraint FK95B940BAA013FCAC
        foreign key (usuario_id)
        references usuario (id);

alter table ejemplar
    add index FK95B940BA930E18AC (libro_id),
    add constraint FK95B940BA930E18AC
        foreign key (libro_id)
        references libro (id);

alter table historico

```

```
add index FK66D8DFF0A013FCAC (usuario_id),
add constraint FK66D8DFF0A013FCAC
foreign key (usuario_id)
references usuario (id);

alter table historico
add index FK66D8DFF0F97DB50 (ejemplar_id),
add constraint FK66D8DFF0F97DB50
foreign key (ejemplar_id)
references ejemplar (id);

alter table multa
add index FK636D531A013FCAC (usuario_id),
add constraint FK636D531A013FCAC
foreign key (usuario_id)
references usuario (id);

alter table reserva
add index FK41640CB8A013FCAC (usuario_id),
add constraint FK41640CB8A013FCAC
foreign key (usuario_id)
references usuario (id);

alter table reserva
add index FK41640CB8930E18AC (libro_id),
add constraint FK41640CB8930E18AC
foreign key (libro_id)
references libro (id);
```

Listamos a continuación como ejemplo las clases EjemplarDomain, UsuarioDomain y sus clases hijas AlumnoDomain y ProfesorDomain:

Clase EjemplarDomain:

```
package es.ua.jtech.jbib.model;

// imports

@Entity
@Table(name = "ejemplar")
public class EjemplarDomain extends DomainObject {

    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    private String idEjemplar;
    @Enumerated(EnumType.STRING)
    @Column(columnDefinition="enum('SALA', 'CASA', 'DEPARTAMENTO')",
            nullable=false)
    private Localizacion localizacion;
    @Temporal(TemporalType.DATE)
    private Date fechaAdquisicion;
    private String observaciones;
    @Temporal(TemporalType.DATE)
    private Date fechaPrestamo;
    @Temporal(TemporalType.DATE)
    private Date fechaDevolucion;

    @ManyToOne
    @JoinColumn(nullable = false)
    private LibroDomain libro;
```

```

@OneToOne
private UsuarioDomain usuario;
@OneToMany(mappedBy = "ejemplar")
private Set<PrestamoHistoricoDomain> prestamosHistoricos =
    new HashSet<PrestamoHistoricoDomain>();

private EjemplarDomain() {}

public EjemplarDomain(LibroDomain libro, String idEjemplar) {
    super();
    this.libro = libro;
    this.idEjemplar = idEjemplar;
    this.localizacion = Localizacion.SALA;
}

public Long getId() {
    return id;
}

// getters y setters

public EstadoEjemplar getEstado() {
    if (usuario == null)
        return EstadoEjemplar.DISPONIBLE;
    else
        return EstadoEjemplar.PRESTADO;
}

@Override
public int hashCode() {
    final int prime = 31;
    int result = 1;
    result = prime * result
        + ((idEjemplar == null) ? 0 : idEjemplar.hashCode());
    result = prime * result
        + ((libro == null) ? 0 : libro.hashCode());
    return result;
}

@Override
public boolean equals(Object obj) {
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (getClass() != obj.getClass())
        return false;
    EjemplarDomain other = (EjemplarDomain) obj;
    if (idEjemplar == null) {
        if (other.idEjemplar != null)
            return false;
    } else if (!idEjemplar.equals(other.idEjemplar))
        return false;
    if (libro == null) {
        if (other.libro != null)
            return false;
    } else if (!libro.equals(other.libro))
        return false;
    return true;
}
}

```

Es importante hacer notar lo siguiente:

- Añadimos la anotación `@Entity`, la anotación `@Table` para especificar el nombre de la

tabla y anotaciones `@Column` para el nombre de ciertas columnas. Añadimos también la anotación `@Id` para definir la clave primaria y las relaciones `@OneToMany` y `@ManyToOne` con la especificación de qué atributo se usa para la clave ajena (con `mappedBy`).

- Añadimos también un constructor vacío privado en todas las entidades, necesario para Hibernate.
- Definimos el mapeado del atributo enumerado `localización` con `columnDefinition="enum( 'SALA' , 'CASA' , 'DEPARTAMENTO' )"`. Esto genera esa definición en la base de datos, para que el valor de la columna pueda ser únicamente SALA, CASA O DEPARTAMENTO. Un inconveniente de esta definición es que no es estándar y sólo se puede utilizar en MySQL y otras pocas bases de datos.

Debemos añadir esta definición en todos los atributos enumerados.

- Utilizamos el modificador `nullable = false` en todos los campos que queremos mapear a columnas NOT NULL. Es recomendable definir en la entidad un constructor que contenga todos los campos definidos como no nulos. De esta forma se obliga a proporcionar esos valores en su creación. En este caso se inicializa a SALA en el constructor.
- Utilizamos el modificador `unique = true` en todos los campos que queremos mapear a columnas UNIQUE. Es recomendable hacerlo en aquellos campos por los que después se va a realizar una búsqueda, a parte de la clave primaria. Por ejemplo, el ISBN en un libro o el correo electrónico en un usuario.
- Es destacable el método `getEstado()` que devuelve el estado de un ejemplar. El ejemplar no tiene ningún atributo que defina su estado, sino que se devolverá DISPONIBLE O PRESTADO dependiendo de si tiene algún usuario asignado o no.

Clase UsuarioDomain:

```
package es.ua.jtech.jbib.model;

// imports

@Entity
@Table(name = "usuario")
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "tipo",
                      discriminatorType=DiscriminatorType.STRING)
@DiscriminatorOptions(force = true)
public abstract class UsuarioDomain extends DomainObject {
    private static final long serialVersionUID = 1L;

    @Id @GeneratedValue
    private Long id;

    @Column(unique=true)
    private String login;
    private String password;
    @Column(unique=true)
    private String email;
    private String nombre;
    private String apellido1;
    private String apellido2;
    @Embedded
```

```

private Direccion direccion;

@Enumerated(EnumType.STRING)
@Column(columnDefinition="enum('ACTIVO', 'MOROSO', 'MULTADO')",
        nullable=false)
private EstadoUsuario estado;

@OneToMany(mappedBy="usuario")
private Set<EjemplarDomain> prestamos = new HashSet<EjemplarDomain>();
@OneToMany(mappedBy="usuario")
private Set<ReservaDomain> reservas = new HashSet<ReservaDomain>();
@OneToMany(mappedBy="usuario")
private Set<PrestamoHistoricoDomain> prestamosHistoricos =
        new HashSet<PrestamoHistoricoDomain>();
@OneToMany(mappedBy="usuario")
private Set<MultadaDomain> multas = new HashSet<MultadaDomain>();

// getters, setters, hashCode y equals
}

```

Además de las anotaciones similares a las de `Ejemplar` debemos añadir las anotaciones que mapean la relación de herencia con la anotación `@Inheritance`. Usamos la estrategia de tabla única. La columna discriminante la definimos con la anotación `@DiscriminatorColumn`. Definimos que su nombre es `tipo` y que los valores en la columna van ser `STRING`.

La anotación `@DiscriminatorOptions(force = true)` es necesaria para que Hibernate pueda construir correctamente objetos de las clases hijas a partir de consultas realizadas sobre la clase padre. En nuestro caso, definiremos una consulta `UsuarioDomain.findAll` que va devolver todos los objetos `AlumnoDomain` y `ProfesorDomain` existentes en la base de datos.

Clase `AlumnoDomain`:

```

package es.ua.jtech.jbib.model;

// imports

@Entity
@DiscriminatorValue(value = "ALUMNO")
public class AlumnoDomain extends UsuarioDomain {

    static final long serialVersionUID = 1L;
    String tutor;

    private AlumnoDomain() {}

    public AlumnoDomain(String login, String password) {
        super();
        this.setLogin(login);
        this.setPassword(password);
        this.setEstado(EstadoUsuario.ACTIVO);
    }

    public String getTutor() {
        return tutor;
    }

    public void setTutor(String tutor) {
        this.tutor = tutor;
    }
}

```

```
    }
```

Clase ProfesorDomain:

```
package es.ua.jtech.jbib.model;

// imports

@Entity
@DiscriminatorValue(value = "PROFESOR")
public class ProfesorDomain extends UsuarioDomain {
    private static final long serialVersionUID = 1L;
    private String departamento;

    private ProfesorDomain {}

    public ProfesorDomain(String login, String password) {
        super();
        this.setLogin(login);
        this.setPassword(password);
        this.setEstado(EstadoUsuario.ACTIVO);
    }

    public String getDepartamento() {
        return departamento;
    }

    public void setDepartamento(String departamento) {
        this.departamento = departamento;
    }
}
```

Lo único que hay que definir en las clases hijas es la anotación `@DiscriminatorValue` que define el valor de la columna discriminante que define objetos de cada entidad.

Escribe el resto de entidades, hasta que no haya ningún error al lanzar el test con `$mvn test`

#### 2.1.4. Validación del esquema resultante

Vamos a usar el esquema SQL anterior para validar el mapeo. Esta validación constituye un test de integración, en el que se comprueba que las entidades desarrolladas van a mapearse bien con la base de datos que tenemos en producción. Para ello hacemos lo siguiente:

1. Creamos la base de datos a partir del esquema SQL anterior. Para ello borramos la base de datos `biblioteca` con el administrador de MySQL y copiamos el código SQL anterior en un fichero, por ejemplo `biblioteca.sql`. Para crear la base de datos `biblioteca` ejecutamos los comandos SQL:

```
$ mysql -u root -p < biblioteca.sql
```

2. Copiamos el fichero `persistence.xml` a la ruta `src/main/resources/META-INF/persistence.xml` y cambiamos el valor de `hibernate.hbm2ddl.auto` a `validate`.

3. Creamos un programa en el paquete es.ua.jtech.main en el ámbito principal de Maven que cargue esta unidad de persistencia y realice la validación.

En src/main/java:

```
package es.ua.jtech.jbib.main;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

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

public class BibliotecaMain {
    private static Log logger = LogFactory.getLog(BibliotecaMain.class);

    public static void main(String[] args) {
        logger.info("Building JPA EntityManagerFactory for DB validation");
        EntityManagerFactory emf =
Persistence.createEntityManagerFactory("biblioteca");
        EntityManager em = emf.createEntityManager();
        em.close();
        emf.close();
        logger.info("JPA EntityManagerFactory closed");
    }
}
```

4. Ejecutamos el programa desde Eclipse o desde Maven con:

```
$ mvn exec:java -Dexec.mainClass="es.ua.jtech.jbib.main.BibliotecaMain"
```

Comprobamos que la validación es correcta y no hay ningún error.

## 2.1.5. Clases DAO

Una vez construidas las entidades debemos crear las clases DAO que abstraerán las operaciones comunes CRUD y las consultas. Las debemos crear en el paquete es.ua.jtech.jbib.persistence. Nos basamos en el patrón visto en la sesión correspondiente de JPA. Creamos las siguientes clases:

- Dao
- BibliotecarioDao
- EjemplarDao
- LibroDao
- MultaDao
- PrestamoHistoricoDao
- ReservaDao
- UsuarioDao

Definiremos las funciones de creación, actualización y borrado, y también una consulta para cada una de las entidades que devuelve todas las instancias (hay que añadirla en cada una de las clases de dominio). Más adelante incluiremos más consultas.

Listamos a continuación la clase genérica Dao, ligeramente modificada con respecto a la vista en la sesión de JPA.

```

package es.ua.jtech.jbib.persistence;

import javax.persistence.EntityExistsException;
import javax.persistence.EntityManager;
import javax.persistence.TransactionRequiredException;

abstract class Dao<T, K> {
    EntityManager em;

    /**
     * Constructor del DAO. Se pasa como parámetro un entity manager que va
     * a utilizar en todas sus operaciones. El entity manager debe tener
     * abierta
     * una transacción. Quien llama al DAO es el responsable de abrir y
     * cerrar la
     * transacción y el entity manager.
     *
     * @param em entity manager
     * @throws TransactionRequiredException cuando se llama al constructor
     * del
     *         DAO sin que se haya abierto una transacción en el entity
     *         manager
     */
    public Dao(EntityManager em) {
        if (!em.getTransaction().isActive())
            throw new TransactionRequiredException(
                "Es necesaria una transacción activa para construir el
DAO");
        this.em = em;
    }

    /**
     * @return el entity manager asociado al DAO
     */
    public EntityManager getEntityManager() {
        return this.em;
    }

    /**
     * Devuelve la entidad correspondiente a una clave primaria.
     *
     * @param id clave primaria
     * @return la entidad encontrada o null si la entidad no existe
     */
    public abstract T find(K id);

    /**
     * Hace persistente una entidad, obteniéndose su clave primaria en el
     * caso de
     * que ésta sea generada. La entidad no debe estar gestionada por el
     * entity
     * manager.
     *
     * @param t entidad que se quiere hacer persistente
     * @return entidad persistente gestionada por el entity manager, con su
     * clave
     *         primaria generada
     * @throws DaoException si la entidad que se pasa ya está creada
     * @throws IllegalArgumentException si el objeto que se pasa no es una

```

```

        *
        * entidad
    */
public T create(T t) {
    try {
        em.persist(t);
        em.flush();
        em.refresh(t);
        return t;
    } catch (EntityExistsException ex) {
        throw new DaoException("La entidad ya existe", ex);
    }
}

/**
 * Actualiza los valores de una entidad en el gestor de persistencia.
La
 * entidad debe tener la clave primaria y puede no estar gestionada por
el
 * entity manager.
*
* @param t la entidad persistente que se quiere actualizar
* @return la entidad persistente gestionada
* @throws IllegalArgumentException Si el objeto que se pasa no es una
*         entidad.
*/
public T update(T t) {
    return (T) em.merge(t);
}

/**
 * Elimina la entidad persistente que se pasa como parámetro. Debe
tener la
 * clave primaria y puede no estar gestionada por el entity manager.
*
* @param t la entidad persistente que se quiere eliminar
* @throws IllegalArgumentException Si el objeto que se pasa no es una
*         entidad
*/
public void delete(T t) {
    t = em.merge(t);
    em.remove(t);
}
}

```

Cambios con respecto a la sesión de JPA:

- Se comprueba que existe una transacción abierta en el *entity manager* al crear el DAO.
- Se documentan las excepciones de tipo Runtime que arrojan los métodos.
- Se crea una excepción de tipo *DaoException* que envuelve la excepción de JPA *EntityExistsException*.

Listamos también las clases *EjemplarDao* y *DaoException*:

```

package es.ua.jtech.jbib.persistence;

import java.util.List;
import javax.persistence.EntityManager;
import es.ua.jtech.jbib.model.BibliotecarioDomain;

public class BibliotecarioDao extends Dao<BibliotecarioDomain, Long> {

```

```
public BibliotecarioDao(EntityManager em) {
    super(em);
}

@Override
public BibliotecarioDomain find(Long id) {
    EntityManager em = this.getEntityManager();
    return em.find(BibliotecarioDomain.class, id);
}

/**
 * @return una lista con todas las entidades Bibliotecario existentes
 */
@SuppressWarnings("unchecked")
public List<BibliotecarioDomain> getAllBibliotecarios() {
    EntityManager em = this.getEntityManager();
    // getResultList puede devolver una QueryTimeoutException,
    // pero no la vamos a tratar, porque se produce en una situación
    // excepcional no controlable
    return (List<BibliotecarioDomain>) em
        .createNamedQuery("Bibliotecario.findAll").getResultList();
}
}
```

La clase DaoException:

```
package es.ua.jtech.jbib.persistence;

public class DaoException extends RuntimeException {
    private static final long serialVersionUID = 1L;

    public DaoException(String mensaje) {
        super(mensaje);
    }

    public DaoException(String mensaje, Throwable causa) {
        super(mensaje, causa);
    }
}
```

### 2.1.5.1. Tests de los DAO con DbUnit

Vamos por último a escribir los tests que comprueban los DAOs desarrollando utilizando DbUnit para poblar la base de datos en el ámbito de test de Maven.

Añadimos la dependencia en el POM del proyecto jbib-modelo:

```
<!-- DbUnit -->
<dependency>
    <groupId>org.dbunit</groupId>
    <artifactId>dbunit</artifactId>
    <version>2.4.9</version>
    <scope>test</scope>
</dependency>
```

Creamos una primera versión de la clase de test es.ua.jtech.jbib.persistence.DaoTest en src/test/java/ en la que comprobamos cómo cargar la conexión de DbUnit:

```

package es.ua.jtech.jbib.persistence;

// imports

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

    @BeforeClass
    public static void initDatabaseTest() {
        try {
            emf = Persistence.createEntityManagerFactory("biblioteca");
            Class.forName("com.mysql.jdbc.Driver");
            Connection jdbcConnection = (Connection) DriverManager
                .getConnection(
                    "jdbc:mysql://localhost:3306/biblioteca",
                    "root", "expertojava");
            connection = new DatabaseConnection(jdbcConnection);

            FlatXmlDataSetBuilder flatXmlDataSetBuilder = new
FlatXmlDataSetBuilder();
            flatXmlDataSetBuilder.setColumnSensing(true);
            dataset = flatXmlDataSetBuilder.build(Thread.currentThread()
                .getContextClassLoader()
                .getResourceAsStream("test-dao-dataset.xml"));
        } catch (Exception ex) {
            ex.printStackTrace();
            fail("Exception during JPA EntityManager instantiation.");
        }
    }

    @AfterClass
    public static void closeEntityManagerFactory() throws Exception {
        DatabaseOperation.DELETE_ALL.execute(connection, dataset);
        if (emf != null)
            emf.close();
    }

    @Before
    public void cleanDB() throws Exception {
        DatabaseOperation.CLEAN_INSERT.execute(connection, dataset);
    }

    @Test
    public void bibliotecarioDaoCreateTest() {
        EntityManager em = emf.createEntityManager();
        em.getTransaction().begin();

        // Obtenemos el BibliotecarioDao
        BibliotecarioDao bibliotecarioDao = new BibliotecarioDao(em);

        // Creamos un nuevo bibliotecario
        BibliotecarioDomain bibliotecario = new BibliotecarioDomain(
            "francisco.perez", "123456");
        bibliotecarioDao.create(bibliotecario);
        assertNotNull(bibliotecario.getId());

        // Guardamos su id y cerramos la transacción y el em
        Long id = bibliotecario.getId();
        em.getTransaction().commit();
        em.close();

        // Abrimos otro em para comprobar que el bibliotecario
        // está en la BD
    }
}

```

```
        em = emf.createEntityManager();
        bibliotecarioDao = new BibliotecarioDao(em);
        BibliotecarioDomain bib2 = bibliotecarioDao.find(id);
        assertNotNull(bib2);
        assertTrue(bib2.getLogin().equals(bibliotecario.getLogin()))
            && bib2.getPassword().equals(bibliotecario.getPassword()));
        em.close();
    }
}
```

Algunos comentarios:

- La anotación `@BeforeClass` hace que el método `initDatabaseTest()` se ejecute una sola vez antes de todos los tests.
- Este método `initDatabaseTest` obtiene la factoría de entity managers, el driver MySQL, la conexión a la base de datos y el dataset `test-dao-dataset.xml`
- El método `closeEntityManagerFactory()` se ejecuta al final de todos los tests, cierra la factoría de entity managers y borra todos los registros en las tablas del dataset
- El método `cleanDB()` tiene la anotación `@Before` que hace que se ejecute antes de cada uno de los tests. En él se inicializa la base de datos con los registros definidos en el dataset. De esta forma nos aseguramos que cada test comienza con el mismo estado de la base de datos.
- El test `bibliotecarioDaoCreateTest()` no obtiene ningún dato del dataset. Crea un bibliotecario nuevo en la base de datos usando el `BibliotecarioDao` y después comprueba que se ha insertado correctamente buscándolo por su clave primaria.

Creamos el fichero `src/test/resources/test-daos-dataset.xml` vacío:

```
<?xml version='1.0' encoding='UTF-8'?>
<dataset>
</dataset>
```

Lanzamos el test `DaoTest` y comprobamos que no hay ningún error.

Terminamos añadiendo algunos datos al dataset y añadiendo algunos tests al `DaoTest`:

```
<?xml version='1.0' encoding='UTF-8'?>
<dataset>
    <bibliotecario id="1" login="javier.huertas" password="11111"/>
    <bibliotecario id="2" login="francisco.martinez" password="22222"/>
    <usuario tipo="ALUMNO"/>
    <usuario tipo="ALUMNO"/>
    <usuario tipo="PROFESOR"/>
</dataset>
```

Tests:

```
// Test que busca el bibliotecario con id=1
@Test
public void bibliotecarioDaoFindTest() {
    EntityManager em = emf.createEntityManager();
    em.getTransaction().begin();
    BibliotecarioDao bibliotecarioDao = new
    BibliotecarioDao(em);
    BibliotecarioDomain bib2 = bibliotecarioDao.find(1L);
```

```

        assertNotNull(bib2);
        assertTrue(bib2.getLogin().equals("javier.huertas")
                   && bib2.getPassword().equals("11111"));
        em.getTransaction().commit();
        em.close();
    }

    // Test que comprueba que getAllBibliotecarios() devuelve 2 bibliotecarios
    @Test
    public void getAllBibliotecariosTest() {
        EntityManager em = emf.createEntityManager();
        em.getTransaction().begin();

        BibliotecarioDao bibliotecarioDao = new BibliotecarioDao(em);
        List<BibliotecarioDomain> lista =
        bibliotecarioDao.getAllBibliotecarios();
        assertTrue(lista.size() == 2);

        em.getTransaction().commit();
        em.close();
    }

    // Test que comprueba que getAllUsuarios devuelve 3 usuarios
    @Test
    public void getAllUsuariosTest() {
        EntityManager em = emf.createEntityManager();
        em.getTransaction().begin();

        UsuarioDao usuarioDao = new UsuarioDao(em);
        List<UsuarioDomain> listaUsuarios = usuarioDao.getAllUsuarios();
        assertTrue(listaUsuarios.size() == 3);

        em.getTransaction().commit();
        em.close();
    }
}

```

Añade registros en el dataset y tests que comprueben:

- Todos los métodos *getAll* del resto de los DAOs

### 2.1.6. Singleton PersistenceManager

Desde la capa de negocio de la aplicación vamos a necesitar obtener *entity managers*. Para eso añadimos el singleton `PersistenceManager` que proporciona esta utilidad.

En el método de creación del singleton obtenemos la factoría de entity managers y la guardamos en una variable estática. Definimos en el singleton un método `createEntityManager` que devuelve un nuevo entity manager obtenido a partir de esta factoría. Definimos también el método `setEntityManagerFactory` para dar flexibilidad a la clase y poder *inyectar* en ella una nueva factoría de entity managers. De esta forma podríamos cambiar en tiempo de ejecución la unidad de persistencia con la que trabaja la aplicación.

```

package es.ua.jtech.jbib.persistence;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;

```

```
import javax.persistence.Persistence;

public class PersistenceManager {
    static private final String PERSISTENCE_UNIT_NAME =
"biblioteca";
    protected static PersistenceManager me = null;
    private EntityManagerFactory emf = null;

    private PersistenceManager() {
    }

    public static PersistenceManager getInstance() {
        if (me == null) {
            me = new PersistenceManager();
        }
        return me;
    }

    public void setEntityManagerFactory(EntityManagerFactory myEmf)
{
    emf = myEmf;
}

    public EntityManager createEntityManager() {
        // Si el emf no se ha inicializado o se ha cerrado, se
vuelve a abrir
        if (emf == null) {
            emf = Persistence
                .createEntityManagerFactory(PERSISTENCE_UNIT_NAME);
            this.setEntityManagerFactory(emf);
        }
        return emf.createEntityManager();
    }

    public void close() {
        if (emf != null)
            emf.close();
        emf = null;
    }
}
```

Por último, crea el siguiente test en el directorio de tests que comprueba que el singleton funciona correctamente:

```
package es.ua.jtech.jbib.persistence;

import static org.junit.Assert.*;
import javax.persistence.EntityManager;
import org.junit.Test;

public class PersistenceManagerTest {
    @Test
    public void obtenerEntityManagerTest() {
        EntityManager em =
PersistenceManager.getInstance().createEntityManager();
        em.getTransaction().begin();
        assertTrue(em.isOpen());
        em.getTransaction().commit();
        em.close();
        assertFalse(em.isOpen());
    }
}
```

## 2.2. Capa de lógica de negocio

---

### 2.2.1. Introducción

---

Una vez mapeadas las entidades y creados los DAO con los que recuperarlas y actualizarlas crearemos en este apartado las clases con los métodos de servicio que permitan trabajar con la biblioteca:

- Operaciones: listar libros disponibles, listar libros reservados, reservar libros, ...
- CRUD de libros: creación, recuperación, actualización y borrado

Vamos ahora a definir las clases que implementan la lógica de negocio. Tal y como hemos comentado, estas clases de negocio son muy importantes, porque definen las funcionalidades que exportamos a otras capas, como la capa web. Los objetos que devuelven son objetos de dominio desconectados de la base de datos. Reciben como parámetro identificadores y datos elementales relacionados con la operación que se quiere realizar (un identificador de ejemplar, o un identificador de usuario por ejemplo). Las clases de negocio encapsulan toda la funcionalidad y de nuestra aplicación y proporcionan un *front-end* de la capa de persistencia.

Todos los métodos de las clases de negocio son transaccionales. Si el método termina correctamente estaremos seguros de que todas las actualizaciones se han realizado en la base de datos. Si algo falla en el método, se devolverá una excepción y se hará un rollback del estado de la base de datos, de forma que todo volverá al estado previo a la realización de la llamada al método.

La capa de negocio se implementará en un nuevo proyecto llamado `jbib-negocio` que dependerá del anterior `jbib-modelo` que es el que implementa la capa de persistencia.

Debido a la gran cantidad de código a desarrollar y al poco tiempo que tenemos para hacerlo, vamos a proporcionar un código mínimo y te vamos a pedir que amplíes muy pocas funcionalidades. Es importante que no sólo copies el código, sino que también lo leas y reflexiones sobre su funcionamiento y estructura. Más adelante iremos añadiendo nuevos métodos a la capa de negocio.

### 2.2.2. Proyecto jbib-negocio

---

Empezamos creando el nuevo proyecto `jbib-negocio` que va a contener todas las clases e interfaces relacionadas con la lógica de negocio.

En el espacio de trabajo del proyecto de integración crea un nuevo proyecto `jbib-negocio`. Escoge la opción de Eclipse *New > Project... > Maven > Maven Module*, marcando la opción de *Create simple project* e introduciendo:

- *Module Name:* `jbib-negocio`
- *Parent Project:* `proyint-jbib`

- *Name:* jbib-negocio

Añade las siguientes dependencias al POM, necesarias para que funcionen correctamente las llamadas a la capa de datos:

```
<dependencies>
    <dependency>
        <groupId>org.slf4j</groupId>
        <artifactId>slf4j-log4j12</artifactId>
        <version>1.6.1</version>
        <scope>runtime</scope>
    </dependency>
    <dependency>
        <groupId>es.ua.jtech.proyint</groupId>
        <artifactId>jbib-modelo</artifactId>
        <version>0.0.1-SNAPSHOT</version>
    </dependency>
</dependencies>
```

Para probar que se accede correctamente a la capa de persistencia desde la capa de negocio, copiamos el test `PersistenceManagerTest` en este proyecto, en el directorio `src/test/java`.

Ejecutamos el test desde Eclipse y desde línea de comandos ejecutamos todos los tests de todos los proyectos, desde el proyecto padre:

```
$ cd proyint-expertojava/jbib
$ mvn test
...
[INFO]
-----
[INFO] Reactor Summary:
[INFO]
[INFO] jbib ..... SUCCESS
[0.097s]
[INFO] jbib-modelo ..... SUCCESS
[6.499s]
[INFO] jbib-negocio ..... SUCCESS
[2.159s]
[INFO]
-----
[INFO] BUILD SUCCESS
[INFO]
```

### 2.2.3. Capa de lógica de negocio

Vamos ahora a definir las clases que implementan la lógica de negocio. Tal y como hemos comentado, estas clases de negocio son muy importantes, porque definen las funcionalidades que exportamos a otras capas, como la capa web. Los objetos que devuelven son objetos de dominio desconectados de la base de datos. Reciben como parámetro identificadores y datos elementales relacionados con la operación que se quiere realizar (un identificador de ejemplar, o un identificador de usuario por ejemplo). Las clases de negocio encapsulan toda la funcionalidad y de nuestra aplicación y

proporcionan un *front-end* de la capa de persistencia.

Todos los métodos de las clases de negocio son transaccionales. Si el método termina correctamente estaremos seguros de que todas las actualizaciones se han realizado en la base de datos. Si algo falla en el método, se devolverá una excepción y se hará un rollback del estado de la base de datos, de forma que todo volverá al estado previo a la realización de la llamada al método.

Utilizaremos el patrón *factoría* para definir las clases. Especificaremos las clases con interfaces y una factoría devolverá la implementación concreta. Esto nos permitirá en el futuro sustituir la implementación de la capa de negocio por otra que, por ejemplo, utilice EJBs.

En cuanto a notación, llamaremos a las clases de negocio con un nombre que agrupe los servicios que ofrecen y el *Service*:

- OperacionService: va a permitir realizar operaciones como solicitar préstamo por un usuario, solicitar una reserva, etc.
- LibroService: CRUD sobre libros y ejemplares, listados de consultas
- UsuarioService: operaciones para recuperar y consultar usuarios

#### 2.2.4. Interfaces de las clases de negocio

A continuación listamos las interfaces de las clases de negocio. Definen las funcionalidades que se ofrece a otras capas de la aplicación y que debemos implementar con las clases de negocio. Estas funcionalidades son las mínimas definidas. Más adelante añadiremos más funcionalidades conforme las vayamos necesitando.

Definimos todas las clases y todas las excepciones necesarias en el paquete es.ua.jtech.jbib.service. Todas las excepciones las debes crear de tipo RuntimeException.

Interfaz IOperacionService:

```
package es.ua.jtech.jbib.service;

import es.ua.jtech.jbib.model.EjemplarDomain;

public interface IOperacionService {
    /**
     * Realiza la petición de un préstamo de un usuario sobre un ejemplar.
     * El usuario tiene que ser un profesor o un alumno activo que no tiene
     * cubierto el cupo de préstamos. El ejemplar debe estar disponible.
     *
     * @param usuarioId el id del usuario
     * @param ejemplarId el id del ejemplar
     * @return el ejemplar con los datos del préstamo: usuario y fecha de
     *         devolución
     * @throws OperacionServiceException en las siguientes condiciones:
     *         <ul>
     *         <li>el usuario o el ejemplar no existen</li>
```

```
*             <li>el usuario no está activo</li>
*             <li>el usuario tiene completo el cupo de
préstamos</li>
*             <li>el ejemplar no está disponible</li>
*
* /
EjemplarDomain pidePrestamo(Long usuarioId, Long ejemplarId);
}
```

Vemos que la interfaz define el método `pidePrestamo(usuarioId, ejemplarId)` con el que se realiza un préstamo de un ejemplar a un usuario. La operación tiene un conjunto de condiciones que se deben cumplir para que funcione correctamente. En el caso en que alguna condición no se satisfaga, se lanza una excepción de tipo `OperacionServiceException`. Se trata de una excepción runtime, por lo que no hay que declararla con la cláusula `throws` en el perfil del método, pero sí que hay que documentarla.

La excepción `OperacionServiceException` es hija de `BibliotecaException` y se define de la siguiente forma:

```
package es.ua.jtech.jbib.service;

import es.ua.jtech.jbib.BibliotecaException;

public class OperacionServiceException extends BibliotecaException {

    private static final long serialVersionUID = 1L;

    public OperacionServiceException() {
        super();
    }

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

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

Vemos el resto de interfaces y los métodos que definen:

Interfaz `ILibroService`:

```
package es.ua.jtech.jbib.service;

import java.util.List;

import es.ua.jtech.jbib.model.EjemplarDomain;
import es.ua.jtech.jbib.model.LibroDomain;

public interface ILibroService {
    List<LibroDomain> listaLibros();

    /**
     * Recupera un libro de la biblioteca por su identificador. Junto con
     * el
     *     * libro se recuperan también sus ejemplares. Si el libro no existe
     * devuelve
}
```

```

    * null.
    *
    * @param libroId el id del libro a recuperar
    * @return el libro o null
    */
LibroDomain recuperaLibro(Long libroId);

/**
 * Recupera un ejemplar de un libro de la biblioteca por su
identificador.
 *
 * @param ejemplarId el id del ejemplar a recuperar
 * @return el libro o null
 */
EjemplarDomain recuperaEjemplar(Long ejemplarId);

/**
 * Recupera un libro de la biblioteca por su ISBN. En el libro se
recuperan
 * también sus ejemplares. Si el libro no existe devuelve null.
 *
 * @param isbn el ISBN del libro a recuperar
 * @return el libro o null
 */
LibroDomain buscaLibroPorIsbn(String isbn);

/**
 * Busca todos los libros cuyo autor+título contengan una palabra.
 *
 * @param keyword palabra a buscar
 * @return lista con todos los libros que contienen la palabra
 */
public List<LibroDomain> buscaLibros(String keyword);

/**
 * Devuelve una lista de todos los libros de la biblioteca, ordenados
por
 * título.
 *
 * @return la lista de libros
 */
public List<LibroDomain> listaLibrosPorTitulo();

/**
 * Devuelve una lista paginada de los libros de la biblioteca ordenados
por
 * título, empezando por el número firstResult (siendo 0 el primero).
Devuelve los
 * maxResults siguientes.
 *
 * @param firstResult
 * @param maxResults
 * @return la lista de libros
 */
public List<LibroDomain> listaLibrosPorTitulo(int firstResult,
                                              int maxResults);
}

```

## Interfaz IUsuarioService:

```

package es.ua.jtech.jbib.service;
import es.ua.jtech.jbib.model.UsuarioDomain;
public interface IUsuarioService {

```

```
    /**
     * Recupera un usuario de la biblioteca por su identificador. Junto con
el
     * usuario se recupera también su colección de libros prestados y
reservados.
     * Si el usuario no existe devuelve null.
     *
     * @param usuarioId el id del usuario a recuperar
     * @return el usuario o null
     */
    UsuarioDomain recuperaUsuario(Long idUsuario);

    /**
     * Busca un usuario de la biblioteca por su login. Junto con el usuario
se
     * recupera también su colección de libros prestados y reservados. Si
el usuario no
     * existe devuelve null.
     *
     * @param login el login del usuario a recuperar
     * @return el usuario o null
     */
    UsuarioDomain buscaUsuarioPorLogin(String login);
}
```

## 2.2.5. Factoría de clases de negocio

La factoría de las clases de negocio permite obtener las implementaciones concretas `LibroService`, `OperacionService` y `UsuarioService` (todavía sin implementar):

```
package es.ua.jtech.jbib.service;

public class FactoriaServices {

    public static FactoriaServices me;

    protected FactoriaServices() { }

    public static FactoriaServices getInstance() {
        if (me == null) {
            me = new FactoriaServices();
        }
        return me;
    }

    public ILibroService getLibroService() {
        return new LibroService();
    }

    public IOperacionService getOperacionService() {
        return new OperacionService();
    }

    public IUsuarioService getUsuarioService() {
        return new UsuarioService();
    }
}
```

## 2.2.6. Consultas

Para implementar las funcionalidades de la capa de negocio, debemos ampliar los DAO del proyecto jbib-modelo con los siguientes métodos. También debemos añadir las consultas JPA asociadas en las clases entidad:

LibroDao:

- LibroDomain findByIsbn(String isbn): Devuelve el libroDomain correspondiente al isbn que se le pasa como parámetro
- List<LibroDomain> findAllLibrosByTitulo(): Devuelve todos los libros ordenados por título
- List<LibroDomain> buscaLibros(String palabra): Devuelve todos los libros que contienen la palabra en el título o autor

UsuarioDao:

- UsuarioDomain findByLogin(String login): Devuelve el usuarioDomain correspondiente al login que se le pasa como parámetro

## 2.2.7. Preparación de las pruebas con DbUnit

---

Utiliza también DbUnit para probar la capa de servicios.

Puedes usar como base este fichero de prueba que debes colocar en src/test/resources:

Fichero src/test/resources/test-service-dataset.xml

```
<?xml version='1.0' encoding='UTF-8'?>
<dataset>
    <!-- Bibliotecarios -->
    <bibliotecario id="1" login="javier.huertas" password="11111"/>

    <!-- Usuarios -->
    <usuario tipo="PROFESOR" id="1" login="aitor" password="aitor"
        email="a@gmail.com" nombre="Aitor" apellido1="Medrano"
        apellido2="Escrig"
        departamento="Informatica" estado="ACTIVO" calle="Bartolome"
        numero="11" piso="5 B" ciudad="Helsinki" codigoPostal="12345" />
    <usuario tipo="ALUMNO" id="2" login="juan" password="antonio"
        email="juan@decepcion.es" nombre="Juan Antonio" apellido1="Perez"
        apellido2="Perez" tutor="Aitor Medrano" estado="ACTIVO"
        calle="Almassera"
        numero="15" piso="1" ciudad="El Verger" codigoPostal="03770" />

    <!-- Libros -->
    <libro id="1" isbn="0131401572" titulo="Data Access Patterns"
        autor="Clifton Nock" numPaginas="512" fechaAlta="2007-02-22"
        numDisponibles="2" />
    <libro id="2" isbn="0321127420"
        titulo="Patterns Of Enterprise Application Architecture"
        autor="Martin Fowler"
        numPaginas="533" fechaAlta="2008-10-23" numDisponibles="1" />
    <libro id="3" isbn="0321180860" titulo="Understanding SOA with Web
        Services"
        autor="Eric Newcomer and Greg Lomow" numPaginas="465"
        fechaAlta="2007-01-25"
```

```

        numDisponibles="2" />

        <!-- Ejemplares de Data Access Patterns -->
        <ejemplar id="1" idEjemplar="001" fechaAdquisicion="2007-02-22"
            libro_id="1" localizacion="SALA" />
        <ejemplar id="2" idEjemplar="002" fechaAdquisicion="2007-02-22"
            libro_id="1" localizacion="SALA" />

        <!-- Ejemplares de Patterns Of Enterprise Application Architecture -->
        <!-- Prestado a Aitor Medrano -->
        <ejemplar id="3" idEjemplar="001" fechaAdquisicion="2008-10-23"
            libro_id="2" localizacion="DEPARTAMENTO" usuario_id="1" />
        <!-- Disponible -->
        <ejemplar id="4" idEjemplar="002" fechaAdquisicion="2008-10-23"
            libro_id="2" localizacion="SALA" />

        <!-- Ejemplares de Understanding SOA with Web Services -->
        <ejemplar id="5" idEjemplar="001" fechaAdquisicion="2007-01-25"
            libro_id="3" localizacion="SALA" />
        <ejemplar id="6" idEjemplar="002" fechaAdquisicion="2007-01-25"
            libro_id="3" localizacion="SALA" />

        <!-- Insertamos un elemento de cada una de las tablas restantes
            para asegurarnos de que la base de datos se inicializa
            correctamente (DBUnit
            sólo borra las tablas que aparecen en el dataset) -->

        <reserva/>
        <historico/>
        <multa/>

</dataset>
```

Define en `src/test/java/` el test `es.ua.jtech.jbib.service.ServiceTest.java` en el que se cargue el anterior dataset. Para ello define los métodos `initDatabaseTest()`, `closeEntityManagerFactory()` y `cleanDB()` como hicimos en el proyecto `jbib-modelo`.

Añade algunos tests que comprueben los métodos de negocio. Por ejemplo el siguiente comprueba que el listado de todos los libros debe tener tres elementos. Obtenemos el `libroService` a partir de la factoría y llamamos a su método `listaLibros`:

```

    @Test
    public void listaLibrosTest() {
        ILibroService libroService =
FactoriaServices.getInstance().getLibroService();
        List<LibroDomain> libros = libroService.listaLibros();
        assertTrue(libros.size()==3);
    }
}
```

El siguiente test comprueba un préstamo de un ejemplar:

```

    @Test
    public void operacionServicePidePrestamoTest() {
        IOperacionService operacionService = FactoriaServices.getInstance()
                                            .getOperacionService();
        ILibroService libroService =
FactoriaServices.getInstance().getLibroService();
        Long ejemplarId = 5L;
        Long usuarioId = 2L;
        operacionService.pidePrestamo(usuarioId, ejemplarId);
```

```

        EjemplarDomain ejemplar = libroService.recuperaEjemplar(ejemplarId);
        LibroDomain libro =
libroService.recuperaLibro(ejemplar.getLibro().getId());
        assertTrue(libro.getNumDisponibles().equals(1));
        assertTrue(ejemplar.getUsuario().getId().equals(usuarioId));
    }
}

```

Ahora el trabajo que falta es implementar los servicios para que estos tests funcionen correctamente.

## 2.2.8. Implementación de los servicios

Terminamos con un ejemplo de implementación de un método de `LibroService`:

```

package es.ua.jtech.jbib.service;

// imports

public class LibroService implements ILibroService {

    @Override
    public List<LibroDomain> listaLibros() {
        EntityManager em = PersistenceManager.getInstance()
            .createEntityManager();
        em.getTransaction().begin();

        LibroDao libroDao = new LibroDao(em);
        List<LibroDomain> lista = libroDao.getAllLibros();

        em.getTransaction().commit();
        em.close();
        return lista;
    }

    @Override
    public LibroDomain recuperalibro(Long libroId) {
        EntityManager em = PersistenceManager.getInstance()
            .createEntityManager();
        em.getTransaction().begin();

        LibroDao libroDao = new LibroDao(em);
        LibroDomain libro = libroDao.find(libroId);
        if (libro != null) {
            // accedemos a la lista de ejemplares para cargarlos en memoria
            libro.getEjemplares().size();
        }
        em.getTransaction().commit();
        em.close();
        return libro;
    }

    @Override
    public LibroDomain buscaLibroPorIsbn(String isbn) {
        // TODO Auto-generated method stub
        return null;
    }

    @Override
    public List<LibroDomain> buscaLibros(String keyword) {
        // TODO Auto-generated method stub
        return null;
    }
}

```

```
@Override  
public List<LibroDomain> listaLibrosPorTitulo() {  
    // TODO Auto-generated method stub  
    return null;  
}  
  
@Override  
public List<LibroDomain> listaLibrosPorTitulo(int firstResult,  
    int maxResults) {  
    // TODO Auto-generated method stub  
    return null;  
}  
  
@Override  
public EjemplarDomain recuperaEjemplar(Long ejemplarId) {  
    // TODO Auto-generated method stub  
    return null;  
}
```

Algunos comentarios sobre los métodos en la capa de negocio:

- Gestionamos correctamente la transacción y el entity manager: cerramos siempre el entity manager poniéndolo en el *finally* y hacemos un *rollback* de la transacción si hay algún error
- En todos los métodos comprobamos que se cumplen las precondiciones y lanzamos una excepción en el caso en que no sea así
- Los argumentos que se toman como parámetros son datos sencillos (el identificador de un usuario y el isbn de un libro, por ejemplo) y se devuelven objetos de dominio desconectados del contexto de persistencia
- Hay que tener cuidado en cargar en memoria todos los objetos que vayan a necesitar los que llamen al método. Los atributos *lazy* del objeto devuelto no van a estar disponibles, porque el objeto está desconectado del entity manager.

Y también listamos la clase OperacionService:

```
package es.ua.jtech.jbib.service;  
  
// imports  
  
public class OperacionService implements IOperacionService {  
    private Log logger = LogFactory.getLog(OperacionService.class);  
  
    @Override  
    public EjemplarDomain pidePrestamo(Long usuarioId, Long ejemplarId) {  
        EntityManager em = PersistenceManager.getInstance()  
            .createEntityManager();  
        EntityTransaction tx = em.getTransaction();  
  
        try {  
            tx.begin();  
            UsuarioDao usuarioDao = new UsuarioDao(em);  
            LibroDao libroDao = new LibroDao(em);  
            EjemplarDao ejemplarDao = new EjemplarDao(em);  
  
            UsuarioDomain usuario = usuarioDao.find(usuarioId);  
            EjemplarDomain ejemplar = ejemplarDao.find(ejemplarId);  
        } catch (Exception e) {  
            logger.error("Error al realizar el préstamo: " + e.getMessage());  
            tx.rollback();  
        } finally {  
            em.close();  
        }  
    }  
}
```



## 2.2.9. Javadoc

Es muy sencillo crear la documentación Javadoc del proyecto utilizando el plugin javadoc de Maven:

```
$ cd jbib
$ mvn javadoc:javadoc
```

La documentación javadoc se genera en el directorio *target/site/apidocs* de cada proyecto:



## 2.3. Resumen

Además de construir los proyectos *jbib-modelo* y *jbib-negocio* con las clases anteriores, debes realizar lo siguiente:

- Terminar de implementar la clase `LibroService` con los métodos definidos en la interfaz
- Implementar la clase `UsuarioService` con los métodos definidos en la interfaz
- Comprobar el funcionamiento correcto de las clases Service con un conjunto de tests

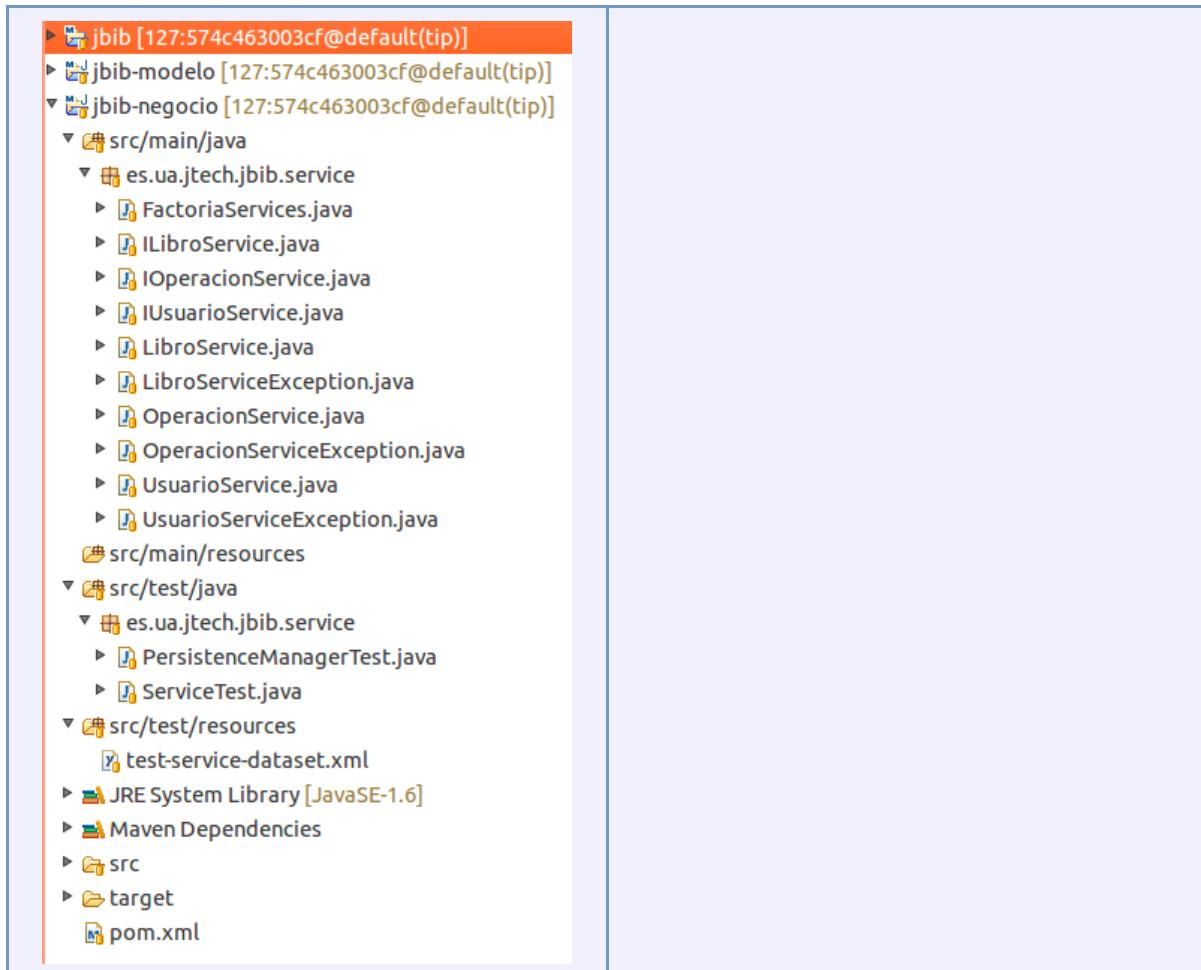
Las siguientes imágenes muestran todos los archivos que debe contener el proyecto Eclipse *jbib-modelo*:

```

jbib-modelo [127:574c463003cf@default(tip)]
└── src
    ├── main
    │   ├── java
    │   │   ├── es.ua.jtech.jbib
    │   │   └── es.ua.jtech.jbib.main
    │   │       ├── BibliotecaMain.java
    │   │       └── ...
    │   ├── resources
    │   └── ...
    └── test
        ├── java
        │   ├── es.ua.jtech.jbib
        │   └── es.ua.jtech.jbib.model
        │       ├── BibliotecarioDomainTest.java
        │       ├── EjemplarDomainTest.java
        │       ├── LibroDomainTest.java
        │       ├── MultaDomainTest.java
        │       ├── PrestamoHistoricoDomainTest.java
        │       ├── ReservaDomainTest.java
        │       └── UsuarioDomainTest.java
        └── resources
            ├── META-INF
            │   ├── persistence.xml
            │   ├── commons-logging.properties
            │   ├── log4j.properties
            │   └── test-dao-dataset.xml
            └── ...

```

Y las siguientes muestran el proyecto *jbib-negocio*:



Para la entrega se deberá etiquetar el repositorio bitbucket con el tag *entrega-proyint-negocio*.

## 3. Capa web con servlets y JSPs

### 3.1. Introducción

En esta sesión de integración construiremos sobre la capa de negocio desarrollada en sesiones anteriores una aplicación web dinámica que, mediante servlets y JSPs que utilicen estas clases previas, expondrá una interfaz web que dará acceso a varias de las funcionalidades de la biblioteca.

### 3.2. Proyecto jbib-web

Vamos a crear en primer lugar un nuevo proyecto con Maven, de nombre `jbib-web`, utilizando para ello el arquetipo `webapp-javaee6`, para tenerlo configurado directamente con la versión de Java EE que vamos a utilizar.

Para que Eclipse reconozca un proyecto web Maven como proyecto web dinámico de Eclipse WTP será necesario instalar el plugin *Maven Integration for Eclipse WTP*. De no hacer esto, Eclipse no reconocería el proyecto como proyecto web, y por lo tanto no podríamos ejecutar la aplicación directamente desde este entorno. Podemos ir a *Help > About Eclipse > Installation Details* para comprobar si tenemos instalada la extensión *Maven Integration for Eclipse WTP (m2e-wtp)* del plugin de Maven. Si no es así, procederemos a instalarla como se muestra a continuación:

- Entramos en *Help > Eclipse Marketplace ...*
- Introducimos el texto "*maven*" en el cuadro de búsqueda.
- Buscamos entre los resultados de la búsqueda el plugin *Maven Integration for Eclipse WTP (Incubation)* y pulsamos sobre el botón *Install*.
- Pulsamos el botón *Next* y seguimos las instrucciones del proceso de instalación.

Ahora ya podemos crear el nuevo proyecto `jbib-web` que va a contener todos los componentes de la capa web.

En el espacio de trabajo del proyecto de integración crea un nuevo proyecto `jbib-web`. Escoge la opción de Eclipse *New > Project... > Maven > Maven Module*, sin marcar la opción de *Create simple project* e introduciendo:

- *Module Name:* `jbib-web`
- *Paquete:* `es.ua.jtech.jbib.web`
- *Parent Project:* `jbib`
- *Archetype:* `webapp-javaee6` (se indica en la segunda pantalla del asistente)

En el editor del POM, añade la dependencia con el proyecto `jbib-negocio` y con `JSTL 1.2`:

```
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>jstl</artifactId>
```

```
<version>1.2</version>
</dependency>

<dependency>
    <groupId>es.ua.jtech.proyint</groupId>
    <artifactId>jbib-negocio</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <type>jar</type>
    <scope>compile</scope>
</dependency>
```

Podemos eliminar del POM el bloque `<build> ... </build>` ya que no es necesario. Deberemos también eliminar las etiquetas `groupId` y `version` de nuestro proyecto para que coja los valores del proyecto padre. Deberá quedar como se muestra a continuación:

```
<parent>
    <artifactId>jbib</artifactId>
    <groupId>es.ua.jtech.proyint</groupId>
    <version>0.0.1-SNAPSHOT</version>
</parent>

<artifactId>jbib-web</artifactId>
<packaging>war</packaging>

<name>jbib-web</name>
```

El fichero completo debe quedar tal como se indica en este [pom.xml](#).

#### Nota

Es posible que en algunas ocasiones Eclipse no reconozca correctamente los módulos de un multiproyecto Maven. Para solucionar este problema eliminaremos todos los proyectos Maven (sin borrarlos del disco), y los volveremos a importar con *File > Import ... > Maven > Existing Maven Projects*, seleccionando únicamente el proyecto padre. Eclipse importará de forma automática todos los módulos que contiene.

Ahora podemos ejecutar nuestra aplicación web directamente desde Eclipse como hemos hecho a lo largo del curso (*Run As > Run on server*). Tendremos que configurar Tomcat en Eclipse, si no está configurado ya en este *workspace*. Veremos una página de prueba creada automáticamente por Maven, que estará desplegada en la siguiente dirección:

<http://localhost:8080/jbib-web/>

### 3.3. Configuración de la fuente de datos

En las sesiones de integración anteriores se accedía a base de datos a través de una unidad de persistencia JPA en la que se configuraba directamente el driver y la url a la que realizar la conexión. Ahora que vamos a pasar a una aplicación web, vamos a mejorar el rendimiento accediendo al pooling de conexiones que nos ofrece Tomcat.

Para ello creamos un fichero `context.xml` en la carpeta `META-INF` de nuestro proyecto web (`src/main/webapp/META-INF`), donde definiremos las características del pooling:

```
<Context>
```

```

<Resource
    name="jdbc/biblioteca"
    type="javax.sql.DataSource"
    auth="Container"
    username="root"
    password="expertojava"
    driverClassName="com.mysql.jdbc.Driver"
    url=
        "jdbc:mysql://localhost:3306/biblioteca?autoReconnect=true" />

<ResourceParams name="jdbc/biblioteca">

    <parameter>
        <name>maxActive</name>
        <value>20</value>
    </parameter>

    <parameter>
        <name>maxIdle</name>
        <value>5</value>
    </parameter>

    <parameter>
        <name>maxWait</name>
        <value>10000</value>
    </parameter>
</ResourceParams>

</Context>

```

También modificaremos el fichero `web.xml` de la aplicación para que refiera al recurso creado en el paso anterior. Para eso, añadimos estas líneas:

```

<resource-ref>
    <res-ref-name>jdbc/biblioteca</res-ref-name>
    <res-type>javax.sql.DataSource</res-type>
    <res-auth>Container</res-auth>
</resource-ref>

```

Si en el proyecto no se ha creado el descriptor de despliegue de forma automática, podéis utilizar este [web.xml](#) como plantilla.

Finalmente, tendremos que añadir al proyecto web una unidad de persistencia que acceda a la fuente de datos de Tomcat. Para ello:

- Creamos una carpeta `resources` en `src/main` (`src/main/resources`).
- Actualizamos en Eclipse la configuración del proyecto Maven, para que el IDE reconozca la carpeta anterior como carpeta de fuentes (pulsamos con el botón derecho sobre el proyecto, y seleccionamos *Maven > Update project ...*)
- Creamos dentro de `resources` una carpeta `META-INF`.
- Creamos dentro de la carpeta `META-INF` anterior un fichero `persistence.xml` como el siguiente:

```

<?xml version="1.0" encoding="UTF-8"?>
<persistence version="1.0"
    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/persistence_1_0.xsd">

```

```
<persistence-unit name="biblioteca"
    transaction-type="RESOURCE_LOCAL">
    <non-jta-data-source>
        java:comp/env/jdbc/biblioteca
    </non-jta-data-source>
    <class>es.ua.jtech.jbib.model.EjemplarDomain</class>
    <class>es.ua.jtech.jbib.model.LibroDomain</class>
    <class>es.ua.jtech.jbib.model.ReservaDomain</class>
    <class>es.ua.jtech.jbib.model.UsuarioDomain</class>
    <class>es.ua.jtech.jbib.model.AlumnoDomain</class>
    <class>es.ua.jtech.jbib.model.ProfesorDomain</class>
    <class>es.ua.jtech.jbib.model.BibliotecarioDomain</class>
    <class>es.ua.jtech.jbib.model.PrestamoHistoricoDomain</class>
    <class>es.ua.jtech.jbib.model.MultaDomain</class>

    <properties>
        <property name="hibernate.dialect"
            value="org.hibernate.dialect.MySQLInnoDBDialect" />
        <property name="hibernate.hbm2ddl.auto" value="validate" />
        <property name="hibernate.show_sql" value="true" />
    </properties>
</persistence-unit>

</persistence>
```

Puedes ver aquí el fichero [persistence.xml](#) completo. Al crear este fichero en el proyecto web se sobrescribirá el definido en `jbib-modelo`, de forma que ahora JPA accederá a la base de datos mediante el *pool* de conexiones de Tomcat, en lugar de conectar directamente.

#### Cuidado

El directorio META-INF donde debemos poner `persistence.xml` no es el META-INF general de la aplicación web donde se puso `context.xml`. El META-INF donde debemos guardarlo debe estar en el *classpath*, por eso lo ponemos en la carpeta de fuentes `resources`, para que durante la compilación sea volcado automáticamente a los directorios del *classpath* de la aplicación.

Una vez hechos estos cambios, instalaremos de nuevo el proyecto padre en el repositorio de Maven para comprobar que las pruebas siguen funcionando correctamente.

Sobre el driver de MySql deberemos tener en cuenta que ya no basta con incluirlo como dependencia del proyecto. Esto bastaría si sólo necesitásemos la base de datos para acceder a ella desde el código de nuestra aplicación. Sin embargo, dado que vamos a utilizar fuentes de datos del servidor, el servidor Tomcat también deberá acceder a ella para obtener conexiones de la fuente de datos. Por lo tanto, no bastará con que el driver esté disponible para nuestra aplicación, sino que deberá estar disponible para Tomcat. Deberemos copiar el [driver de MySql](#) al directorio de librerías comunes de Tomcat (`/opt/apache-tomcat-7.0.xx/lib`).

### 3.4. Configuración de la seguridad

Vamos ahora a definir seguridad declarativa para nuestra aplicación. Utilizaremos

seguridad basada en formularios para proteger todo el sitio web, de forma que sólo puedan acceder los usuarios definidos en la base de datos.

### 3.4.1. Realm de seguridad

Vamos a utilizar la propia tabla `usuario` para validar quién entra. En principio dejaremos pasar a cualquiera que esté registrado en esa tabla, y de forma programática haremos que según su perfil (rol) pueda realizar unas determinadas operaciones.

Para poder utilizar nuestra base de datos, debemos añadir un bloque `Realm` en el contexto de nuestra aplicación, que especifique que utilizaremos un `JDBCRealm` conectado a nuestra base de datos. Para ello, añadimos este bloque dentro de la etiqueta `Context` de nuestro fichero `META-INF/context.xml`:

```
<Realm className="org.apache.catalina.realm.JDBCRealm"
    driverName="com.mysql.jdbc.Driver"
    connectionURL=
        "jdbc:mysql://localhost:3306/biblioteca?autoReconnect=true"
    connectionName="root" connectionPassword="expertojava"
    userTable="usuario" userNameCol="login" userCredCol="password"
    userRoleTable="usuario" roleNameCol="tipo" />
```

### 3.4.2. Autentificación basada en formularios

En segundo lugar, debemos definir las páginas `login.jsp` y `errorLogin.jsp` en nuestra carpeta `WebContent`, con el formulario de validación y la página de error para utilizar seguridad basada en formularios.

#### Nota

Es importante que ambas páginas tengan extensión `.jsp`, ya que de lo contrario Eclipse no las admitirá como páginas de login y error válidas. Además, en futuras sesiones será mejor tener páginas JSP a las que se pueda incorporar contenido dinámico.

En el fichero `web.xml`, añadimos el bloque `login-config`:

```
<login-config>
    <auth-method>FORM</auth-method>
    <form-login-config>
        <form-login-page>/login.jsp</form-login-page>
        <form-error-page>/errorLogin.jsp</form-error-page>
    </form-login-config>
</login-config>
```

### 3.4.3. Protección de recursos

Finalmente, queda añadir las líneas de configuración de seguridad al fichero `web.xml` para proteger los recursos de toda la web. En primer lugar declararemos los roles que tienen acceso:

```
<security-role>
    <role-name>ADMIN</role-name>
</security-role>
<security-role>
    <role-name>BIBLIOTECARIO</role-name>
</security-role>
<security-role>
    <role-name>PROFESOR</role-name>
</security-role>
<security-role>
    <role-name>ALUMNO</role-name>
</security-role>
```

Tras esto, protegeremos todos los recursos de la web para que sólo los usuarios con estos roles tengan acceso:

```
<security-constraint>
    <web-resource-collection>
        <web-resource-name>Sitio web completo</web-resource-name>
        <url-pattern>/</url-pattern>
    </web-resource-collection>
    <auth-constraint>
        <role-name>ADMIN</role-name>
        <role-name>BIBLIOTECARIO</role-name>
        <role-name>PROFESOR</role-name>
        <role-name>ALUMNO</role-name>
    </auth-constraint>
</security-constraint>
```

Además, vamos a definir una serie de JSPs que sólo deberán ser accesibles de forma interna, ya que normalmente desde el navegador se hará la petición a un servlet, y este servlet delegará en un JSP para generar la respuesta. Por este motivo, deberemos proteger dichos JSPs para que no se pueda acceder a ellos desde el navegador. Para ello crearemos un directorio `jsp` dentro de la raíz de la web, y lo protegeremos con seguridad declarativa de forma que ningún rol tenga acceso a él:

```
<security-constraint>
    <web-resource-collection>
        <web-resource-name>JSPs</web-resource-name>
        <url-pattern>/jsp/*</url-pattern>
    </web-resource-collection>
    <auth-constraint></auth-constraint>
</security-constraint>
```

Haciendo esto, la única forma de acceder a los recursos del directorio `jsp` será hacer un `include` o un `forward` desde dentro de nuestra propia aplicación.

### 3.5. Componentes web

Vamos a crear ahora los componentes necesarios para construir la interfaz web de nuestra aplicación. Tendremos una serie de servlets que recogerán la petición llegada desde el cliente, comprobarán los permisos del usuario, y utilizarán los objetos de negocio definidos en sesiones anteriores para realizar la operación solicitada. Una vez realizada la operación, suministrará los datos obtenidos a un JSP para que se ocupe de generar la presentación y enviársela al cliente.

Vamos a comenzar implementando la funcionalidad de listado de libros.

### 3.5.1. Servlets

En primer lugar implementaremos el servlet `es.ua.jtech.jbib.ListarLibrosServlet` que obtendrá el listado de todos los libros mediante el BO correspondiente, y le pasará un atributo `listaLibros` en el ámbito de la petición al JSP `/jsp/biblio/listadoLibros.jsp` o `/jsp/usuario/listadoLibros.jsp`, según si el usuario que ha entrado es de tipo bibliotecario o alumno/profesor, para así mostrar un listado adaptado a cada tipo de usuario. Por el momento el servlet sólo podrá ser accedido por estos tres tipos de usuarios, no se proporciona ningún listado de libros para administradores.

```
@WebServlet("/ListarLibrosServlet")
public class ListarLibrosServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;

    static Log logger = LogFactory.getLog(ListarLibrosServlet.class);

    protected void doGet(HttpServletRequest request,
                         HttpServletResponse response)
                         throws ServletException, IOException {
        try {
            ILibroService ils = FactoriaServices.getInstance()
                .getLibroService();
            List<LibroDomain> lista = ils.listaLibrosPorTitulo();
            request.setAttribute("listaLibros", lista);

            RequestDispatcher rd = null;
            if (request.isUserInRole("BIBLIOTECARIO")) {
                rd = this.getServletContext().getRequestDispatcher(
                    "/jsp/biblio/listadoLibros.jsp");
            } else if (request.isUserInRole("PROFESOR") ||
                       request.isUserInRole("ALUMNO")) {
                rd = this.getServletContext().getRequestDispatcher(
                    "/jsp/usuario/listadoLibros.jsp");
            } else {
                rd = this.getServletContext().getRequestDispatcher(
                    "/jsp/error.jsp");
                request.setAttribute("error",
                    "Pagina no disponible para el usuario actual");
            }
            rd.forward(request, response);
        } catch (BibliotecaException ex) {
            request.setAttribute("error",
                "Error obteniendo el listado de libros. " + ex);
            logger.error("Error obteniendo el listado de libros. " + ex);

            RequestDispatcher rd = this.getServletContext()
                .getRequestDispatcher("/jsp/error.jsp");
            rd.forward(request, response);
        }
    }

    protected void doPost(HttpServletRequest request,
                         HttpServletResponse response) throws ServletException,
    IOException {
        doGet(request, response);
    }
}
```

}

Destacamos también que en todos nuestros servlets en caso de haber un error se hará un forward al JSP `error.jsp`. Este será un documento común que utilizaremos para mostrar cualquier error que se produzca en nuestra aplicación. En el caso del servlet anterior lo utilizamos para dar un error cuando un administrador intente acceder al listado de libros, o cuando se produzca un error en las operaciones de negocio.

Vamos a utilizar este listado de libros como punto de entrada a la aplicación. Por lo tanto, lo incluiremos entre las páginas de bienvenida, para que se cargue como página principal `/`:

```
<welcome-file-list>
    <welcome-file>ListarLibrosServlet</welcome-file>
</welcome-file-list>
```

#### Ayuda

Si aparecen fallos de compilación provenientes del directorio `target`, normalmente se solucionarán haciendo un `mvn clean`.

### 3.5.2. JSPs

Vamos a pasar ahora a crear los JSPs necesarios para presentar los resultados producidos por los servlets anteriores. En primer lugar crearemos el JSP para mostrar el listado de libros, tanto para bibliotecarios (`jsp/biblio/listadoLibros.jsp`), como para usuarios profesores y alumnos (`jsp/usuario/listadoLibros.jsp`), cada uno de ellos en el directorio correspondiente a su tipo de usuario.

Tras esto introduciremos la página de error común (`jsp/error.jsp`) que será mostrada cada vez que ocurra un error en alguna operación.

Estos JSP todavía no se podrán ejecutar, ya que todos ellos utilizan una serie de fragmentos comunes que deberemos crear a continuación.

### 3.5.3. Fragmentos de JSP

Vamos a introducir una serie de elementos comunes a todas las páginas del sitio web. Estos elementos son la cabecera, el menú de opciones y el pie. Los definiremos en ficheros `.jspf` independientes, y se incluirán en todas las páginas mediante la directiva `include`.

El fragmento para la cabecera estará dentro del directorio `jsp` creado anteriormente y se llamará `cabecera.jspf`.

Para el menú, crearemos el fragmento `jsp/biblio/menu.jspf` con las opciones para los bibliotecarios, y el fragmento `jsp/usuario/menu.jspf` con las opciones para los

alumnos y profesores. Separaremos de esta forma, en directorios distintos, los JSP destinados a cada tipo de usuario.

Por último, introduciremos el fragmento del pie de página común [jsp/pie.jspf](#).

### 3.5.4. Hoja de estilo

Podemos observar que los elementos de nuestros JSPs se encuentran organizados dentro de una serie de bloques `div` que representan los diferentes componentes de nuestros documentos web (cabecera, menú lateral, cuerpo, pie). Esta organización nos permitirá aplicar una hoja de estilo CSS para indicar el aspecto y la ubicación que tendrá cada uno de estos elementos. Podemos crear una hoja de estilo como el siguiente fichero [/css/style.css](#) dentro de la carpeta web de nuestra aplicación.

## Biblioteca jTech

Listado de libros					
	ISBN	Título	Autor	Páginas	Ejemplares disponibles
0131401572	Data Access Patterns	Clifton Nock	512	2	
0321127420	Patterns Of Enterprise Application Architecture	Martin Fowler	533	1	
0321180860	Understanding SOA with Web Services	Eric Newcomer and Greg Lomow	465	1	

© 2012-13 Experto Universitario Java Enterprise

### Listado de libros

### 3.5.5. Ejercicios

Vamos ahora a añadir nuevas funcionalidades a la aplicación, siguiendo el mismo esquema de separación de responsabilidades entre servlets y JSPs que hemos visto para la anterior. Concretamente, añadiremos las siguientes operaciones:

- Listar los ejemplares de un libro, junto con su disponibilidad, y la posibilidad de pedir un préstamo.
- Pedir el préstamo de un ejemplar, obteniendo la fecha en la que deberemos devolverlo.
- Mostrar un listado de los libros que tenemos actualmente prestados (tanto si están en casa como en sala).
- Cerrar sesión (*logout*).

Deberemos implementar los siguientes servlets en el paquete `es.ua.jtech.jbib.web`:

- `EjemplaresLibroServlet`: Recibe un parámetro `id_libro` con el identificador del libro del que queremos consultar los ejemplares. Obtendrá la lista de ejemplares a partir de los métodos de negocio implementados en sesiones anteriores, y se la pasará al JSP `/jsp/usuario/ejemplaresLibro.jsp` para que la muestre. Esta operación

sólo la podrán realizar los usuarios con rol ALUMNO/PROFESOR.

- `PedirPrestamoEjemplarServlet`: Recibe un parámetro `id_ejemplar` con el identificador del ejemplar a pedir prestado. El usuario que pide el préstamo deberá obtenerse a partir del usuario `logueado` actualmente en la web. Una vez pedido el préstamo, pasará el objeto `EjemplarDomain` obtenido a la página `/jsp/usuario/confirmacionPedirPrestamo.jsp` para que muestre la confirmación de la petición del préstamo y los datos de la misma (fecha de préstamo, fecha de devolución, etc). Esta operación sólo la podrán realizar los usuarios con rol ALUMNO/PROFESOR.
- `ListarMisLibrosServlet`: Muestra el listado de los ejemplares prestados por el usuario `logueado` actualmente, tanto si están en sala como en casa. Una vez obtenido este listado de ejemplares, se lo pasa al JSP `/jsp/usuario/misLibros.jsp` para que lo muestre. Esta operación sólo la podrán realizar los usuarios con rol ALUMNO/PROFESOR.
- `LogoutServlet`: Cierra la sesión y vuelve a la página principal de listado de libros (mostrará el formulario de `login`).

Si se produjese algún error en cualquiera de los servlets anteriores, se realizará un `forward` a `error.jsp` para mostrar la información del error producido.

Hemos visto que los servlets anteriores necesitan una serie de JSPs para presentar los datos. Estos JSPs son:

- `/jsp/usuario/ejemplaresLibro.jsp`: Muestra una tabla con la lista de ejemplares obtenidos, y de cada ejemplar muestra: número de ejemplar, localización, fecha de adquisición, estado, y operaciones. Si el ejemplar está disponible, en `operaciones` veremos un enlace que nos permitirá pedirlo prestado, llamando a `PedirPrestamoEjemplarServlet` pasando como parámetro el identificador del ejemplar. En caso de no estar disponible, si lo tenemos prestado nosotros mostrará el texto "*Ya lo tienes*", y si lo tiene otro usuario la columna aparecerá vacía.

## Biblioteca jTech

Usuario: altor  
Cerrar sesión

Ejemplares de Understanding SOA with Web Services				
Número de ejemplar	Localización	Fecha de adquisición	Estado	Operaciones
002	SALA	25/01/2007	DISPONIBLE	<a href="#">Reservar</a>
001	SALA	25/01/2007	PRESTADO	

© 2012-13 Experto Universitario Java Enterprise

Listado de ejemplares (I)

# Biblioteca jTech

Ejemplares de Understanding SOA with Web Services					
	Número de ejemplar	Localización	Fecha de adquisición	Estado	Operaciones
	002	SALA	25/01/2007	PRESTADO	Ya lo tienes
	001	SALA	25/01/2007	PRESTADO	

© 2012-13 Experto Universitario Java Enterprise

## Listado de ejemplares (II)

- /jsp/usuario/confirmacionPedirPrestamo.jsp: Muestra los datos de la petición de préstamo que acabamos de realizar: ISBN, título, número de ejemplar, fecha de préstamo, fecha de devolución.

# Biblioteca jTech

Confirmación de la reserva					
	ISBN	Título	Número de ejemplar	Fecha de préstamo	Fecha de devolución
	0321180860	Understanding SOA with Web Services	002	29/11/2012	29/12/2012

© 2012-13 Experto Universitario Java Enterprise

## Confirmación de pedir préstamo

- /jsp/usuario/misLibros.jsp: Muestra una tabla con el listado de ejemplares que tenemos prestados actualmente. Para cada ejemplar se muestra: ISBN, número de ejemplar, título, fecha de devolución, y observaciones. En *observaciones* mostraremos el texto "Pendiente de recoger" si el libro todavía está en SALA.

# Biblioteca jTech

Listado de mis libros					
	ISBN	Número de ejemplar	Título	Fecha de devolución	Observaciones
	0321127420	001	Patterns Of Enterprise Application Architecture		
	0321180860	002	Understanding SOA with Web Services	29/12/2012	Pendiente de recoger

© 2012-13 Experto Universitario Java Enterprise

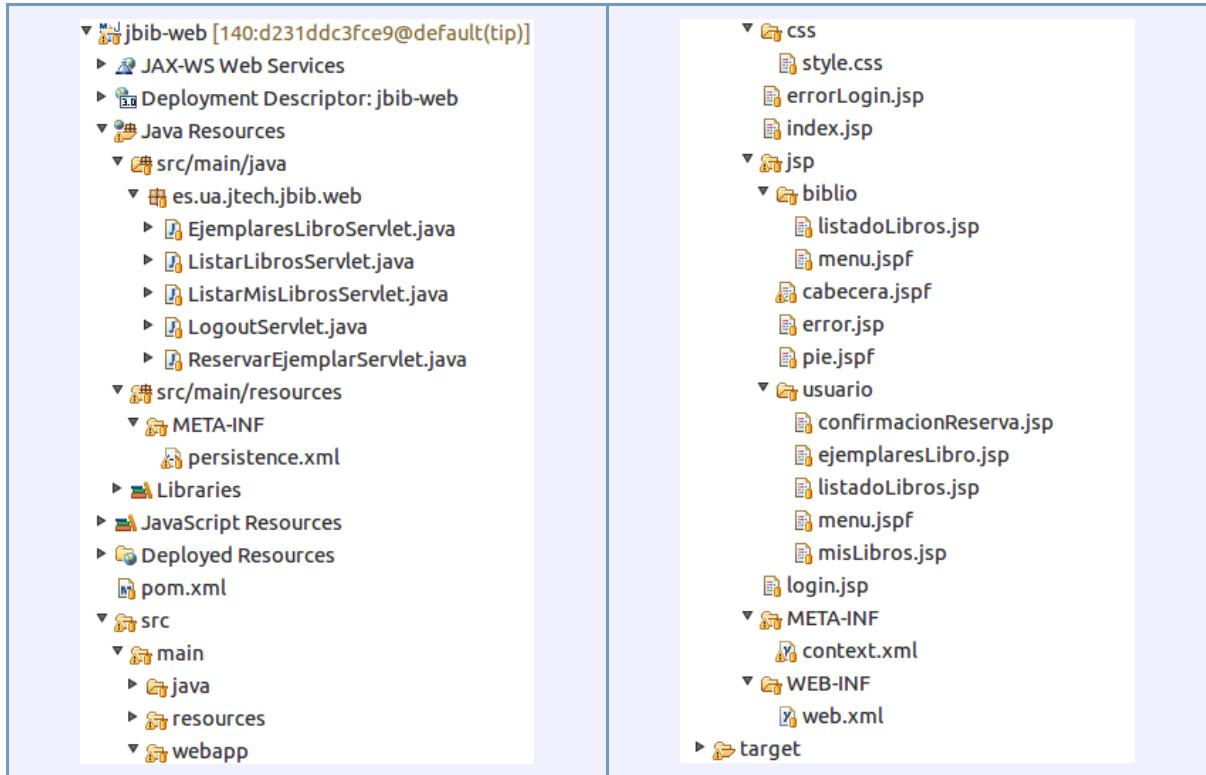
## Listado de mis libros

Estos JSPs deberán tener todos los elementos comunes de la estructura de las páginas de nuestro sitio web (cabecera, menú y pie).

### 3.6. Resumen

Como ayuda, proporcionamos la siguiente figura con la estructura de directorios

resultante de esta sesión.



Para la entrega se deberá etiquetar repositorio bitbucket con el tag entrega-proyint-web.

## 4. Servicios RESTful con Jersey

### 4.1. Introducción

En esta sesión de integración construiremos una interfaz web alternativa a la desarrollada en la sesión anterior. En este caso, en lugar de crear una interfaz HTML para acceder a la aplicación desde un navegador web, crearemos una serie de servicios RESTful para que otras aplicaciones puedan acceder a la información sobre los libros de la biblioteca. Este proyecto también se construirá sobre la capa de negocio desarrollada en sesiones anteriores, al igual que el proyecto web desarrollado en la sesión anterior.

### 4.2. Proyecto jbib-rest

Vamos a crear en primer lugar un nuevo proyecto con Maven, de nombre `jbib-rest`, utilizando para ello el arquetipo `webapp-javaee6`, al igual que el proyecto web de la sesión anterior. Por lo tanto, necesitaremos contar también con el plugin *Maven integration for WTP (optional)*, que ya deberemos tener instalado en Eclipse (si no es así, deberá instalarse siguiendo las instrucciones indicadas en la sesión anterior). Este nuevo proyecto contendrá todos los componentes necesarios para implementar los servicios RESTful.

En el espacio de trabajo del proyecto de integración crea un nuevo proyecto `jbib-rest`. Escoge la opción de Eclipse *New > Project... > Maven > Maven Module*, sin marcar la opción de *Create simple project* e introduciendo:

- *Module Name*: `jbib-rest`
- *Parent Project*: `proyint-jbib`
- *Archetype*: `webapp-javaee6` (se indica en la segunda pantalla del asistente)
- *Paquete*: `es.ua.jtech.jbib.rest` (se indica en la tercera pantalla del asistente)

Al igual que en el proyecto web, deberemos eliminar las etiquetas `groupId` y `version` del fichero POM del nuevo proyecto para que coja los valores del proyecto padre. También eliminaremos el bloque `<build> ... </build>` ya que no es necesario.

```

<parent>
    <artifactId>jbib</artifactId>
    <groupId>es.ua.jtech.proyint</groupId>
    <version>0.0.1-SNAPSHOT</version>
</parent>

<artifactId>jbib-rest</artifactId>
<packaging>war</packaging>

<name>jbib-rest</name>

```

También en este fichero POM, justo antes de las dependencias, deberemos añadir el repositorio de Maven donde se encuentra Jersey:

```
<repositories>
    <repository>
        <id>maven2-repository.java.net</id>
        <name>Java.net Repository for Maven</name>
        <url>http://download.java.net/maven/2</url>
        <layout>default</layout>
    </repository>
    <repository>
        <id>maven-repository.java.net</id>
        <name>Java.net Maven 1 Repository</name>
        <url>http://download.java.net/maven/1</url>
        <layout>legacy</layout>
    </repository>
</repositories>
```

Tras esto añade la dependencia con el proyecto jbib-negocio, jersey-server, jersey-servlet, y jersey-json:

```
<dependency>
    <groupId>com.sun.jersey</groupId>
    <artifactId>jersey-server</artifactId>
    <version>1.14</version>
</dependency>

<dependency>
    <groupId>com.sun.jersey</groupId>
    <artifactId>jersey-servlet</artifactId>
    <version>1.14</version>
</dependency>

<dependency>
    <groupId>com.sun.jersey</groupId>
    <artifactId>jersey-json</artifactId>
    <version>1.14</version>
</dependency>

<dependency>
    <groupId>es.ua.jtech.proyint</groupId>
    <artifactId>jbib-negocio</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <type>jar</type>
    <scope>compile</scope>
</dependency>
```

El fichero completo debe quedar tal como se indica en este [pom.xml](#).

Ahora podemos ejecutar nuestra aplicación web directamente desde Eclipse como hemos hecho a lo largo del curso (*Run As > Run on server*). Tendremos que configurar Tomcat en Eclipse, si no está configurado ya en este *workspace*. Veremos una página de prueba creada automáticamente por Maven, que estará desplegada en la siguiente dirección:

<http://localhost:8080/jbib-rest/>

## 4.3. Configuración del acceso a datos

Vamos ahora a configurar el pool de conexiones y el realm de seguridad de la misma forma que con el proyecto de la sesión anterior.

Para ello creamos un fichero `context.xml` en la carpeta `META-INF` de nuestro proyecto

REST:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<Context>
  <Resource
    name="jdbc/biblioteca"
    type="javax.sql.DataSource"
    auth="Container"
    username="root"
    password="expertojava"
    driverClassName="com.mysql.jdbc.Driver"
    url=
      "jdbc:mysql://localhost:3306/biblioteca?autoReconnect=true" />

  <ResourceParams name="jdbc/biblioteca">

    <parameter>
      <name>maxActive</name>
      <value>20</value>
    </parameter>

    <parameter>
      <name>maxIdle</name>
      <value>5</value>
    </parameter>

    <parameter>
      <name>maxWait</name>
      <value>10000</value>
    </parameter>
  </ResourceParams>

  <Realm className="org.apache.catalina.realm.JDBCRealm"
    driverName="com.mysql.jdbc.Driver"
    connectionURL=
      "jdbc:mysql://localhost:3306/biblioteca?autoReconnect=true"
    connectionName="root" connectionPassword="expertojava"
    userTable="usuario" userNameCol="login" userCredCol="password"
    userRoleTable="usuario" roleNameCol="tipo" />
</Context>
```

También modificaremos el fichero `web.xml` de la aplicación para que refiriese al recurso creado en el paso anterior. Para eso, añadimos estas líneas:

```
<resource-ref>
  <res-ref-name>jdbc/biblioteca</res-ref-name>
  <res-type>javax.sql.DataSource</res-type>
  <res-auth>Container</res-auth>
</resource-ref>
```

Si en el proyecto no se ha creado el descriptor de despliegue de forma automática, podéis utilizar este [web.xml](#) como plantilla.

También deberemos configurar la unidad de persistencia al igual que en la sesión anterior para que acceda a la fuente de datos que acabamos de crear. Se deberá poner este fichero [persistence.xml](#) en el directorio `src/main/resources/META-INF` del nuevo proyecto. También se copió en sesiones anteriores el driver de MySql al directorio de librerías de Tomcat. Si no lo tuviésemos copiado, deberemos hacerlo para que la aplicación funcione correctamente.

## 4.4. Configuración de Jersey

Vamos ahora a configurar en `web.xml` el servlet de Jersey mediante el cual accederemos a los servicios REST. Los recursos de Jersey estarán en el paquete `es.ua.jtech.jbib.rest.resources`, y accederemos a ellos a través de la URL `/resources`:

```
<servlet>
    <servlet-name>Jersey Web Application</servlet-name>
    <servlet-class>
        com.sun.jersey.spi.container.servlet.ServletContainer
    </servlet-class>
    <init-param>
        <param-name>
            com.sun.jersey.config.property.packages
        </param-name>
        <param-value>es.ua.jtech.jbib.rest.resources</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>Jersey Web Application</servlet-name>
    <url-pattern>/resources/*</url-pattern>
</servlet-mapping>
```

## 4.5. Configuración de la seguridad

Vamos ahora a definir seguridad declarativa para nuestra aplicación. Utilizaremos seguridad básica para proteger la operación de pedir préstamo, de forma que sólo puedan utilizarla los alumnos y profesores registrados.

### 4.5.1. Protección de recursos

Vamos a añadir las líneas de configuración de seguridad al fichero `web.xml` para proteger las operaciones que realizan modificaciones sobre los recursos (POST, PUT y DELETE). En primer lugar declaramos los roles que tendrán acceso:

```
<security-role>
    <role-name>PROFESOR</role-name>
</security-role>
<security-role>
    <role-name>ALUMNO</role-name>
</security-role>
```

Tras esto, protegeremos todos los recursos de la web para que sólo los usuarios con estos roles tengan acceso a las operaciones que alteran los datos:

```
<security-constraint>
    <web-resource-collection>
        <web-resource-name>Todos los recursos</web-resource-name>
        <url-pattern>/*</url-pattern>
        <http-method>DELETE</http-method>
        <http-method>POST</http-method>
        <http-method>PUT</http-method>
```

```
</web-resource-collection>
<auth-constraint>
    <role-name>PROFESOR</role-name>
    <role-name>ALUMNO</role-name>
</auth-constraint>
</security-constraint>
```

#### 4.5.2. Autentificación básica

En este caso ya no podemos utilizar seguridad basada en formularios, ya que necesitamos proporcionar una interfaz con la que otras aplicaciones puedan autenticarse de forma sencilla. Por lo tanto, utilizaremos seguridad de tipo BASIC. En el fichero web.xml, añadimos el bloque login-config:

```
<login-config>
    <auth-method>BASIC</auth-method>
</login-config>
```

### 4.6. Recursos REST

Vamos a crear recursos para gestionar el conjunto de libros y para manipular cada libro individual. Estos recursos se llamarán LibrosResource y LibroResource respectivamente, siendo el primero un recurso raíz mapeado a la ruta /libros y el segundo un subrecurso de éste, al que se accede cuando se añade el identificador de un libro a la ruta anterior. Todos los métodos soportarán las representaciones application/xml y application/json, salvo que se especifique lo contrario.

Vamos a utilizar beans JAXB para mapear los objetos Java que encapsulan la información de los libros a sus representaciones XML/JSON.

#### 4.6.1. Beans JAXB

Los beans estarán en un paquete es.ua.jtech.jbib.rest.beans. En la clase del bean añadiremos métodos para poder crear de forma sencilla estos objetos a partir de su correspondiente objeto de dominio:

```
public class LibroBean {

    Long id;
    String isbn;
    String titulo;
    String autor;
    int numPaginas;

    public LibroBean() {
    }

    public LibroBean(LibroDomain libro) {
        this.id = libro.getId();
        this.titulo = libro.getTitulo();
        this.autor = libro.getAutor();
        this.numPaginas = libro.getNumPaginas();
        this.isbn = libro.getIsbn();
    }
}
```

```
    }
    // Getters y setters
    ...
}
```

Añade a esta clase las anotaciones JAXB necesarias para que la etiqueta con los datos del libro tenga nombre `libro` y para que todos sus campos se muestren como atributos.

Tendremos también otro bean para representar el conjunto de libros:

```
public class LibrosBean {
    List<LibroBean> libros;
    public LibrosBean() {
    }
    public LibrosBean(List<LibroDomain> libros) {
        this.libros = new ArrayList<LibroBean>(libros.size());
        for(LibroDomain libro: libros) {
            this.libros.add(new LibroBean(libro));
        }
    }
    public List<LibroBean> getLibros() {
        return libros;
    }
    public void setLibros(List<LibroBean> libros) {
        this.libros = libros;
    }
}
```

Añade a esta clase las etiquetas necesarias para que la lista de libros se englobe en una etiqueta de nombre `libros`, y dentro de ella tengamos cada libro como un elemento de nombre `libro`.

#### 4.6.2. Listado y búsqueda

Cuando accedamos mediante GET a `/libros`, obtendremos la lista de todos los libros registrados en la biblioteca. Mediante una serie de parámetros podremos realizar búsquedas o paginar la lista de libros.

Si especificamos un parámetro `keyword` en la query, realizará una búsqueda de los libros que contengan la palabra especificada en su título o autor.

```
http://localhost:8080/jbib-rest/resources/libros?keyword=Patterns
<libros>
    <libro autor="Clifton Nock" id="1" isbn="0131401572"
           numPaginas="512" titulo="Data Access Patterns"/>
    <libro autor="Martin Fowler" id="2" isbn="0321127420"
           numPaginas="533"
           titulo="Patterns Of Enterprise Application Architecture"/>
</libros>
```

Si especificamos los parámetros `firstResult` y `maxResults` se mostrarán sólo

*maxResults* libros en la lista empezando desde el que se encuentra en la posición *firstResult* en la lista completa.

```
http://localhost:8080/jbib-rest/resources/libros?firstResult=1&maxResults=1

<libros>
    <libro autor="Martin Fowler" id="2" isbn="0321127420"
        numPaginas="533"
        titulo="Patterns Of Enterprise Application Architecture"/>
</libros>
```

Si no se especifica ninguno de los parámetros anteriores, se obtendrá el listado de todos los libros.

```
http://localhost:8080/jbib-rest/resources/libros
```

```
<libros>
    <libro autor="Clifton Nock" id="1" isbn="0131401572"
        numPaginas="512" titulo="Data Access Patterns"/>
    <libro autor="Martin Fowler" id="2" isbn="0321127420"
        numPaginas="533"
        titulo="Patterns Of Enterprise Application Architecture"/>
    <libro autor="Eric Newcomer and Greg Lomow" id="3"
        isbn="0321180860" numPaginas="465"
        titulo="Understanding SOA with Web Services"/>
</libros>
```

Todas las operaciones anteriores deben implementarse en un único método de `LibrosResource`, que según los parámetros recibidos realizará un filtrado distinto de los libros utilizando para ello las funciones implementadas en los objetos de negocio.

#### 4.6.3. Consulta de libros

Si a la ruta raíz `/libros` le añadimos el *id* de un libro, y accedemos a esta nueva ruta mediante GET, veremos los datos de un libro concreto. Esto lo implementaremos en el subrecurso `LibroResource`. Si el *id* especificado no existe en la base de datos, devolverá un código 404 Not found.

```
http://localhost:8080/jbib-rest/resources/libros/3
```

```
<libro autor="Eric Newcomer and Greg Lomow" id="3" isbn="0321180860"
    numPaginas="465" titulo="Understanding SOA with Web Services"/>
```

#### 4.6.4. Imágenes de las portadas

Podremos también obtener las portadas de los libros de la base de datos. Para acceder a ellas añadiremos a la ruta de un libro individual, tras el *id*, el fragmento `/imagen`. Haciendo GET sobre dicha ruta obtendremos la imagen codificada mediante `image/jpeg`.

Para tener accesibles las imágenes primero descargaremos este [pack de portadas](#) y las copiaremos al directorio `/imagenes` de nuestro proyecto. Tras esto, podremos utilizar un código como el siguiente para obtener una imagen:

```
@GET
```

```
@Path("/imagen")
@Produces("image/jpeg")
public Response getImagen(@Context ServletContext sc) {
    InputStream is =
        sc.getResourceAsStream("/imagenes/" +
            this.libro.getIsbn() + ".jpg");
    if(is==null) {
        return Response.status(Status.NOT_FOUND).build();
    } else {
        return Response.ok(is).build();
    }
}
```

Podemos ver que hemos inyectado el objeto `ServletContext` para poder localizar así la ruta en el disco donde están las imágenes y abrir un flujo de entrada para leerlas. Si la imagen existe devolvemos directamente el flujo de entrada para que los bytes de la imagen JPEG se envíen al cliente. Si no existe devolvemos 404 Not found.

#### 4.6.5. Estilo HATEOAS

Vamos a añadir ahora enlaces a las representaciones anteriores siguiendo el estilo HATEOAS, con la codificación Atom. Para ello podemos utilizar una clase `LinkBean` como la vista en el módulo, y modificar `LibroBean` para que incluya una lista de enlaces.

En el caso del listado de libros, cada libro deberá incluir un enlace con relación `self` que apuntará a la URI que da acceso al libro individual.

```
<libros>
    <libro autor="Clifton Nock" id="1" isbn="0131401572" numPaginas="512"
           titulo="Data Access Patterns">
        <link href="http://localhost:8080/jbib-rest/resources/libros/1"
              rel="self"/>
    </libro>
    <libro autor="Martin Fowler" id="2" isbn="0321127420" numPaginas="533"
           titulo="Patterns Of Enterprise Application Architecture">
        <link href="http://localhost:8080/jbib-rest/resources/libros/2"
              rel="self"/>
    </libro>
    <libro autor="Eric Newcomer and Greg Lomow" id="3" isbn="0321180860"
           numPaginas="465" titulo="Understanding SOA with Web Services">
        <link href="http://localhost:8080/jbib-rest/resources/libros/3"
              rel="self"/>
    </libro>
</libros>
```

En el caso de la consulta de un libro individual, deberemos obtener los siguientes enlaces:

- `self`: Apunta a la misma URI del libro actual.
- `libro/imagen`: Apunta a la URI en la que se encuentra la imagen del libro actual.

```
<libro autor="Eric Newcomer and Greg Lomow" id="3" isbn="0321180860"
       numPaginas="465" titulo="Understanding SOA with Web Services">
    <link href="http://localhost:8080/jbib-rest/resources/libros/3"
          rel="self"/>
    <link href="http://localhost:8080/jbib-rest/resources/libros/3/imagen"
          rel="libro/imagen"/>
</libro>
```

#### 4.6.6. Acceso a ejemplares

Vamos a añadir la posibilidad de acceder a los ejemplares de los libros. Para ello, añadiremos el fragmento /ejemplares a la ruta de un libro:

```
http://localhost:8080/jbib-rest/resources/libros/3/ejemplares
```

Con esto veremos la lista de ejemplares del libro indicado, envuelta en una etiqueta con los datos del libro:

```
<libro autor="Eric Newcomer and Greg Lomow" id="3" isbn="0321180860">
    numPaginas="465" titulo="Understanding SOA with Web Services">
    <ejemplares>
        <ejemplar fechaAdquisicion="2007-01-25T00:00:00+01:00" id="6"
            localizacion="SALA" numEjemplar="002">
            <link href="http://localhost:8080/jbib-rest/resources/ejemplar/6"
                rel="self"/>
        </ejemplar>
        <ejemplar fechaAdquisicion="2007-01-25T00:00:00+01:00"
            fechaDevolucion="2012-12-12T00:00:00+01:00"
            fechaPrestamo="2012-12-05T00:00:00+01:00"
            id="5" localizacion="SALA" numEjemplar="001">
            <link href="http://localhost:8080/jbib-rest/resources/ejemplar/5"
                rel="self"/>
        </ejemplar>
    </ejemplares>
</libro>
```

Para hacer esto:

- Crearemos un nuevo *bean* JAXB EjemplarBean.
- Introduciremos una lista de ejemplares en el objeto LibroBean (sólo tomará un valor cuando se solicite la lista de ejemplares).
- Introducimos en LibroResource la operación necesaria para obtener la lista de ejemplares.
- Al obtener un libro, deberemos añadir el enlace que nos da acceso a sus ejemplares:

```
<libro autor="Eric Newcomer and Greg Lomow" id="3" isbn="0321180860">
    numPaginas="465" titulo="Understanding SOA with Web Services">
    <link href="http://localhost:8080/jbib-rest/resources/libros/3"
        rel="self"/>
    <link href="http://localhost:8080/jbib-rest/resources/libros/3/imagen"
        rel="libro/imagen"/>
    <link href=
        "http://localhost:8080/jbib-rest/resources/libros/3/ejemplares"
        rel="libro/ejemplares"/>
</libro>
```

Por último, tendremos que dar la posibilidad de acceder a un ejemplar individual. La ruta que nos dará acceso a ellos es:

```
http://localhost:8080/jbib-rest/resources/ejemplar/5
```

Hay que destacar que ya no está bajo la ruta del libro, sino que los ejemplares individuales tienen su propia ruta y su propio recurso raíz. Del ejemplar obtendremos la siguiente información:

```
<ejemplar fechaAdquisicion="2007-01-25T00:00:00+01:00"
           fechaDevolucion="2012-12-12T00:00:00+01:00"
           fechaPrestamo="2012-12-05T00:00:00+01:00" id="5"
           localizacion="SALA" numEjemplar="001">
    <link href="http://localhost:8080/jbib-rest/resources/ejemplar/5"
          rel="self"/>
</ejemplar>
```

Para hacer esto implementaremos un nuevo recurso raíz `EjemplarResource` que nos permita obtener ejemplares individualmente proporcionando su *id*.

#### 4.6.7. Petición de préstamos

Vamos también a permitir pedir préstamos mediante REST. Para ello añadiremos a la ruta de un ejemplar individual el fragmento `/prestamo`, y realizaremos POST sobre esta ruta. Esta operación sólo la podrán realizar los usuarios de tipo alumno o profesor, si cualquier otro usuario intentase pedir un préstamo obtendrá un código 403 `Forbidden`. Para realizar esta comprobación inyectaremos un objeto de tipo `SecurityContext`.

La petición del préstamo se realizará para el usuario autenticado actualmente. Podemos obtenerlo mediante el método `getUserPrincipal().getName()` del contexto de seguridad.

##### Nota

Para probar esta funcionalidad ya no nos bastará con un navegador web, ya que debemos hacer una petición POST. Podemos utilizar el cliente genérico Swing visto en clase para realizar estas pruebas. Se debe introducir un login y password para poder realizar la operación, ya que está protegida mediante seguridad declarativa.

Una vez realizado el préstamo, esta operación nos devolverá los datos del ejemplar en los que ya figurará la fecha de préstamo y la fecha de devolución.

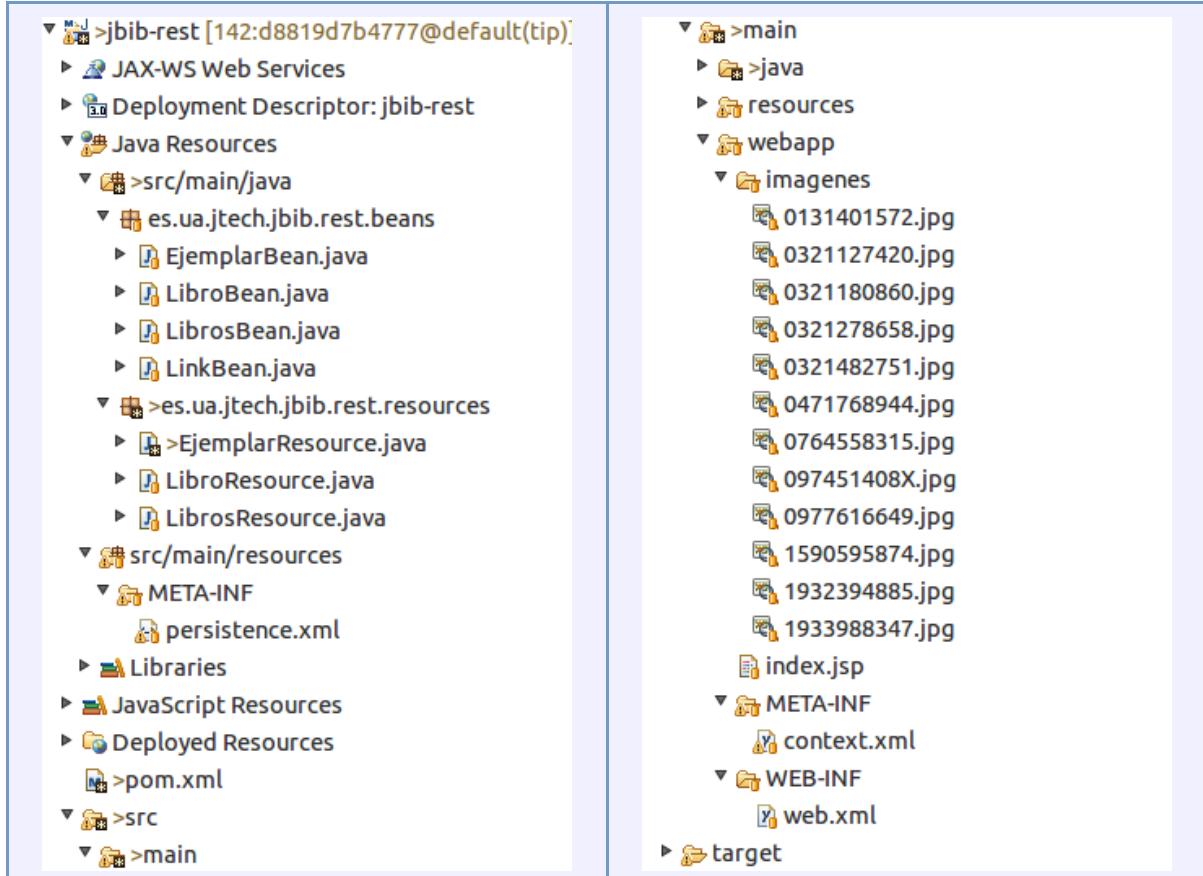
Además, haremos que los ejemplares que estén disponibles para préstamo tengan un enlace con relación `ejemplar/prestamo` a la URI correspondiente a la petición de préstamo. En caso de que el ejemplar no esté disponible, dicho enlace no debe aparecer.

```
<ejemplar fechaAdquisicion="2007-01-25T00:00:00+01:00"
           fechaDevolucion="2012-12-12T00:00:00+01:00"
           fechaPrestamo="2012-12-05T00:00:00+01:00" id="5"
           localizacion="SALA" numEjemplar="001">
    <link href="http://localhost:8080/jbib-rest/resources/ejemplar/5"
          rel="self"/>
    <link href=
          "http://localhost:8080/jbib-rest/resources/ejemplar/5/prestamo"
          rel="ejemplar/prestamo"/>
</ejemplar>
```

#### 4.7. Resumen

Como ayuda, proporcionamos la siguiente figura con la estructura de directorios

resultante de esta sesión.



Para la entrega se deberá etiquetar el repositorio bitbucket con el tag entrega-proyint-rest.

## 5. Sprint Web

### 5.1. Introducción

El objetivo de este Sprint es refactorizar la aplicación, añadir nuevos casos de uso, practicar un poco de I+D y reforzar los contenidos de los módulos de Componentes Web y JPA. Para ello se va a añadir parte de la funcionalidad explicada en la 1<sup>a</sup> sesión de Caso de Estudio, centrándonos en el punto de vista del socio (alumno o profesor).

Todo ello dándole importancia a la limpieza del código, minimizando la redundancia del mismo y favoreciendo los principios de bajo acoplamiento y alta cohesión.

Así pues, partiremos de la solución del proyecto de Componentes Web, pero sin tocar nada del proyecto REST.

### 5.2. Funcionalidades a Desarrollar

Lo primero que haremos será darle un lavado de cara al interfaz de usuario, refactorizando la página de login e introduciendo algunas imágenes que mejorar la apariencia de la aplicación.

#### 5.2.1. Login

Para entrar a la aplicación se nos presenta una página de **login**, la cual debe comprobar el rol del usuario y redirigirlo a su mini-site particular. Es importante considerar a los usuarios que puedan tener **multas** o sean **morosos**, ya que no deberán poder realizar ninguna acción.

The diagram illustrates the user authentication and role-based navigation process. It shows two states of the user on the left and their corresponding access paths on the right:

- user active:** Points to the 'Libros' (Books) section of the main library interface. This section displays a list of books with columns for ISBN, Title, Author, Pages, and Available Copies. It includes links for borrowing ('Elige Ejemplar a Prestar') and reserving ('Reservar').
- user delinquent or fined:** Points to the 'Usuario Multado' (Fined User) page. This page informs the user they have a fine or pending return and directs them to the library main page.

### Seguridad Declarativa

Tal como tenemos la aplicación, la página de login no permite el uso de hojas de estilo al estar nuestra hoja dentro de la parte protegida de la aplicación. Por lo tanto, hay que modificar el esquema de seguridad para proteger solo cierta parte de la webapp, no toda.

Independientemente del tipo de usuario, se mostrará una cabecera con información respecto al **usuario** conectado y el **rol** de dicho usuario.

### Login, datos y tipos de usuario

Para que la aplicación pueda decidir hacia qué vista redirigir al entrar a la aplicación (dependiendo de si el usuario es activo o moroso/multado), necesitamos un Servlet previo al listado de libros. En este Servlet, además, podremos obtener el rol del usuario y meterlo en la sesión

## 5.2.2. Operaciones

Cuando un usuario entra a la aplicación, se le mostrará un listado con todos los libros, mostrando la cantidad de ejemplares disponibles, y las **operaciones** que puede realizar el usuario sobre dicho libro (de momento reservar un libro o elegir el ejemplar sobre el que realizar el préstamo).

Además, la aplicación ofrece un par de **listados** que permitirán mostrar tantos los ejemplares prestados que tiene un usuario como los libros que tiene reservados; se entiende que solo se van a mostrar las operaciones *activas*, las visualización de las operaciones *históricas* sale fuera del alcance de esta sesión.

The screenshot shows the 'Listado de libros' (List of Books) page. On the left, there's a sidebar with 'Libros' and 'Mis operaciones'. Under 'Mis operaciones', it says 'Prestados Reservados'. The main area shows a table with columns: ISBN, Título, Autor, Páginas, Ejemplares disponibles, and two buttons: 'Elige Ejemplar a Prestar' and 'Reservar'. The table data is as follows:

	ISBN	Título	Autor	Páginas	Ejemplares disponibles	Opciones
	<a href="#">0131401572</a>	Data Access Patterns	Clifton Nock	512	2	<a href="#">Elige Ejemplar a Prestar</a>
	<a href="#">0321127420</a>	Patterns Of Enterprise Application Architecture	Martin Fowler	533	0	<a href="#">Reservar</a>
	<a href="#">0321180860</a>	Understanding SOA with Web Services	Eric Newcomer and Greg Lomow	465	1	<a href="#">Elige Ejemplar a Prestar</a>

At the top right, there's a logo for 'EXPERTO UNIVERSITARIO EN JAVA ENTERPRISE' and 'CERRAR SESIÓN'.

Dependiendo del estado del libro/ejemplar, el usuario podrá realizar las siguientes operaciones.

- **Préstamo de un ejemplar** . Operación ya realizada en sesiones anteriores.

- **Reserva de un libro** . Cuando un libro no disponga ningún ejemplar disponible, el usuario podrá realizar una reserva del libro siempre y cuando el usuario este *Activo* y tenga hueco en su cupo de operaciones. Además, debemos comprobar que un usuario no pueda realizar más de una reserva sobre el mismo libro.
- **Anular la reserva de un libro** . Sobre las reservas que tiene un usuario, o sobre una reserva que se acaba de realizar, se le permitirá al usuario cancelar la misma. En este caso, la reserva pasará a *Histórica* con un tipo de finalización de *Cancelado* .



- **Anular un préstamo que esté en Sala** . Cuando se ha realizado un préstamo de un ejemplar, y éste todavía no ha sido recogido por el usuario y esté pendiente de recoger, se podrá anular la operación. Así pues, se creará un *PrestamoHistorico* y dependiendo de si el libro tenía alguna reserva previa, pasaremos la reserva *Histórica* con un tipo de finalización de *Finalizado* , creando un nuevo préstamo con el usuario de la reserva. Si el libro no tenía ninguna reserva, se liberará el libro de modo que haya un ejemplar más disponible.

### 5.3. A Entregar

Para facilitar la labor de compresión de la funcionalidad y el flujo de navegación, se ha colgado un solución "no testeada" en [server.jtech.ua.es/jbib-web](http://server.jtech.ua.es/jbib-web).

El sprint tiene una calificación máxima de 10 puntos, repartidos de la siguiente manera:

- Calidad del código: limpieza, bajo acoplamiento, alta cohesión (2 ptos)
- Login y cabecera de la aplicación (2 pto)
- Casos de Uso (6 ptos)
  - Reserva de un libro (2 pto)
  - Anular reserva (2 pto)
  - Cancelación de un préstamo (2 ptos)

Si la sesión hubiese sido más larga, podríamos haber realizado las siguientes cuestiones:

- Mostrar los libros que tiene que devolver un usuario moroso.
- Mostrar la fecha de finalización de la multa de un usuario multado.
- Los errores que se muestran al usuario deberían ser más amigables.
- Mostrar el cupo de operaciones de un usuario en la cabecera, para que sepa cuantas tiene activas y su tipo, y cuantas puede realizar.
- Realizar un buscador de libros.
- Envío de un mail al producirse cualquier préstamo/reserva y sus correspondientes anulaciones.

En futuras sesiones trataremos la caducidad de los préstamos de manera automática, de modo que el estado de un usuario pase a Moroso si no ha devuelto el ejemplar a tiempo, a no ser que el ejemplar siguiese en la sala y se anularía el préstamo sin consecuencias para el usuario.

Recordar que la fecha de entrega es el **17 de Enero** y que el proyecto se realiza por parejas. . Para la entrega utilizad la etiqueta **entrega-sprint** . Si no entregais en plazo, usad la etiqueta **entrega-sprint-prorroga**

## 6. Componentes de presentación

### 6.1. Introducción

El objetivo de este módulo es la creación de una parte privada para la gestión de libros por parte de un usuario bibliotecario. En ella podremos realizar el CRUD (Create-Read-Update-Delete) de libros, así como ver un listado de los mismos. Además, introduciremos la posibilidad de subir las portadas de los libros, que se guardarán en base de datos como un LOB. Para ello, deberemos modificar nuestra tabla *libro* añadiendo una nueva columna mediante la siguiente sentencia SQL.

```
alter table `libro` add picture longblob null;
```

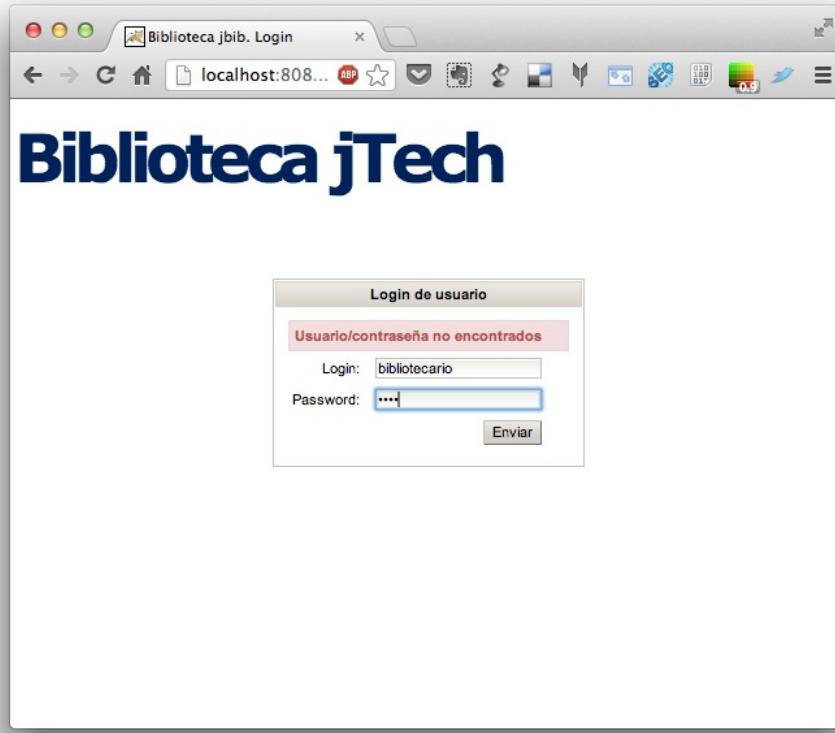
Para la entrega se deberá etiquetar repositorio bitbucket con el tag entrega-proyint-jsf.

### 6.2. Login del bibliotecario

Lo primero que deberemos desarrollar será el login del bibliotecario. Si no hemos creado ningún bibliotecario en nuestra base de datos, podemos crear uno mediante la siguiente sentencia sql:

```
INSERT INTO `biblioteca`.`bibliotecario` (
    `id`,
    `email`,
    `login`,
    `nif`,
    `password`
) VALUES (
    NULL,
    'bibliotecario@expertojee.es',
    'bibliotecario',
    '11222333N',
    'bibliotecario'
);
```

En la pantalla de login, el bibliotecario introducirá su nombre y contraseña. Si no se introducen correctamente o el usuario no está en el sistema, se mostrará un mensaje de error. En caso contrario, accederemos a la pantalla principal.



También, si un usuario se encuentra ya logado en el sistema y quisiéramos acceder a la página de login, seremos redirigidos directamente a la página principal.

Cuando hagamos login, deberemos guardar en sesión un objeto de la clase BibliotecarioDomain. **NO** se permite anotar esta clase con @ManagedBean.

### 6.3. Página principal - Listado de libros

La primera página con la que nos encontraremos nada más realizar el login será el listado de los libros actualmente dados de alta en la aplicación. Éstos se mostrarán en forma de tabla paginada(máximo de ítems por página: 4), con los datos indicados en la siguiente imagen:

ISBN	Título	Autor	Páginas	Fecha de Alta	Disponibles	Carátula	Acciones
1884777651	Distributed Programming with Java	Qusay H. Mahmoud	320	31/12/1998	1		<a href="#">Editar</a> <a href="#">Eliminar</a>
1930110049	Domino Development with Java	Anthony Patton	467	05/08/2000	6		<a href="#">Editar</a> <a href="#">Eliminar</a>
1930110944	EJB Cookbook	Benjamin G. Sullins and Mark B. Whipple	352	16/01/2003	17		<a href="#">Editar</a> <a href="#">Eliminar</a>
9781617290961	Grails in Action, Second Edition	Glen Smith and Peter Ledbrook	525	10/12/2012	11		<a href="#">Editar</a> <a href="#">Eliminar</a>

Observemos que, si un libro no tiene carátula, se mostrará un mensaje indicándolo

En caso de no tener libros con los que operar, [aquí tienes un script de creación de unos cuantos](#)

## 6.4. Menú del usuario

El menú del usuario nos mostrará su nombre, y nos permitirá las opciones de:

- Ver el listado de libros
- Crear un nuevo libro
- Cerrar la sesión. La sesión deberá ser invalidada y deberemos ser redirigidos a la pantalla de login una vez invalidada.



## 6.5. Borrado de libros

A la hora de borrar un libro, siempre preguntaremos primero al usuario de si está seguro de querer realizar la operación.



Si por algún caso no pudiésemos borrar el libro, lo indicaríamos al usuario

Biblioteca jTech

Menú principal (bibliotecario)

Listado de libros

Nuevo libro

Logout

Listado de libros

Error. No se ha podido borrar el libro seleccionado

ISBN	Título	Autor	Páginas	Fecha de Alta	Disponibles	Carátula	Acciones
9781935182849	GWT in Action, Second Edition	Adam Tacy, Robert Hanson, Jason Essington, and Anne Tökke	680	20/01/2013	21		<a href="#">Editar</a> <a href="#">Eliminar</a>

1 2 3 4

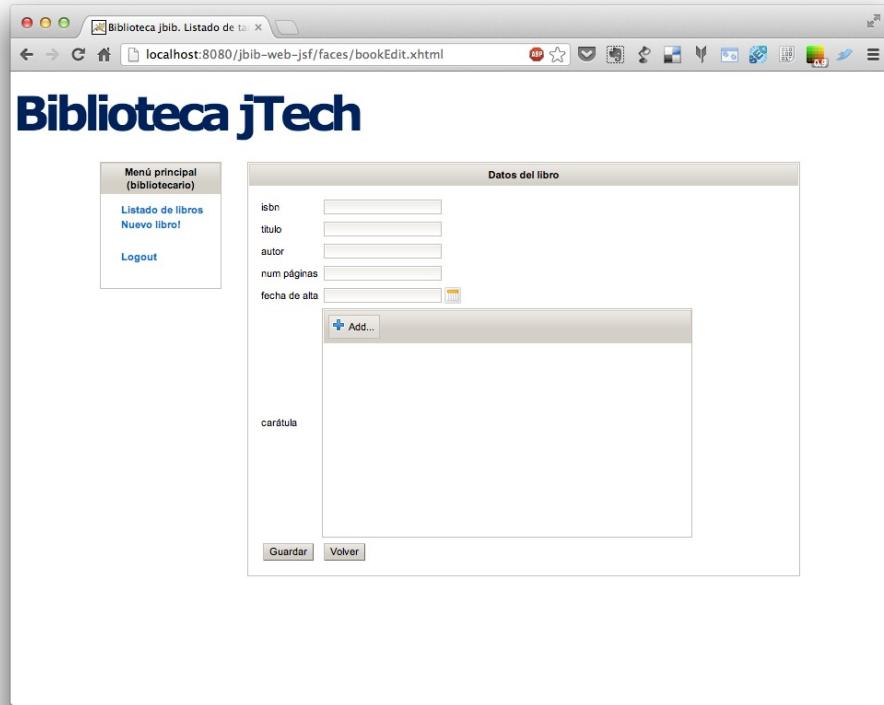
En caso contrario, también indicaremos al usuario que el borrado se ha realizado exitosamente

The screenshot shows a web browser window for 'Biblioteca jTech' at 'localhost:8080/jbib-web-jsf/faces/bookList.xhtml'. The page title is 'Listado de libros'. A message at the top says 'El libro se eliminó con éxito'. The table lists four books:

ISBN	Título	Autor	Páginas	Fecha de Alta	Disponibles	Carátula	Acciones
1930110049	Domino Development with Java	Anthony Patton	467	05/06/2000	6		<button>Editar</button> <button>Eliminar</button>
1930110944	EJB Cookbook	Benjamin G. Sullins and Mark B. Whipple	352	16/01/2003	17		<button>Editar</button> <button>Eliminar</button>
9781617290961	Grails in Action, Second Edition	Glen Smith and Peter Ledbrook	525	10/12/2012	11	Sin imagen	<button>Editar</button> <button>Eliminar</button>
9781935182849	GWT in Action, Second	Adam Tacy, Robert Hanson, Jason	680	20/01/2013	21		<button>Editar</button> <button>Eliminar</button>

## 6.6. Creación de libros

Crearemos un formulario que nos permitirá dar de alta un nuevo libro.



Una vez se haya creado el libro, mostraremos el siguiente mensaje (intenta usar variables flash para ello):

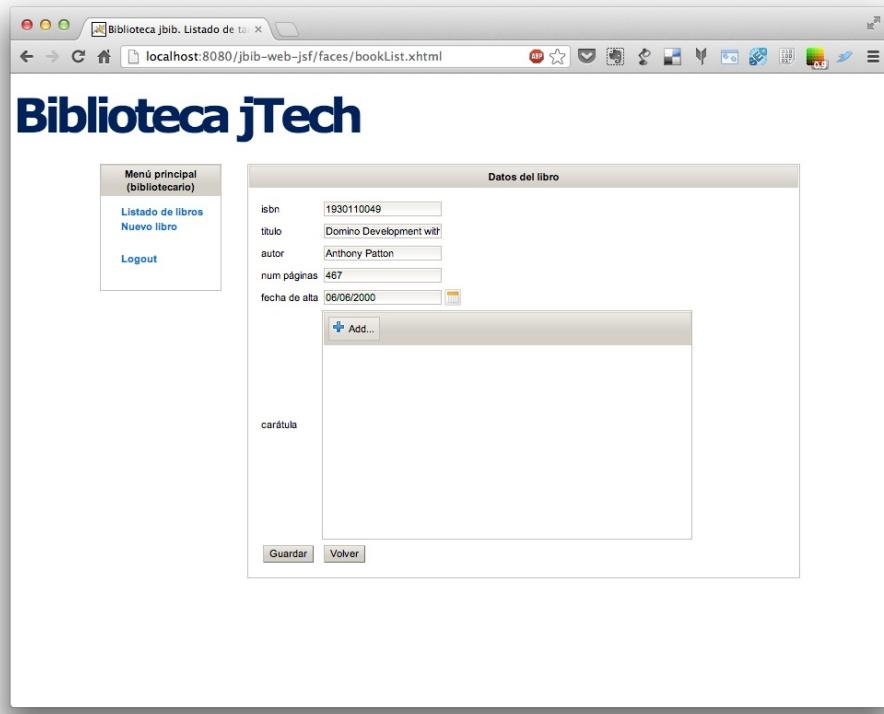
### Listado de libros

El libro se creó con éxito

ISBN	Título	Autor	Páginas	Fecha de Alta	Disponibles	Carátula	Acciones
------	--------	-------	---------	---------------	-------------	----------	----------

## 6.7. Edición de libros

También, tendremos la posibilidad de editar un libro existente. El formulario a utilizar deberá ser obligatoriamente el mismo que hemos empleado para la creación de libros.



Una vez se haya creado el libro, mostraremos el siguiente mensaje (intenta usar variables flash para ello):

Listado de libros							
El libro se actualizó con éxito							
ISBN	Título	Autor	Páginas	Fecha de Alta	Disponibles	Carátula	Acciones
	Grails in ...	Glen					

## 6.8. Validaciones

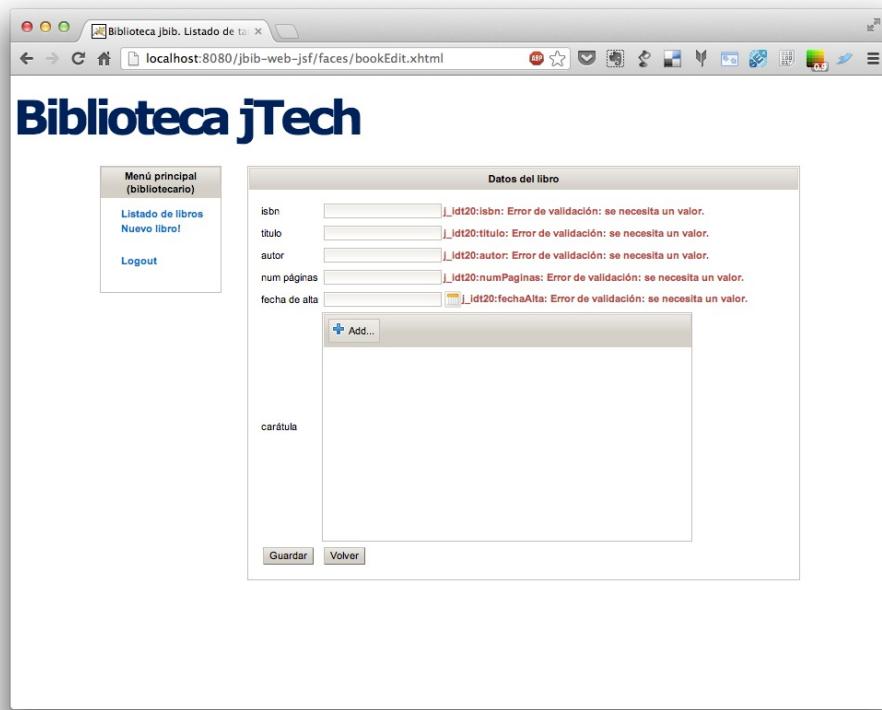
Tanto la creación como edición de libros deberán tener las siguientes validaciones:

- ISBN: 10 dígitos (validar mediante expresión regular)
- Título: obligatorio. Longitud entre 4 y 255 caracteres.
- Autor: obligatorio. Longitud entre 4 y 255 caracteres.
- Número de páginas: obligatorio. Valor mínimo: 1.
- Fecha de alta: obligatorio. Formato: dd/mm/aaaa. Deberemos poder seleccionarla de un *datepicker*.

- Carátula: opcional. Deberemos buscar algún componente en [RichFaces](#) que nos permita permita realizar la subida de ficheros.

No podemos modificar la entidad libro para añadir anotaciones JSR 303.

No es necesario personalizar los mensajes de validación, aceptaremos como válidos los que se muestran por defecto.



## 6.9. Control de acceso

Deberemos realizar un control de acceso mediante eventos para que no se permita el acceso a una vista si el usuario no está logado. Si se da el caso, seremos redirigidos a la pantalla de login

## 6.10. Ayuda: Esqueleto de proyecto

[En este enlace](#) puedes descargar un esqueleto de proyecto. Tiene algunos elementos implementados, así como un servlet que recupera una imagen de un libro de base de datos para mostrarlo en una página web. Además, [aquí](#) podrás descargar las clases que contienen nuevos servicios o modificaciones en los dominios, cuyos cambios serán explicados durante la sesión.



## 7. Integración componentes Java EE: EJB y JMS

### 7.1. GlassFish y Componentes EJB

Vamos a crear una nueva versión del proyecto de Biblioteca basada en una arquitectura EJB. Se debe desarrollar una nueva versión del proyecto, partiendo del último Sprint del proyecto de integración.

Utilizaremos la arquitectura definida en la sesión 4 de EJB, en la que a capa de lógica de datos se implementa con EJBs de sesión en los que se inyectan los DAOs que se encargan de la persistencia.

En la primera parte de la sesión tendrás que integrar la aplicación desarrollada hasta ahora en GlassFish y NetBeans, cambiando el proveedor de persistencia de Hibernate a EclipseLink. Tendrás también que refactorizar la aplicación a la nueva arquitectura en la que los servicios se implementan con componentes EJB.

#### 7.1.1. Pasos

Vista general de los pasos a realizar:

1. Crear el dominio de GlassFish
2. Configurar la fuente de datos en el dominio
3. Clonar de Bitubucket la estructura y proyectos POM de los módulos de la solución
4. Completar los módulos:
  - Completar el módulo con el proyecto modelo con las entidades JPA
  - Completar el módulo con la capa de persistencia (clases DAO)
  - Crear la capa de lógica de negocio como EJBs que acceden a los DAO por inyección de dependencias, haciendo remotos los métodos relacionados con el bibliotecario
5. Crear una aplicación cliente

#### 7.1.2. Configuración del dominio GlassFish

Debes crear un dominio nuevo llamado *jbib-domain*. Créalo *sin password* con el comando:

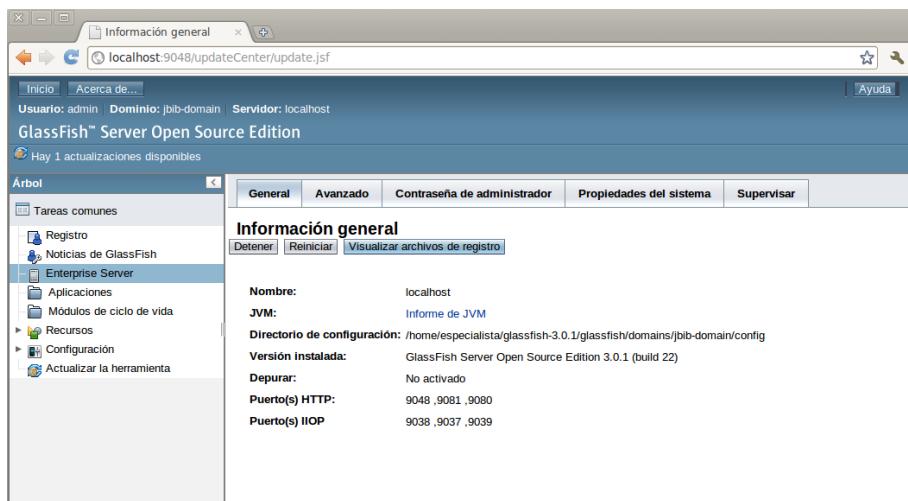
```
./asadmin create-domain 9000 jbib-domain
```

Una vez creado añade un nuevo servidor a NetBeans utilizando este nuevo dominio. Llámalo por ejemplo *Server jbib*:



Nuevo servidor

Lánzalo y abre una consola de administración en <http://localhost:4048>:



Consola de administración

### 7.1.3. Creación de la fuente de datos XA

Configura la fuente de datos en GlassFish en el dominio `jbib-domain` como una fuente de datos XA y dale como nombre `jdbc/jbib`. Copia el driver JDBC de MySQL en el directorio `lib` del dominio. Ahora ya puedes lanzar GlassFish y conectarte a su consola de administración. Es recomendable lanzarlo desde NetBeans, para poder visualizar sus mensajes en el panel de `Output`.

Crea en GlassFish una fuente de datos de tipo XA conectada a la base de datos biblioteca. Primero crea el conjunto de conexiones `BibliotecaPool`:

### Nuevo conjunto de conexiones de JDBC (paso 1 de 2)

Identifique las preferencias generales del conjunto de conexión.

#### Configuración general

Nombre: *	<input type="text" value="BibliotecaPool"/>
Tipo de recurso:	<input type="text" value="javax.sql.XADataSource"/> ▼ Se debe indicar si la clase de fuente de datos implementa
Proveedor de la base de datos:	<input type="text" value="MySQL"/> ▼ Seleccionar o introducir un proveedor de la base de datos

Define los siguientes parámetros:

- user: root
- password: expertojava
- URL: jdbc:mysql://localhost:3306/biblioteca

Termina creando la nueva fuente de datos JDBC con el nombre `jdbc/jbib` basada en el conjunto de conexiones que acabas de crear.

#### 7.1.4. Proyecto enterprise

En el repositorio `java_ua/proyint-ent-expertojava` hay una versión inicial de la solución que contiene la estructura Maven de todos los módulos, con sus ficheros POM. También contiene un ejemplo con una entidad (`BibliotecarioDomain`), y el DAO y el EJB que implementa su capa de servicio.

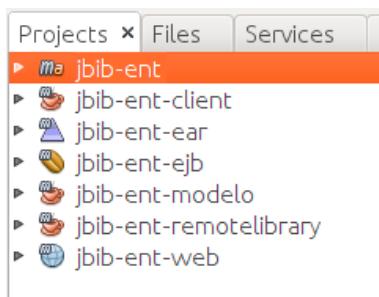
Los proyectos que hay en la solución son:

- `jbib-ent`: Proyecto Maven padre de todos. Contiene las dependencias utilizadas en todos los demás proyectos y sirve para construir todos los artefactos.
- `jbib-ent-ear`: Aplicación EAR que se despliega en el servidor y que contiene los proyectos EJB y WEB.
- `jbib-ent-modelo`: Librería JAR con las clases del modelo y las reglas de negocio. La hemos separado de la capa de persistencia para poder distribuirla más fácilmente a las aplicaciones clientes.
- `jbib-ent-ejb`: Proyecto EJB que contiene la capa de persistencia, con el fichero de configuración `persistence.xml`, las clases DAO que se inyectan en los EJBs de sesión (clases Service) que definen la capa de lógica de negocio.
- `jbib-ent-web`: Proyecto web que se conecta con las interfaces locales de los EJB de la capa de servicio.
- `jbib-ent-remotelibrary`: Librería JAR con las interfaces remotas de los EJB.
- `jbib-ent-client`: Aplicación remota que se trabaja con las interfaces remotas de los EJB.

1. Clona en tu cuenta de bitbucket el repositorio `java_ua/proyint-ent-expertojava`.

Usa el mismo nombre y no olvides quitar la opción de heredar permisos de acceso.

2. Desde Netbeans, descarga en tu disco el repositorio con la opción *Team > Mercurial > Clone Other...*. Pon el nombre del repositorio que acabas de clonar: <https://bitbucket.org/login/proyint-ent-expertojava>. Deja ese mismo repositorio como repositorio push y pull por defecto. Pon como directorio padre el lugar donde quieras que se guarde el repositorio (por ejemplo /home/expertojava/NetBeansProjects/) y deja proyint-ent-expertojava como nombre del repositorio. Di que NetBeans abra los proyectos descargados desplegando el proyecto padre y seleccionando todos.



3. Comprueba que funcionan correctamente todos los módulos:

- Construye todos los módulos pulsando el botón derecho sobre el proyecto padre jbib-ent y seleccionando *Clean and Build*. Se deben pasar todos los tests, entre ellos los de la capa de persistencia que utilizan DbUnit para añadir unos datos iniciales a la única tabla Bibliotecario que hemos incluido en la versión inicial.
- Comprueba que la tabla Bibliotecario se ha creado y que contiene a javier.huertas.
- Despliega la aplicación EAR en el servidor pulsando el botón derecho sobre el proyecto jbib-ent-ear y seleccionando *Run*. Selecciona el servidor cuando NetBeans te lo pida.
- Comprueba en la URL <http://localhost:8080/jbib-ent-web/PruebaServicios> que el servlet se conecta correctamente con la interfaz local del EJB BibliotecarioService.
- Por último lanza la aplicación cliente remota en el proyecto jbib-ent-client para comprobar que la conexión remota con el EJB funciona correctamente.

4. Completa todos los módulos refactorizando el proyecto de integración realizado hasta ahora:

- En el proyecto jbib-ent-modelo añade las entidades con las clases de modelo. Hemos cambiado a EclipseLink como proveedor de persistencia para trabajar mejor con GlassFish, por lo que quizás tengas que cambiar alguna característica dependiente de Hibernate.
- En el proyecto jbib-ent-ejb completa el persistence.xml y el persistence-test.xml para incluir todas las clases de entidad, añade los DAO, más

datos de prueba a DBUnit y los tests de los DAO. Crea por último todos los EJB refactorizando las clases de servicio, construyendo sus interfaces locales y remotas. Incluye en las interfaces remotas las mismas operaciones que en las locales.

- Actualiza el servlet de prueba del proyecto web para probar algún método local de cada servicio que vas añadiendo.

### 7.1.5. Aplicaciones clientes

Crea varias aplicaciones clientes por línea de comando que permitan:

- Gestión de libros: listados, búsquedas y actualización de libros
- Operaciones: probar alguna operación de la biblioteca

## 7.2. JMS y MDBs

El objetivo de esta parte de la sesión es integrar las tecnologías JMS/MDB dentro de nuestro proyecto de integración. Para ello, vamos a permitir que la aplicación envíe mensajes a un cola a través de un EJB, y que consuma mensajes mediante un MDB.

Ambos ejercicios los crearemos dentro del proyecto EJB que ya tenemos.

### 7.2.1. Multas Externas

Crearemos un MDB (`es.ua.jtech.jbib.service.MultasExternasMDB` dentro del proyecto `jbib-ent-ejb`) que escuche peticiones de multa de usuarios de sistemas externos. Así pues, la aplicación escuchará de una cola (`JBibMultasUsuariosQueue`) los envíos de sistemas externos respecto a los usuarios que tienen multas, mediante un mensaje de texto que contiene el login y el número de días de multa, ambos separados mediante el carácter '#'.

Una vez recibido el mensaje, el MDB debe guardar la multa en la base de datos, y cambiar el estado del usuario de ACTIVO a MULTADO. En el caso de que el usuario ya fuese moroso/multado, crearemos una nueva multa activa.

#### Gestión de multas

Para simplificar la gestión de multas, no vamos a comprobar si tenía otra multa activa previamente. Es decir, si el usuario ya tenía una multa previa y nos llega un mensaje con otra multa, tendrá dos multas.

Todas las operaciones deben formar parte de una transacción distribuida, de modo que si salta alguna excepción en alguna operación, se deshagan todas las operaciones.

### 7.2.2. Deshaciendo las Reservas

En este ejercicio, vamos a crear un *Timer* para deshaga las reservas que han caducado.

Vamos a considerar que las reservas que tienen más de 30 días son reservas que se deberían cancelar automáticamente. Así pues, una vez se invoque al método marcado con `@TimeOut`, a parte de borrar las reservas caducadas, vamos a enviar un mensaje a la cola (`JBibReservasCaducadasQueue`) con el login y el isbn de la reserva que acaba de caducar.

Si al mismo tiempo caduca más de una reserva, se enviarán tantos mensajes como reservas caducadas.

Para invocar el *timer*, tenemos que crear un Servlet de inicio de aplicación (pista:`load-on-startup`), el cual llame al método del EJB que lanza el timer.

Aquí tenéis un esqueleto de ejemplo de código de EJB, el cual podeis colocar dentro de uno de los EJBs que ya teneis creado o en un nuevo:

```
public void initTimer() {
    context.getTimerService().createTimer(1000, 1000*10,
    "TimerCaducaReserva");
}

@Timeout
@TransactionAttribute(TransactionAttributeType.REQUIRED)
public void caducaReservas(Timer timer) {
    String nombre = (String) timer.getInfo();

    if ("TimerCaducaReserva".equals(nombre)) {
        // 1º comprobamos si hay reservas caducadas
        // 2º por cada reserva caducada,
        //      cancelamos la reserva y enviamos mensaje a la cola
    }
}
```

Recordad que deberéis crear y cerrar la conexión JMS mediante los *callbacks* EJB, tal como vimos en la cuarta sesión de teoría.

#### Consulta de Reservas Caducadas

Para obtener las reservas que tenéis caducadas, podeis reutilizar la consulta de obtener las reservas activas y recorrer las que han caducado, o aún mejor, crear una consulta que únicamente devuelva las reservas caducadas.

### 7.2.3. Probando...

Para comprobar ambos ejercicios, vamos a crear dos aplicaciones cliente:

- `jbib-multas-client`: cliente JMS que envía un mensaje de texto a `JBibMultasUsuariosQueue` con el login y el número de días de la multa separados por '#'.
- `jbib-reservas-client`: cliente JMS que escucha en modo asíncrono un mensaje de texto de `JBibReservasCaducadasQueue` con el login y el isbn del libro cuya reserva ha caducado, ambos separados por '#'.

Para que ambos ejercicios funcionen, tenéis que crear los recursos JMS necesarios:

- Una factoría de conexiones denominada JBibConnectionFactory.
- 2 colas, una para las multas (JBibMultasUsuariosQueue) y otra para las reservas caducadas (JBibReservasCaducadasQueue).

## 8. Integración con Servicios Web

### 8.1. Introducción

En esta sesión de integración vamos a ofrecer como servicio web algunas de las operaciones implementadas mediante EJBs en sesiones anteriores. También utilizaremos un servicio web externo para obtener información extendida sobre los libros de nuestra biblioteca. Por último, crearemos un cliente Java para nuestro servicio.

#### Nota

Para realizar el trabajo de esta sesión tomaremos como punto de partida el resultado de la sesión de integración con JMS (aunque realmente no vamos a utilizar JMS, por lo que se podría partir del resultado de la integración con EJBs).

Crearemos todos los proyectos de esta sesión dentro de la carpeta `proyint-ent-expertojava/jbib-ent`

### 8.2. Creación de servicios web

Vamos a exponer como un servicio web SOAP la funcionalidad que obtiene información de los libros de nuestra biblioteca, implementada actualmente por un EJB. Dicho servicio, por lo tanto, podrá ser utilizado por otras aplicaciones (de nuestra empresa o fuera de ella).

#### CREAMOS EL PROYECTO

Comenzaremos creando un nuevo proyecto web Maven (dentro de la carpeta `jbib-ent`, que contiene el proyecto multi-módulo de nuestra biblioteca) con los siguientes datos:

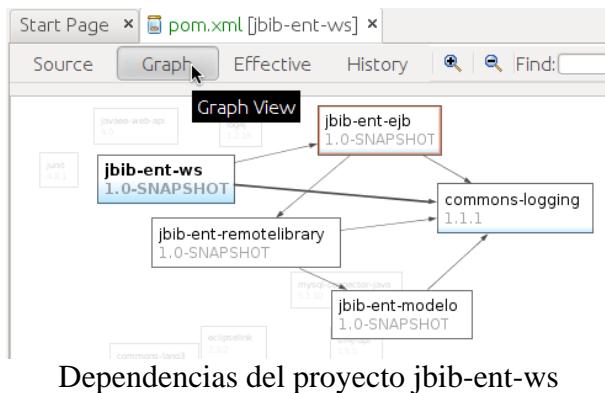
- nombre: `jbib-ent-ws`
- groupID: `es.ua.jtech.proyint`
- version: `1.0-SNAPSHOT`
- package: `es.ua.jtech.jbib.ws`
- server: Server `jbib`

#### MODIFICAMOS LAS DEPENDENCIAS DE LOS PROYECTOS

Nuestro proyecto (`jbib-ent-ws`) necesita acceder al proyecto (va a depender de) `jbib-ent-ejb` por lo que añadiremos las dependencias correspondientes en nuestro `pom`. Pondremos el `scope` a `provided`. Con esto estamos indicando que el `jar` correspondiente se necesitará para compilar el proyecto `jbib-web-services` pero no se incluirá en el empaquetado final del mismo. Para añadir la dependencia tenemos dos opciones: (1) Modificamos directamente el fichero `pom.xml` e incluimos la nueva dependencia con la etiqueta `<dependency>`; (2) En el nodo `Dependencies` de nuestro proyecto, pinchamos

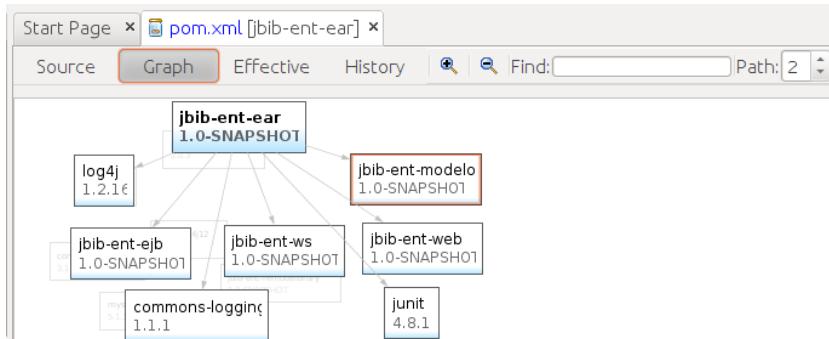
con botón derecho y seleccionamos: **Add Dependency...**

Podemos visualizar gráficamente las dependencias de nuestro proyecto, situándonos sobre él y pinchando con botón derecho sobre *Open POM*. Deberemos cambiar a la vista "Graph" en lugar de "Source", que es la opción por defecto. En la figura mostrada, podemos pinchar sobre *jbib-ent-ws* y veremos lo siguiente:



Dependencias del proyecto *jbib-ent-ws*

También tenemos que añadir, en el proyecto *jbib-ent-ear* la dependencia con nuestro proyecto *jbib-ent-ws*, de la misma forma que hemos comentado (con la diferencia de que el *scope* debe ser el de por defecto, es decir, *compile*). Después de hacer las modificaciones, las dependencias de la aplicación *enterprise* serán:



Dependencias del proyecto *jbib-ent-ear*

Además, nuestro proyecto tiene que incluirse como un módulo de la aplicación *jbib-ent*. Añadimos el módulo *jbib-ent-ws* en el proyecto *jbib-ent*. De nuevo podemos optar por editar directamente el pom.xml. O, de forma alternativa, podemos añadir el módulo desde el nodo *Modules* del proyecto *jbib-ent*, seleccionando con botón derecho la opción **Add Existing Module...**

El contenido del fichero **pom.xml** de nuestro proyecto multimódulo (*jbib-ent*) debe contener la siguiente lista de módulos (en el mismo orden en que se muestran).

```
<project>
  ...

```

```

<modules>
    <module>jbib-ent-modelo</module>
    <module>jbib-ent-ws</module>
    <module>jbib-ent-ejb</module>
    <module>jbib-ent-web</module>
    <module>jbib-ent-ear</module>
    <module>jbib-ent-client</module>
    <module>jbib-ent-remotelibrary</module>
</modules>
...
</project>

```

## CREAMOS EL SERVICIO WEB

El proyecto `jbib-ent-ws` contendrá nuestro servicio web que va a ofrecer dos operaciones, utilizando la funcionalidad de la implementación del ejb que hemos creado en sesiones anteriores. Los datos con los que crearemos nuestro servicio web serán:

- Name: **Librows**
- package: `es.ua.jtech.jbib.ws`
- Marcaremos *Create WS from Existing Session Bean*
- Elegiremos el elemento `LibroService`

Concretamente, las operaciones que deberá ofrecer este servicio serán (únicamente):

- `buscaLibroPorIsbn`
- `listaLibros`

La operación `buscaLibroPorIsbn` nos devuelve los datos del libro solicitado a partir de su `isbn`. La segunda operación (`listaLibros`) permitirá a un cliente externo obtener los datos de todos los libros de nuestra biblioteca.

Tendremos que editar el código generado, comentando el resto de métodos, ya que por defecto Netbeans ha expuesto como operaciones del servicio, todos los métodos del ejb que hemos seleccionado. Además, vamos a modificar el tipo del elemento que retornan las operaciones de nuestro servicio, para que devuelvan objetos de tipo `LibroTO`, y `List<LibroTO>`, respectivamente.

- `public LibroTO buscaLibroPorIsbn(String isbn)`
- `List<LibroTO> listaLibros()`

El objeto `LibroTO` es un *TransferObject* (una clase java plana con atributos y sus correspondientes *getters* y *setters*). Esto nos permitirá tener control sobre las estructuras de datos que se serializan durante la llamada a los servicios y así evitar enviar información no necesaria o con un formato que no resulte adecuado para la conversión a XML. Utilizando entidades (como por ejemplo `LibroDomain`) tendremos problemas con los campos que relacionan dichas entidades con otras (relaciones uno a muchos). Por ejemplo, si devolvemos un `LibroDomain`, el servicio web generado intentará devolver también los ejemplares y reservas, lo cual causará un problema al intentar recuperar esta información de forma *lazy*.

La clase **LibroTO**, en la que volcaremos los datos de nuestros objetos `LibroDomain`, se ubicará en el paquete `es.ua.jtech.jbib.ws` del proyecto `jbib-ent-ejb`, y tendrá la siguiente información:

```
public class LibroTO implements Serializable {  
    private String isbn;  
    private String titulo;  
    private String autor;  
    private Integer numPaginas;  
  
    //constructor vacío, necesario para que desde el  
    //servidor de aplicaciones se pueda crear una instancia  
    //de LibroTO para devolver la respuesta en el caso de que tengamos  
    definido  
    //algún constructor CON parámetros, como en este ejemplo  
    public LibroTO() {  
    }  
  
    public LibroTO(LibroDomain libro) {  
    }  
  
    //getters y setters
```

Podemos ver que esta clase tiene un constructor a partir de un `LibroDomain`, que nos permitirá volcar de forma sencilla los datos de la entidad JPA a nuestro *Transfer Object*. No olvides crear también un constructor sin parámetros. Sin él, el servidor de aplicaciones no será capaz de crear la instancia de nuestro servicio Web.

Podemos consultar el wsdl generado por Glassfish al desplegar nuestro servicio accediendo a:

```
http://localhost:8080/jbib-ent-ws/LibroWS?wsdl
```

Podemos probar el servicio creado con el cliente de pruebas que nos genera Glassfish al desplegar el servicio accediendo a:

```
http://localhost:8080/jbib-ent-ws/LibroWS?tester
```

### 8.3. Acceso a servicios web externos

Vamos a añadir a la aplicación de la biblioteca la funcionalidad de obtener los detalles de un libro (portada, precio, editorial y fecha de publicación). Para hacer esto recurriremos a un servicio web (*InfoLibroSWService*) que hemos creado en el servidor de jtech para el curso de especialista, y al que denominaremos servicio web de Jtech.

#### Opcional

Si queréis probar el servicio web de Jtech lo podéis hacer accediendo a:

<http://server.jtech.ua.es:3700/servcweb-infoLibro-jtech/InfoLibrosWSService?tester>.  
Puedes utilizar el siguiente valor como isbn: 0321127420

Primero vamos a añadir un nuevo *Transfer Object* LibroInfoTO, en el paquete es.ua.jtech.jbib.ws del proyecto jbib-ent-ejb:

```
public class LibroInfoTO implements Serializable{

    private String isbn;
    private float precio;
    private String imagen;
    private int imagenAncho;
    private int imagenAlto;
    private String editorial;
    private String anyoEdicion;

    //getters y setters
}
```

Creamos dentro del módulo EJB (*jbib-ent-ejb*) un cliente para acceder al servicio web de Jtech, cuyo documento WSDL se puede encontrar en la siguiente dirección:

<http://server.jtech.ua.es:3700/servcweb-infoLibro-jtech/InfoLibrosWSService?wsdl>

Indicaremos que el paquete en el que deben generar las clases al ejecutar *wsimport* será: *es.ua.jtech.jbib.ws*

Después de crear el cliente (y por lo tanto generar los stubs necesarios para acceder al servicio), tenemos que asegurarnos de que en el pom del proyecto *jbib-ent-ejb*, se encuentra el directorio *src/main/resources* como un directorio de recursos:

```
<build>
  <resources>
    ...
    <resource>
      <directory>src/main/resources</directory>
    </resource>
  </resources>
  ...
</build>
```

#### Importante. Ficheros de recursos

Por defecto, todos los ficheros incluidos en el directorio *src/main/resources* se copian en el directorio raíz del jar/war generado. Así, por ejemplo, podemos ver que en dicho directorio (podemos ir a la pestaña "Files", o bien desde la pestaña "Projects", pinchamos sobre el nodo "Other Sources") tenemos el fichero "META-INF/persistence.xml", que se copiará en el jar generado (ver el contenido del fichero "jbib-ent-ejb-1.0-SNAPSHOT.jar" en el directorio tarjet). Ahora bien, cuando creamos el cliente del servicio Web, lo que hace NetBeans de forma automática es modificar el pom añadiendo el plugin *wsimport* y ejecutar dicho pom, generando así los *stubs*. Además de incluir el plugin, de forma automática se modifica el directorio de recursos por defecto, de forma que, además del directorio */src/main/resources*, se incluyan los nuevos recursos a copiar en el jar, que serán: el directorio *src/wsdl*, más el fichero *src/jax-ws-catalog.xml*.

Añadiremos una nueva operación `recuperaLibroInfo` en la interfaz local del ejb (`ILibroService`):

```
es.ua.jtech.jbib.ws.LibroInfoTO recuperarLibroInfo (String isbn);
```

Y a continuación implementaremos dicho método en `LibroService.java`. Aquí es donde incluiremos el código para llamar al servicio web de Jtech, concretamente tendremos que utilizar la operación `buscaLibro`, del servicio `InfoLibroWSService`. Dicha operación devuelve información detallada de un libro a partir de su isbn.

#### Importante. Bug de Glassfish

Debido a un *bug* de Glassfish, la inyección de la referencia al servicio web (anotación `@WebServiceReference`) no funciona correctamente, de forma que, al ejecutar el servicio web, el servidor no encuentra el wsdl asociado a dicha referencia. En consecuencia, las llamadas al objeto Service provocará una excepción. Para solucionar este problema tendremos que quitar la anotación `@WebServiceReference`, y en su lugar, crearemos "a mano" nosotros dicha instancia en lugar de dejar que Glassfish lo haga. Por lo tanto añadiremos la línea `service = new InfoLibroWSService();` antes de realizar la llamada `service.getInfoLibroSWPort()`.

### 8.3.1. Probamos el acceso al servicio Web externo

Para probar la llamada al servicio Web de Jtech, lo haremos desde un nuevo *servlet*, al que llamaremos `InfoLibrosJtech`, que crearemos en la aplicación web `jbib-ent-web`. Para crear el nuevo servlet utilizaremos los siguientes parámetros:

- Name: **InfoLibrosJtech**
- package: `es.ua.jtech.jbib.servlet`
- servletName: `InfoLibrosJtech`
- URL: `/InfoLibrosJtech`

El código del servlet hará una llamada a una instancia de `LibroService`, accediendo al método `recuperaLibroInfo` utilizando, por ejemplo, el valor de isbn `"0321180860"`. Se mostrarán por pantalla los datos: isbn, editorial, año de edición, precio, dirección de descarga de la portada del libro. A continuación mostramos un ejemplo de ejecución del servlet:



### Resultado de la ejecución del servlet InfoLibrosJtech

Para mostrar la imagen con la portada, podéis utilizar un código similar a éste:

```
out.println("<img src= \\""+imagen+"\\" alt=\\"Portada\\>")
```

Siendo *imagen* la variable (de tipo *String*) que contiene la dirección url del fichero de imagen

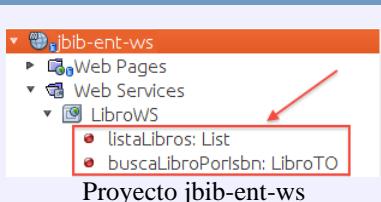
## 8.4. Cliente para el servicio Web

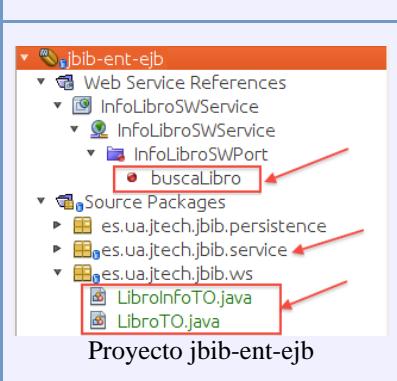
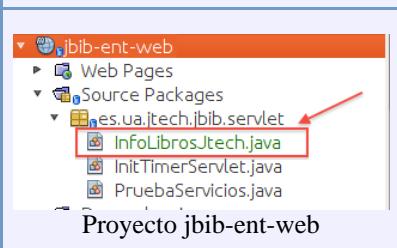
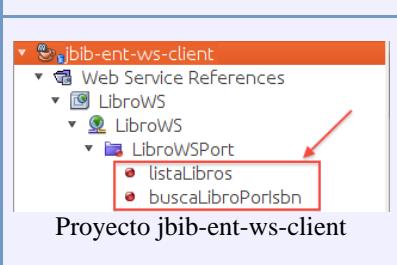
Crear un cliente Java con Netbeans para el servicio web de la biblioteca (*LibroWS*), en un nuevo proyecto Maven Java al que llamaremos *jbib-ent-ws-client*.

El programa deberá invocar la operación *listaLibros* y mostrar en la consola el listado de libros.

De cada libro se mostrará su ISBN, su título, y su autor.

## 8.5. Resumen

 <p>Proyecto jbib-ent-ws</p>	<p>Se deberá añadir un nuevo proyecto Maven Web <i>jbib-ent-ws</i> con el siguiente servicio web:</p> <ul style="list-style-type: none"> <li>• <i>LibroWS</i>: Servicio web que ofrecerá las operaciones:           <ul style="list-style-type: none"> <li>• <i>buscaLibroPorIsbn</i>, y</li> <li>• <i>listaLibros</i></li> </ul> </li> </ul>
---	---

 <p>Proyecto jbib-ent-ejb</p>	<p>Se deberán añadir los siguientes componentes al módulo jbib-ent-ejb:</p> <ul style="list-style-type: none"> <li>• LibroTO: <i>Transfer Object</i> con los datos detallados de un libro proporcionados por Jtech.</li> <li>• LibroInfoTO: <i>Transfer Object</i> con los datos detallados de un libro proporcionados por Jtech.</li> <li>• Cliente del servicio de Jtech: <i>Stub</i> para acceder a los servicios de Jtech.</li> <li>• Operación <code>recuperaInfoLibro</code>, en <i>LibroService</i>, que obtiene los datos detallados de un libro mediante el servicio de JTec</li> </ul>
 <p>Proyecto jbib-ent-web</p>	<p>Se deberá añadir al proyecto Maven Web jbib-ent-web:</p> <ul style="list-style-type: none"> <li>• <i>Servlet InfoLibrosJtech</i>: para probar el cliente del servicio Web de Jtech desde el ejb.</li> </ul>
 <p>Proyecto jbib-ent-ws-client</p>	<p>Se deberá añadir un nuevo proyecto Maven Java jbib-ent-ws-client, que contendrá:</p> <ul style="list-style-type: none"> <li>• Un cliente Java que acceda a nuestro servicio. Obtendrá la lista de todos los libros de nuestra biblioteca, y la mostrará por la consola</li> </ul>

## 9. Sprint Enterprise

### 9.1. Introducción

El objetivo de este Sprint Enterprise es repasar los últimos contenidos aprendidos, de modo que integremos la solución que teníamos funcional en el Sprint Web (con su interfaz de usuario amigable), y desarrollar nuevos casos de uso para codificar teniendo en cuenta todas las capas de una aplicación empresarial completa.

### 9.2. Funcionalidades a Desarrollar

#### 9.2.1. Seguridad Declarativa mediante JDBCRealm

Lo primero que haremos será integrar la seguridad declarativa. Para ello, mediante un JDBCRealm de Glassfish tenéis que integrar el login que teníamos en el proyecto web.

Recordad que no es recomendable utilizar contraseñas en texto plano. Para ello, vamos crear una nueva columna en la tabla usuario para almacenar la contraseña encriptada mediante MD5, utilizaremos el siguiente comando:

```
ALTER TABLE biblioteca.usuario ADD COLUMN PASSWORD2 VARCHAR(255) AFTER  
PASSWORD;  
UPDATE usuario SET PASSWORD2=MD5(PASSWORD);
```

Así pues el JDBCRealm utilizará esta nueva columna para comprobar la contraseña del usuario.

#### 9.2.2. Integración de la capa web con EJBs

A continuación, vamos a modificar los servlets para que en vez de pasar por la factoría accedamos a los EJB directamente desde los Servlets.

Para ello, en todos los Servlets hemos de eliminar el uso de `FactoriaService` y utilizar las anotaciones `@EJB` para referenciar a los servicios de negocio.

En este momento, la aplicación debe funcionar tal cual la teníamos en el Sprint Web.

#### 9.2.3. Casos de Uso

A partir de aquí, vamos a desarrollar las siguientes operaciones mediante el uso de Timers:

- Devolución automática de un libro que está prestado en Sala: comprobar si ha finalizado la fecha de devolución de un libro, y si todavía se encuentra en SALA,

pasarlo a DISPONIBLE o RESERVADO dependiendo de si tenía reservas pendientes. Las reglas de negocio de esta operación son muy parecidas a las de anularPrestamo que ya codificamos en el sprint web.

- Asignación de estado de usuario a MOROSO si ha finalizado la fecha de devolución de un préstamo y todavía no ha devuelto el libro.
- Caducidad de multas y asignación de estado de usuario a ACTIVO si ha finalizado la fecha de finalización de multa. Dependiendo de la gestión de multas que hicisteis en el MDB de la sesión de Componentes Enterprise, es posible que un usuario tenga más de una multa activa en un instante determinado. Por ello, en este caso, habréis de comprobar esta casuística, y en dicho caso, si el usuario sigue teniendo alguna multa activa, pasar a HISTORICO la multa que ha caducado pero mantener al usuario como MULTADO.

En estos tres casos de uso, un vez realizada la operación, la aplicación debe mandar un mail para informar al usuario de lo que acaba de suceder. El envío de emails se debe realizar desde una clase POJO ubicada en el proyecto EJB, ya sea mediante el uso directo de *JavaMail* o con la librería *commons-email*. Se recomienda crear una cuenta de Google para hacer las pruebas de envío de mail, ya que la cuenta de la UA sólo funciona como servidor SMTP desde dentro de la universidad.

### 9.3. A Entregar

El sprint enterprise tiene una calificación máxima de 10 puntos, repartidos de la siguiente manera:

- Calidad del código: limpieza, bajo acoplamiento, alta cohesión (1 ptos)
- Seguridad declarativa (1.5 pto)
- Integración de capa web con EJBs (1.5 pto)
- Casos de uso (4.5 ptos)
  - Devolución de libros en SALA (1.5 ptos)
  - Usuarios MOROSO (1.5 ptos)
  - Usuario ACTIVO (1.5 ptos)
- Envío de emails (1.5 ptos)

Recordar que la fecha de entrega es el **16 de Mayo** y que el proyecto se realiza por parejas. Para la entrega utilizad la etiqueta **entrega-sprint-ent** . Si no entregáis en plazo, usad la etiqueta **entrega-sprint-entprorroga**

