

P06A- Proyectos Maven multimódulo

Os recordamos que el plazo para entregar los ejercicios de esta práctica (P06A) termina justo ANTES de comenzar la siguiente sesión de prácticas (P06B) en el laboratorio (los grupos de los lunes, por lo tanto, el próximo lunes, los de los martes, el próximo martes, ...). Pasado este plazo los profesores no revisarán vuestro trabajo sobre P06A (de forma individual), con independencia de que se indiquen en clase las soluciones que os permitirán saber si las vuestras son correctas.

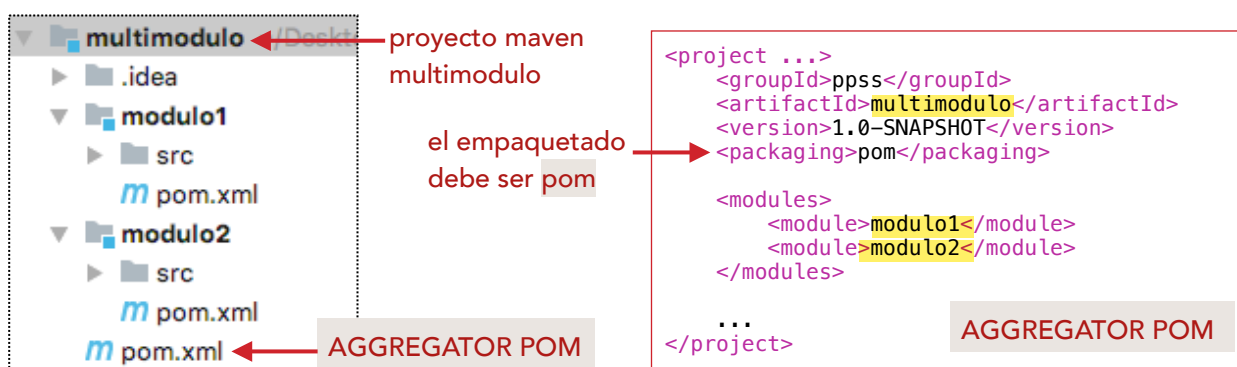
Proyectos Maven multimódulo

Dependiendo de lo “grande” que sea nuestro proyecto software, es habitual “distribuir” el código en varios subproyectos. Hasta ahora hemos trabajado con proyectos maven independientes, de forma que cada uno contenía “todo” nuestro código.

Maven permite trabajar con proyectos que a su vez están formados por otros proyectos maven, a los que llamaremos “**módulos**”, de forma que podemos establecer diferentes “**agrupaciones**” entre ellos, tantas como sean necesarias. Un proyecto Maven que “agrupa” a otros proyectos se denomina proyecto maven multimódulo

Un proyecto maven multimódulo, tiene un empaquetado (etiqueta <packaging>) con el valor “pom”, y contiene una lista de módulos “agregados” (que son otros proyectos Maven). Por eso a este pom también se le denomina “**aggregator pom**”.

A continuación mostramos un ejemplo de un proyecto maven multimódulo que “contiene” dos módulos. Dichos módulos pueden estar físicamente en cualquier ruta de directorios, pero lo más práctico es que los módulos se encuentren físicamente en el directorio del *pom aggregator*



Observa que el proyecto multimódulo sólo contiene el pom.xml, NO tiene código (carpeta src). En el directorio del proyecto **añadiremos tantos módulos como queramos agrupar, y lo indicaremos en el pom anidando etiquetas <module>.**

Los módulos agregados de nuestro proyecto multimódulo son proyectos maven “normales”, y **pueden construirse de forma separada o a través del aggregator pom.**

Un **proyecto multimódulo se construye a partir de su aggregator pom**, que gestiona al grupo de módulos agregados. Cuando construimos el proyecto a través del *aggregator pom*, cada uno de los proyectos maven agrupados se construyen si tienen un empaquetado distinto de *pom*.

El mecanismo usado por maven para gestionar los proyectos multimódulo recibe el nombre de reactor. Se encarga de “recopilar” todos los módulos, los ordena para construirlos en el orden correcto, y finalmente realiza la construcción de todos los módulos en ese orden. El orden en el que se construyen los módulos es importante, ya que éstos pueden tener dependencias entre ellos.

Mostramos un ejemplo de construcción de nuestro proyecto multimódulo. En este caso vamos a ejecutar el comando:

mvn compile.

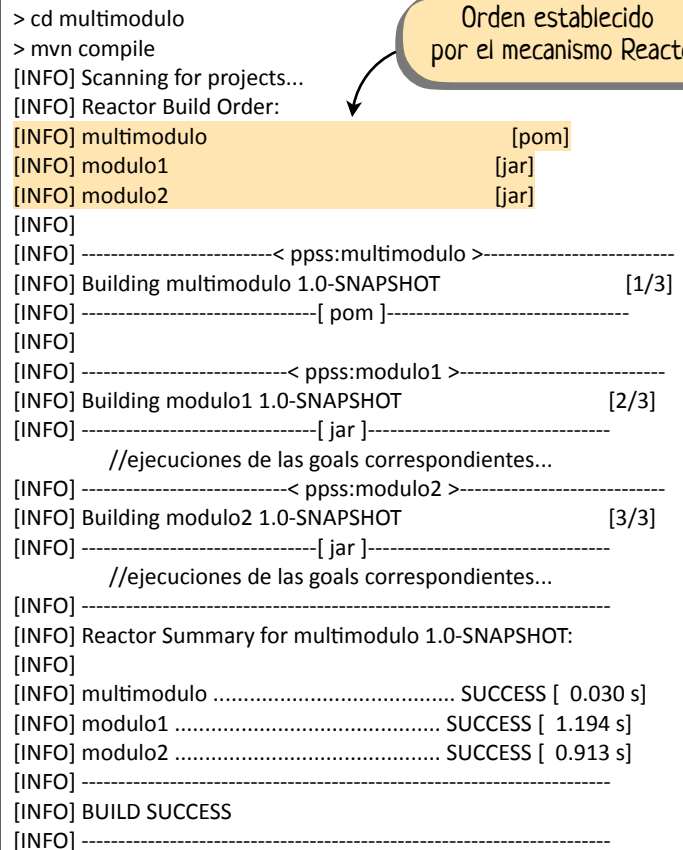
Podemos observar que se ha establecido un orden entre los tres módulos:

- multimodulo,
- modulo1 y
- modulo2.

Y a continuación se ejecuta el comando para todos y cada uno de los módulos en el orden establecido por *reactor*.

Cuando en el empaquetado es pom, por defecto solamente tiene asociadas las goals *install:install*, y *deploy:deploy* a las fases *install* y *deploy*, respectivamente.

Por lo tanto, el proyecto que contiene el aggregator pom, no ejecutará ninguna acción en el resto de fases durante el proceso de construcción.



```
> cd multimodulo
> mvn compile
[INFO] Scanning for projects...
[INFO] Reactor Build Order:
[INFO] multimodulo [pom]
[INFO] modulo1 [jar]
[INFO] modulo2 [jar]
[INFO] -----< ppss:multimodulo >-----
[INFO] Building multimodulo 1.0-SNAPSHOT [1/3]
[INFO] -----[ pom ]-----
[INFO] -----< ppss:modulo1 >-----
[INFO] Building modulo1 1.0-SNAPSHOT [2/3]
[INFO] -----[ jar ]-----
//ejecuciones de las goals correspondientes...
[INFO] -----< ppss:modulo2 >-----
[INFO] Building modulo2 1.0-SNAPSHOT [3/3]
[INFO] -----[ jar ]-----
//ejecuciones de las goals correspondientes...
[INFO] -----
[INFO] Reactor Summary for multimodulo 1.0-SNAPSHOT:
[INFO] multimodulo ..... SUCCESS [ 0.030 s]
[INFO] modulo1 ..... SUCCESS [ 1.194 s]
[INFO] modulo2 ..... SUCCESS [ 0.913 s]
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
```

Beneficios de usar multimódulos

La **ventaja** significativa del uso de **multimódulos** es la **reducción de duplicación**. Por ejemplo, podremos construir todos los módulos con **un único comando** (aplicado al aggregator pom), sin preocuparnos por el orden de construcción, ya que maven tendrá en cuenta las dependencias entre ellos y los ordenará convenientemente.

También podremos “compartir” elementos como **propiedades, dependencias, plugins**, entre los módulos agregados usando el **mecanismo de herencia** que nos proporciona maven.

Maven soporta la relación de herencia, de forma que podemos crear un pom que nos sirva para identificar a un proyecto “padre”. Podemos incluir cualquier configuración en el pom del proyecto padre, de forma que sus módulos “hijo” la hereden, evitando de nuevo duplicaciones.

Siguiendo con nuestro ejemplo, el **proyecto multimodulo será nuestro proyecto “padre”, y sus hijos serán los módulos modulo1 y modulo2**. Para ello, tendremos que referenciar al proyecto “padre” en cada uno de los módulos “hijo” usando la etiqueta **<parent>**.

```
<project ...>
  <parent>
    <artifactId>multimodulo</artifactId>
    <groupId>ppss</groupId>
    <version>1.0-SNAPSHOT</version>
  </parent>
  <artifactId>modulo1</artifactId>
  ...
</project>
```

modulo1/pom.xml

```
<project ...>
  <parent>
    <artifactId>multimodulo</artifactId>
    <groupId>ppss</groupId>
    <version>1.0-SNAPSHOT</version>
  </parent>
  <artifactId>modulo2</artifactId>
  ...
</project>
```

modulo2/pom.xml

En este caso el proyecto maven *multimodulo* es el proyecto “padre. NO tenemos que indicar en el pom “padre” quienes son sus “hijos”, sino que en cada módulo “hijo” añadiremos las coordenadas del proyecto “padre”, del cual heredarán su configuración. Prácticamente se heredan la mayoría de elementos: `<properties>`, `<dependencies>`, `<plugins>`, `<artifactId>`, `<version>`,... Por eso **no es necesario añadir las coordenadas** `<groupId>` y `<version>` en los “hijos” si van a ser las mismas que las del proyecto “padre”. Entre las (pocas) etiquetas que NO se heredan están `<artifactId>` y `<name>`.

Podemos “sobreescribir” cualquier elemento heredado o usar el del proyecto “padre”.

No es necesario usar los mecanismos de herencia y agregación conjuntamente. Es decir, podemos tener únicamente una relación de herencia entre nuestros proyectos maven, o sólo relaciones de agregación, o ambas, como hemos mostrado en nuestro ejemplo.

Obviamente, la reducción de duplicaciones será mayor si usamos a la vez herencia y agregación en nuestros proyectos maven.

Bitbucket

El trabajo de esta sesión también debes subirlo a *Bitbucket*. Todo el trabajo de esta práctica deberá estar en el directorio **P06A-Multimodulo**, dentro de tu espacio de trabajo, es decir, dentro de tu carpeta: `ppss-2021-Gx-apellido1-apellido2..`

Ejercicios

En esta sesión vamos a crear UN único proyecto IntelliJ, el cual contendrá DOS proyectos Maven multimódulo.

En el directorio **Plantillas-P06A**, os proporcionamos las carpetas con los ficheros que usaremos en cada uno de los ejercicios.

Primero crearemos un proyecto IntelliJ **vacío** e iremos añadiendo los módulos (proyectos Maven multimódulo), que correspondan en cada caso.

NOTA: Cuidado: no confundas el proyecto IntelliJ con el proyecto maven, son completamente diferentes. El proyecto IntelliJ es simplemente la carpeta contenedora todo nuestro trabajo. Nuestro "proyecto IntelliJ" puede contener proyectos maven "simples", como hemos usado hasta ahora, o proyectos maven multimódulo, una mezcla de ambos, o incluir además, cualquier otro tipo de proyecto (proyectos NO maven).

Los proyectos contenidos en nuestro proyecto IntelliJ también se llaman módulos, pero no tienen nada que ver con los módulos de un proyecto maven multimódulo! Recordad que los nombres que las herramientas que usamos dan a los diferentes elementos pueden confundiros. Hay que tener claro en todo momento qué estamos haciendo y por qué lo estamos haciendo así y no de otra forma. Cuidado con eso!!!

Para **crear el proyecto IntelliJ**, tendremos que realizar lo siguiente:

- **File→New Project.** A continuación elegimos “Empty Project” y pulsamos sobre Next,
- **Project name :** “**multimodulos**”. **Project Location:** “`$HOME/ppss-2021-Gx-.../P06A-Multimodulo/multimodulos`”.

Fíjate que la carpeta **multimódulos** contiene nuestro **proyecto IntelliJ**.

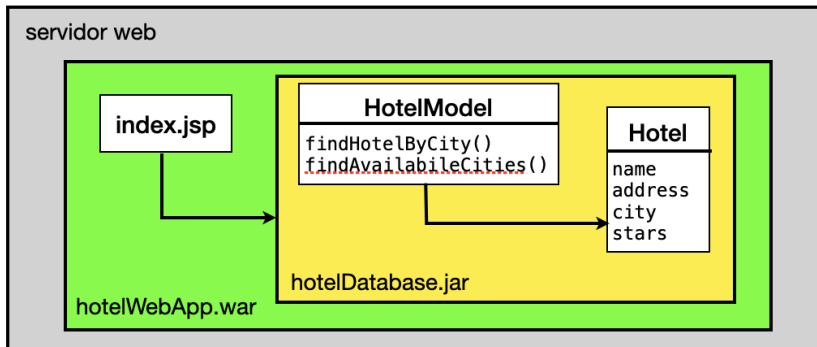
No vamos a añadir todavía ningún módulo, por lo que terminamos el proceso de forma que efectivamente tenemos un proyecto IntelliJ “vacío”.

🔗 Ejercicio 1: proyecto multimódulo *hotel*

Nuestro proyecto multimódulo **hotel** estará formado por:

- una aplicación Web (que empaquetaremos como *hotelWebApp.war*), y
- una aplicación java (que empaquetaremos como *hotelDatabase.jar*).

A continuación mostramos de forma gráfica en la **Figura 1** la relación entre ambas aplicaciones.



La aplicación **hotelWebApp** se ejecutará en un contenedor web de un servidor web o un servidor de aplicaciones. Por eso tendrá un empaquetado **war**.

La aplicación **hotelDatabase** formará parte del fichero war, y tendrá un empaquetado **jar**.

Figura 1. Componentes de la aplicación multimódulo hotel

Ambas aplicaciones son proyectos maven que pueden desarrollarse de forma independiente por diferentes grupos de trabajo. Pero ambos proyectos maven forman parte de la misma aplicación.

Recordemos que hemos creado un proyecto IntelliJ vacío con nombre *multimodulos*.

Vamos a añadir un primer módulo a nuestro proyecto IntelliJ. En la ventana que nos muestra IntelliJ, desde **Project Settings → module**, pulsamos sobre **"→New Module"**:

- Seleccionamos **Maven**, y nos aseguramos de elegir el **JDK 11**
- Seleccionamos **"Create from archetype"**, y buscamos en la lista de arquetipos :
"org.codehaus.mojo.archetypes:pom-root".

Si no aparece en la lista lo añadiremos desde **"Add archetype"** con los siguiente valores:

- ➡ **GroupId**=org.codehaus.mojo.archetypes
- ➡ **ArtifactId**=pom-root
- ➡ **Version**=1.1

Seleccionamos el arquetipo **pom-root** y pulsamos sobre **Next**

Fíjate que hasta aquí, lo único que hemos hecho es darle las indicaciones a IntelliJ para que nos genere de forma automática un proyecto maven multimódulo a partir de una "plantilla" (artefacto pom-root).

Ahora tenemos que indicarle a IntelliJ cuál va ser el nombre y coordenadas de nuestro proyecto maven multimódulo. Usaremos los siguientes valores.

- **Parent:** *<none>*.
- **Name:** *"hotel"*. **Location:** *"\$HOME/ppss-2021-Gx-.../P06A-Multimodulo/multimodulos/hotel"*
- **Artifact-Coordinates → GroupId:** *"ppss"*; **ArtifactId:** *"hotel"*.

Finalmente pulsamos sobre **Next**, y terminamos pulsando sobre **OK**.

Con esto hemos creado el proyecto maven multimódulo **hotel**.

Proyecto
multimódulo
hotel

De forma alternativa, podríamos haber generado el proyecto multimódulo hotel, desde el **terminal**, usando el siguiente comando maven:

```
> cd P06A-Multimodulo/multimodulos
> mvn archetype:generate \
-DarchetypeGroupId=org.codehaus.mojo.archetypes \
-DarchetypeArtifactId=pom-root \
-DarchetypeVersion=1.1 \
-DgroupId=ppss \
-DartifactId=hotel \
-DinteractiveMode=false
```

Nota: Se trata de un único comando maven que podemos teclear en una única línea. Por claridad, lo hemos escrito en varias líneas. Las “\” se usan para indicar en el terminal que ignore el retorno de carro.

En este caso estamos ejecutando la goal “generate” del plugin “archetype”, y necesitamos indicar las coordenadas del arquetipo pom-root, y las coordenadas del proyecto que queremos crear. El arquetipo pom-root nos genera una “plantilla” de un proyecto maven que únicamente contiene un fichero pom.xml con un empaquetado pom.

En realidad nuestro proyecto maven “hotel” todavía no es un proyecto multimódulo. Lo será cuando agreguemos en el pom la lista de módulos.

Ahora vamos a añadir el módulo “hotelDatabase” a nuestro proyecto “hotel”. Para ello seleccionamos el proyecto “hotel” y usamos (**File→New Module**):

- Seleccionamos **Maven**, y nos aseguramos de elegir **JDK 11**, pulsamos **Next**.
- en la ventana **New Module**: **Parent**: hotel
- **Name**: “*hotelDatabase*”. **Location**: “*\$HOME/ppss-2021-Gx-.../P06A-Multimodulo/multimodulos/hotel/hotelDatabase*”
- **Artifact-Coordinates** → **GroupId**: “*ppss*”; **ArtifactId**: “*hotelDatabase*”.

Asegúrate de que en el pom se ha generado la etiqueta <parent>.

El código fuente del módulo **hotelDatabase** está formado por los ficheros *Hotel.java*, y *HotelModel.java* (del directorio *Plantillas-P06A/hotel/*).

Finalmente añadimos el módulo “hotelWebApp” al proyecto “hotel”. Seleccionamos el proyecto “hotel” y usamos (**File→New Module**):

- Seleccionamos **Maven**, y nos aseguramos de elegir **JDK 11**, pulsamos **Next**.
- en la ventana **New Module**: **Parent**: hotel
- **Name**: “*hotelWebApp*”. **Location**: “*\$HOME/ppss-2021-Gx-.../P06A-Multimodulo/multimodulos/hotel/hotelWebApp*”
- **Artifact-Coordinates** → **GroupId**: “*ppss*”; **ArtifactId**: “*hotelWebApp*”.

Asegúrate de que en el pom se ha generado la etiqueta <parent>.

Tendrás que cambiar el empaquetado del proyecto a **war**. Y debes añadir la carpeta **webapp** en el directorio *src/main*.

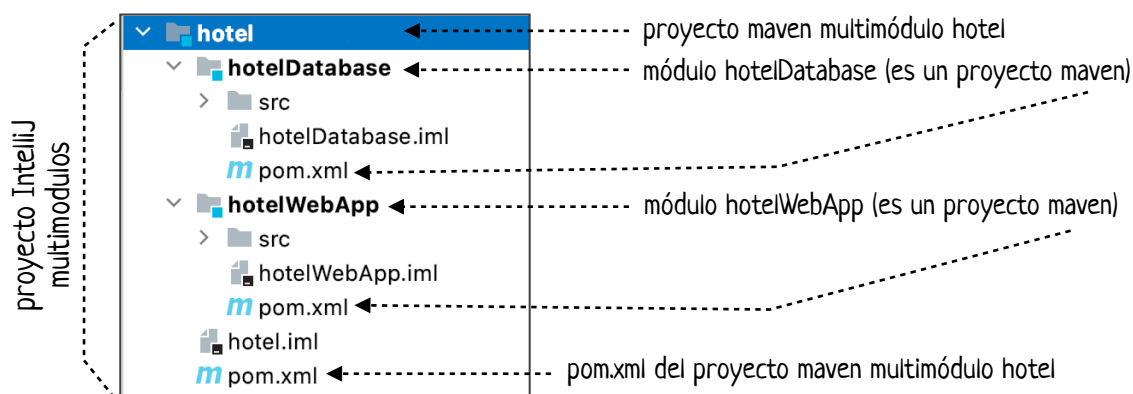
El código fuente del módulo **hotelWebApp** es el fichero *Plantillas-P06A/hotel/index.jsp*, que deberás copiar en el directorio *src/main/webapp*.

En el **pom** del proyecto *hotelWebApp*:

- Añade el plugin *maven-compiler-plugin* con la versión 3.8.0 (cópialo de prácticas anteriores)
- Añade el plugin *maven-war-plugin* con la versión 3.3.1:

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-war-plugin</artifactId>
  <version>3.3.1</version>
  <configuration>
    <failOnMissingWebXml>>false</failOnMissingWebXml>
  </configuration>
</plugin>
```

A continuación mostramos la estructura de tu **proyecto IntelliJ multimódulos** en este momento:



Los proyectos maven **hotelDatabase** y **hotelWebApp** se han añadido como módulos del proyecto maven multimódulo **hotel**. Además, los proyectos **hotelDatabase** y **hotelWebApp** son hijos del proyecto **hotel**. Compruébalo.

Añade en el *pom* del proyecto multimódulo (**hotel**) las propiedades que hemos usado en todos nuestros proyectos maven anteriores y revisa las propiedades de los dos módulos hijo para evitar redundancias.

El módulo **hotelWebApp** “depende” del módulo **hotelDatabase**, tal y como se muestra en la **Figura 1**. Por lo tanto tenemos que añadir esta dependencia en el módulo **hotelWebApp**. Para ello, primero vamos a guardar en nuestro repositorio local el *jar* del módulo **hotelDatabase** (comando *mvn install*, del módulo **HotelDatabase**).

Ahora que ya tenemos el **jar** en nuestro repositorio local, añadiremos dicha dependencia en el *pom* del módulo **hotelWebApp**:

```
<dependency>
  <groupId>${project.groupId}</groupId>
  <artifactId>hotelDatabase</artifactId>
  <version>${project.version}</version>
</dependency>
```

Finalmente, vamos a desplegar nuestra aplicación web y ejecutarla. Necesitamos un servidor de aplicaciones. Lo tenéis comprimido en */home/ppss/Descargas/wildfly-21.0.1.Final.zip*. Podéis en descomprimirlo en vuestro \$HOME. **IMPORTANTE**: NO BORRES el fichero comprimido de la carpeta Descargas!. Lo necesitaremos más adelante.

Añade en el *pom* del proyecto **HotelWebApp** el **plugin wildfly**:

wildfly

```
<plugin>
  <groupId>org.wildfly.plugins</groupId>
  <artifactId>wildfly-maven-plugin</artifactId>
  <version>2.0.2.Final</version>
  <configuration>
    <hostname>localhost</hostname>
    <port>9990</port>
    <jbossHome>/home/ppss/wildfly-21.0.1.Final</jbossHome>
  </configuration>
</plugin>
```

Hemos configurado el plugin indicando la ruta donde tenemos instalado el servidor de aplicaciones. Los valores de *<hostname>* y *<port>* indican en qué máquina y puerto estará “escuchando” nuestro servidor.

Una cosa más: cuando despleguemos nuestro *war* en el servidor, fíjate que el artefacto generado se llamará **hotelWebApp-1.0-SNAPSHOT.war**. Vamos a cambiarle el nombre (para que sólo sea “**hotelWebApp**”), para lo cual añadimos la etiqueta *<finalName>* anidada en la sección *<build>* de nuestro *pom*.

```
<build>
  <!-- Especificamos el nombre del war que será usado como context root
  cuando despleguemos la aplicación -->
  <finalName>${project.artifactId}</finalName>
  <plugins>
    ...
```


Ahora ya podemos desplegar y ejecutar nuestra aplicación web. Para ello tendremos que usar:

- > mvn clean install //fases clean e install desde hotel
- > mvn wildfly:start //arrancamos el servidor de aplicaciones, desde HotelWebApp
- > mvn wildfly:deploy //desplegamos el war generado en el servidor de aplicaciones
- > http://localhost:8080/hotelWebApp //ejecutamos nuestra aplic. web, desde el navegador
- > mvn wildfly:shutdown //cuando queramos detener el servidor

Ejecuta los comandos maven desde la **ventana Maven** de IntelliJ. Verás que, para cada proyecto maven se muestran las fases más comúnmente usadas, los *plugins*,... Para ejecutar sólo las *goals* también puedes hacerlo desde la ventana Maven de IntelliJ (desde los *plugins* de cada proyecto).

Si quieres que se muestren todas las fases del ciclo de vida por defecto (más alguna de los otros dos ciclos), desde la rueda dentada de la ventana Maven, debes desmarcar la opción "Show Basic Phases Only".

🔗 Ejercicio 2: proyecto multimódulo *matriculacion*

En el directorio **Plantillas-P06A/matriculacion** encontraréis los ficheros que vamos a necesitar para crear nuestro proyecto maven multimódulo, al que llamaremos "**matriculacion**".

Vamos a añadir un módulo a nuestro proyecto IntelliJ *multimodulos* (**File→New Module**):

- Seleccionamos **Maven**, y nos aseguramos de elegir el **JDK 11**
- Seleccionamos "**Create from archetype**", y buscamos en la lista de arquetipos :
"org.codehaus.mojo.archetypes:pom-root".

Ahora ya nos debe aparecer en la lista, ya que lo hemos añadido para el ejercicio anterior. Seleccionamos el arquetipo **pom-root** y pulsamos sobre **Next**

Hasta aquí, hemos dado las indicaciones a IntelliJ para que nos genere de forma automática un proyecto maven multimódulo.

Ahora tenemos que indicarle a IntelliJ cuál va ser el nombre y coordenadas de nuestro proyecto maven multimódulo. Usaremos los siguientes valores.

- **Parent:** *<none>*.
- **Name:** "*matriculacion*". **Location:** "*\$HOME/ppss-2021-Gx-.../P06A-Multimodulo/multimodulos/matriculacion*".
- **Artifact-Coordinates → groupId:** "*ppss*"; **ArtifactId:** "*matriculacion*".

Finalmente pulsamos sobre **Next**, y terminamos pulsando sobre **OK**.

Con esto hemos creado el proyecto maven multimódulo **matriculacion**.

Añade al pom creado los valores de las propiedades que hemos usado en todos los ejercicios de prácticas.

El proyecto matriculación será un proyecto multimódulo, formado por los siguientes cuatro módulos:

Nombre del módulo (artifactId)	Paquete que contiene los fuentes	ficheros con los fuentes (carpeta <i>Plantillas-P06A</i>)
matriculacion-comun	ppss.matriculacion.to	en matriculacion/to
matriculacion-dao	ppss.matriculacion.dao	en matriculacion/dao
matriculacion-proxy	ppss.matriculacion.proxy	en matriculacion/proxy
matriculacion-bo	ppss.matriculacion.bo	en matriculacion/bo

Cada uno de los módulos, además, serán "hijos" de *matriculacion*.

Proyecto multimódulo matriculacion

Crea los módulos tal y como hemos explicado en el ejercicio anterior (recuerda que físicamente deberán ser subdirectorios del proyecto padre). Usa los ficheros de las plantillas para añadir el código fuente de cada uno de los módulos.

Nota: IntelliJ te marcará como errores todos los usos de los objetos *TO. No te preocupes. Se solucionarán cuando incluyamos las dependencias entre los módulos.

Modifica el pom de cada módulo para incluir las **dependencias** reflejadas en la **Figura 2** (por ejemplo BO → PROXY, se “lee” como “el módulo matriculacion-bo depende del módulo matriculacion-proxy”).

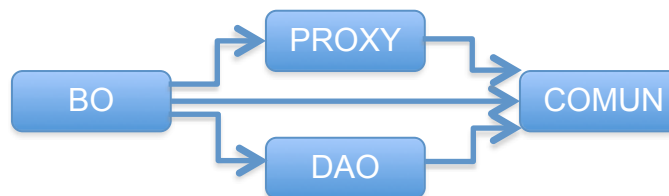


Figura 2. Dependencias para el proyecto *matriculación*

En la **Figura 3** mostramos la vista *Project* de nuestro proyecto **multimodulos**, que contiene dos proyectos maven multimódulo: *hotel* y *matriculacion*

El proyecto **matriculacion** es un proyecto multimódulo (tiene empaquetado pom), y contiene los 4 módulos (proyectos maven) de la Figura 2.

Puesto que hemos "agregado" los cuatro módulos en el pom del proyecto *matriculacion*, cuando ejecutemos un comando maven desde el directorio *matriculacion*, dicho comando se ejecutará para todos los módulos agregados.

No tendremos que preocuparnos de las dependencias entre los módulos gracias al mecanismo reactor de maven, que los ordenará convenientemente.

Es conveniente que tengas esta vista, tal cual se muestra en la Figura 3, para que veas más fácilmente lo que ocurre cuando construimos el proyecto *matriculacion*.

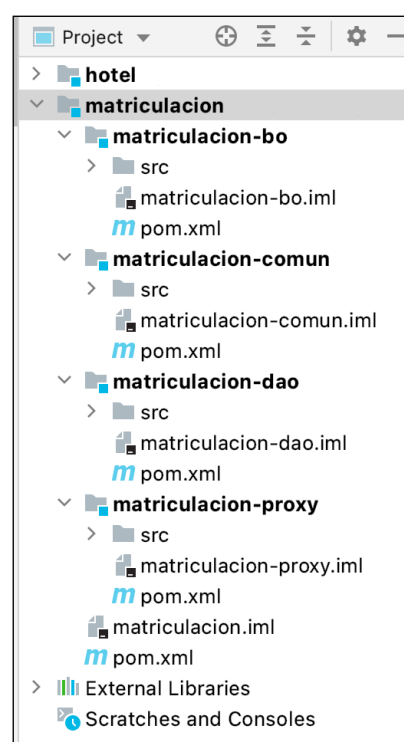


Figura 3. Vista *Project* del proyecto *multimodulos*

Finalmente vamos a construir el proyecto. Realiza las siguientes acciones:

- ejecuta la fase **compile** desde el proyecto matriculacion: verás que en la vista *Project* aparece el directorio target en todos los módulos del proyecto.
- en la vista *Project*, muestra el contenido de los directorios *target* de los módulos de matriculacion. Y a continuación ejecuta la fase **package** desde ese proyecto. Verás que en el target se generan los .jar de cada módulo.
- si ahora ejecutas la fase **clean** desde el proyecto matriculacion, verás que se borran todos los "targets" de los módulos

Cada uno de los módulos, puede construirse por separado, vamos a comprobarlo:

- ejecuta la fase **compile** desde el proyecto *matriculacion-comun*: verás que en la vista *Project* aparece el directorio *target* sólo en ese proyecto.
- ahora ejecuta la misma fase desde el proyecto *matriculacion-dao*. Verás que no compila!!! Al estar ejecutando los proyectos por separado, fíjate que *matriculacion-dao* depende de *matriculacion-comun*. Lo deberías tener indicado en el pom en forma de dependencia. Por lo tanto, maven, busca el artefacto de *matriculacion-comun* en nuestro repositorio local. Como no lo encuentra, intentará buscar en la nube, y tampoco lo va a encontrar, y finalmente al compilar mostrar un error al compilar, ya que en el código fuente de *matriculacion-dao* se usan las clases de *matriculacion-comun*.

Tendrás que ejecutar la fase **install** desde *matriculacion* para que se generen todos los artefactos de cada módulo y se guarden en nuestro repositorio local. Comprueba que están ahí.

Una vez hecho esto, puedes, por ejemplo, ejecutar de nuevo **clean** sobre el proyecto *matriculacion*. Y a continuación vuelve a ejecutar la fase **compile** desde el proyecto *matriculacion-dao*. Ahora debe compilar sin problema.

Observa la salida por pantalla de maven y entiende bien por qué se realizan las acciones de construcción en ese orden. Recuerda que debes tener claro qué ficheros y/o artefactos se generan en cada una de las fases ejecutadas.

Necesitarás este proyecto multimódulo para la siguientes sesión de prácticas en la que añadiremos y ejecutaremos tests unitarios y de integración.

Resumen



¿Qué **conceptos** y **cuestiones** me deben quedar CLAROS después de hacer la práctica?



PROYECTOS INTELLIJ

- Un proyecto IntelliJ puede contener diferentes tipos de proyectos: proyectos maven, java, android, servicios rest, servicios web..., cada uno de ellos se denomina **MÓDULO**.
- Un proyecto IntelliJ no se construye ya que no contiene código, el proceso de construcción se realiza para cada uno de sus módulos.

PROYECTOS MAVEN MULTIMÓDULO

- Un proyecto **MAVEN** puede ser "single" (single-maven project) o puede contener, a su vez, varios proyectos maven. (multimodule maven project), cada uno de los cuales se denomina **MODULO**.
- El proyecto multimódulo tiene un empaquetado pom, en el que se pueden **AGREGAR** tantos módulos (proyectos maven) como se necesite. El mecanismo de agregación permite ejecutar un comando maven (una vez), y se ejecutará automáticamente en todos los módulos agregados, en el orden determinado por las dependencias entre ellos, e identificado por el mecanismo **REACTOR** de maven.
- Se pueden establecer relaciones de **HERENCIA** entre los módulos de un proyecto multimódulo, de esta forma pueden compartir propiedades, plugins, y dependencias.
- En un proyecto multimódulo se pueden usar ambas relaciones (herencia y agregación) o sólo una de ellas.
- Los módulos de un proyecto multimódulo pueden construirse de forma individual: un módulo es un proyecto maven con un empaquetado diferente de pom: puede ser jar, war, ear.

P06B- Pruebas de integración

Os recordamos que el plazo para entregar los ejercicios de esta práctica (P06B) termina justo ANTES de comenzar la siguiente sesión de prácticas (P07) en el laboratorio (los grupos de los lunes, por lo tanto, el próximo lunes, los de los martes, el próximo martes, ...). Pasado este plazo los profesores no revisarán vuestro trabajo sobre P06B (de forma individual), con independencia de que se indiquen en clase las soluciones que os permitirán saber si las vuestras son correctas.

Integración con una base de datos

El objetivo de esta práctica es automatizar pruebas de integración con una base de datos. Para controlar el estado de la base de datos durante los tests utilizaremos la librería DbUnit.

Recordamos que es muy importante, no solamente la implementación de los drivers, sino también dónde se sitúan físicamente dichos drivers en un proyecto maven, cómo configurar correctamente el pom y cómo ejecutar los drivers desde maven, para así obtener el informe de pruebas de forma automática. Al fin y al cabo, lo que buscamos es obtener ese informe que nos permita responder a la pregunta inicial: "¿en nuestro SUT hay algún defecto?"

En este caso, implementaremos drivers con verificación basada en el estado. En las pruebas de integración ya no estamos interesados en detectar defectos en las unidades, por lo tanto la cuestión fundamental no es tener el control de las dependencias externas (una de las cuales será la base de datos, o mejor dicho, el servidor de la base de datos). Precisamente queremos ejercitar dichas dependencias externas durante las pruebas. Nuestro objetivo es detectar defectos en las "interfaces" de las unidades que forman nuestro SUT. Dichas unidades ya han sido probadas previamente, de forma individual, y por lo tanto, ya hemos detectado defectos "dentro" de las mismas. Es decir, buscamos potenciales problemas en los puntos de "interconexión" de dichas unidades. En definitiva, la cuestión fundamental a contemplar en las pruebas de integración es el orden en el que vamos a integrar las unidades (estrategias de integración).

Dado que no podemos hacer pruebas exhaustivas, es perfectamente posible que durante las pruebas de integración "salgan a la luz" defectos "dentro" de las unidades correspondientes, es decir, que un defecto que se nos "había pasado por alto" durante las pruebas unitarias, "se manifieste" ahora. Obviamente, habrá que depurarlo, pero debes tener muy claro que nuestro objetivo no es ese.

En la sesión S01, en la primera definición de *testing*, se indica de forma explícita que la "intención" con la que probamos es fundamental para conseguir nuestro objetivo, lo cual es totalmente lógico. Por lo tanto, cuando realizamos pruebas de integración, y dado que el objetivo es diferente, el conjunto de casos de prueba obtenidos también será diferente (elegiremos conjuntos diferentes de comportamientos a probar), puesto que nuestros esfuerzos estarán centrados en evidenciar problemas en "líneas de código diferentes"!!! En este caso, como hemos visto en la sesión de teoría, usaremos métodos de diseño de casos de prueba de caja negra, teniendo en cuenta las guías generales de pruebas.

Como siempre decimos: si no se tiene claro QUÉ queremos conseguir (qué problema concreto queremos que resolver), es improbable que las acciones que realicemos (el CÓMO) sean las adecuadas para solucionar dicho problema de la mejor forma y lo más eficientemente posible.

En esta práctica nos vamos a centrar en el problema de la integración con una base de datos.

Para implementar los drivers usaremos JUnit5 y Dbunit. Para ejecutarlos usaremos Maven, y Junit5.

Bitbucket

El trabajo de esta sesión también debes subirlo a *Bitbucket*. Todo el trabajo de esta práctica deberá estar en el directorio **P06B-Integracion**, dentro de tu espacio de trabajo, es decir, dentro de tu carpeta: ppss-2021-Gx-apellido1-apellido2.

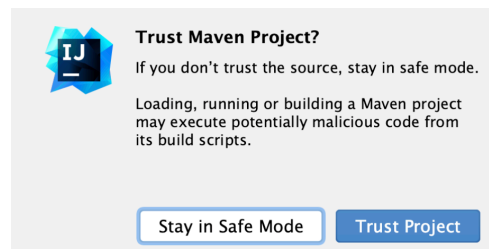
IntelliJ versión 2021.1

Observaréis que IntelliJ se actualiza automáticamente a la última versión en la máquina virtual (ahora mismo deberéis ver la versión 2021.1).

Es posible que os aparezca una ventana emergente como la que mostramos a la derecha.

Debes seleccionar "*Trust Project*", para que IntelliJ siga funcionando tal y como venía haciéndolo hasta ahora.

También es posible que os aparezca el siguiente aviso en el editor de IntelliJ.



Safe mode, limited functionality. Trust the project to access full IDE function... [Trust project...](#) [Read more](#)

En este caso, igualmente debéis seleccionar "*Trust project*".

Base de datos MySQL

En la máquina virtual tenéis instalado un servidor de bases de datos MySQL, al que podéis acceder con el usuario "**root**" y password "**ppss**" a través del siguiente comando, desde el terminal:

```
> mysql -u root -p (Para salir, usa el comando exit;)
```

Dicho servidor se pone en marcha automáticamente cuando arrancamos la máquina virtual.

Vamos a usar un **driver jdbc** para acceder a nuestra base de datos. Por lo tanto, desde el código, usaremos la siguiente cadena de conexión:

```
jdbc:mysql://localhost:3306/DBUNIT?useSSL=false
```

En donde *localhost* es la máquina donde se está ejecutando el servidor mysql. Dicho proceso se encuentra "escuchando" en el puerto 3306, y queremos acceder al esquema *DBUNIT* (el cual crearemos automáticamente durante el proceso de construcción usando el plugin sql).

Ejecución de scripts sql con Maven: plugin *sql-maven-plugin*

Hemos visto en clase que se puede utilizar un plugin para maven (*sql-maven-plugin*) para ejecutar scripts *sql* sobre una base de datos.

El plugin ***sql-maven-plugin*** requiere incluir como dependencia el conector con la base de datos (ya que podemos querer ejecutar el script sobre diferentes tipos de base de datos). En la sección `<configuration>` tenemos que indicar el driver, la cadena de conexión, login y password, para poder acceder a la BD y ejecutar el script. Además podemos configurar diferentes *ejecuciones* de las goals del plugin (cada una de ellas estará en una etiqueta `<execution>`, y tendrá un valor de `<id>` diferente). Para ejecutar cada una de ellas podemos hacerlo de diferentes formas, teniendo en cuenta que:

- ❖ Si hay varias `<execution>` asociadas a la misma fase y ejecutamos dicha fase maven, se ejecutarán todas ellas
- ❖ Si queremos ejecutar solamente una de las `<executions>`, podemos hacerlo con `mvn sql:execute@execution-id`, siendo `execution-id` el identificador de la etiqueta `<id>` de la `<execution>` correspondiente
- ❖ Si ejecutamos simplemente `mvn sql:execute`, sin indicar ninguna ejecución en concreto, por defecto se ejecutará aquella que esté identificada como `<id>default-cli</id>`. Si no hay ninguna `<execution>` con ese nombre, no se ejecutará nada.

Ejemplo: Dado el siguiente fragmento de código con las <executions> del plugin,

```
...
<executions>
  <execution>
    <id>create-customer-table</id>
    <phase>pre-integration-test</phase>
    <goals>
      <goal>execute</goal>
    </goals>
    <configuration>
      <srcFiles>
        <srcFile>src/test/resources/sql/script1.sql</srcFile>
      </srcFiles>
    </configuration>
  </execution>
  <execution>
    <id>create-customer-table2</id>
    <phase>pre-integration-test</phase>
    <goals>
      <goal>execute</goal>
    </goals>
    <configuration>
      <srcFiles>
        <srcFile>src/test/resources/sql/script2.sql</srcFile>
      </srcFiles>
    </configuration>
  </execution>
</executions>
...
```

- ❖ La ejecución **mvn pre-integration-test** ejecutará los scripts script1.sql y script2.sql
- ❖ La ejecución **mvn sql:execute@create-customer-table2** solamente ejecutará el script2.sql
- ❖ La ejecución **mvn sql:execute** no ejecutará nada

Acceso a una base de datos desde IntelliJ

Podemos acceder a nuestro servidor de base de datos MySQL instalado en la máquina virtual, a través de IntelliJ, en concreto, desde la vista "**Database**" (Si no aparece en el lateral, se puede abrir desde el menú de IntelliJ *View→Tool Windows→Database*).

Desde la ventana *Database*, creamos una nueva "conexión" desde "+ → Data Source → MySQL".

A continuación proporcionamos los valores:

- ❖ **Name:** por defecto se muestra @localhost (puedes dejar este nombre o cambiarlo)
- ❖ **Host:** localhost, **Port:** 3306
- ❖ **User:** root, **Password:** ppss, URL: jdbc:mysql://localhost:3306
- ❖ Pulsa sobre "**Test Connection**" para asegurarte de que funciona la conexión (ver ****Nota**)
- ❖ Finalmente pulsamos sobre **OK**.

Ahora ya puedes ver los esquemas de base de datos creadas en el servidor MySQL, e interactuar con él a través de IntelliJ.

Pulsando sobre el segundo icono empezando por la derecha desde esa ventana (Jump To Query Console...), se abre un editor con el que podemos realizar queries sobre la base de datos. Por ejemplo, para ver el contenido de una tabla de la base de datos:

```
USE nombreEsquema;          <---- Nos situamos en el esquema correspondiente
SELECT * FROM nombreTabla; <---- Recuperamos los datos de una tabla ordenados por clave primaria
```

****Nota:** Si al probar la conexión aparece el mensaje: "*Failed. Server returns invalid timezone. Need to set 'serverTimezone' property*", pulsa sobre *Set Time Zone*, asegúrate de que se muestra la propiedad serverTimezone con el valor UTC, y selecciona OK. Intenta nuevamente probar la conexión y comprueba que funciona correctamente.

Ejercicios

En esta sesión trabajaremos con un proyecto IntelliJ **vacío**, e iremos añadiendo los módulos (proyectos Maven), en cada uno de los ejercicios.

Para **crear el proyecto IntelliJ**, simplemente tendremos que realizar lo siguiente:

- **File→New Project**. A continuación elegimos "Empty Project" y pulsamos sobre Next,
- **Project name** : "*dbunit*". **Project Location**: "*\$HOME/ppss-2020-Gx-.../P06B-Integracion/dbunit*".

De forma automática, IntelliJ nos da la posibilidad de añadir un nuevo módulo a nuestro proyecto (desde la ventana **Project Structure**), antes de crear el proyecto. Cada ejercicio lo haremos en un módulo diferente. Recuerda que CADA módulo es un PROYECTO MAVEN.

En ese caso, la carpeta **dbunit** contiene nuestro **proyecto IntelliJ**.

No vamos a añadir todavía ningún módulo, por lo que terminamos el proceso de forma que efectivamente tenemos un proyecto IntelliJ "vacío".

🔗 Ejercicio 1: proyecto dbunitexample

Añade un MÓDULO a nuestro proyecto IntelliJ *dbunit* (**File→New Module**):

- Seleccionamos **Maven**, y nos aseguramos de elegir el **JDK 11**
- **GroupId**: "*ppss*"; **ArtifactId**: "*dbunitexample*".
- **ModuleName**: "*dbunitexample*". **Content Root**: "*\$HOME/ppss-2021-Gx-.../P06B-Integracion/dbunit/dbunitexample*". **Module file location**: "*\$HOME/ppss-2021-Gx-.../P06B-Integracion/dbunit/dbunitexample*".

El proyecto que acabamos de crear es un *single maven project*. Como siempre, necesitamos incluir las `<properties>` en el pom, que copiaremos de proyectos anteriores. Además necesitaremos añadir como dependencias la librería junit5, más las nuevas dependencias que hemos visto en clase.

Con respecto a los plugins, recuerda que tienes que añadir el compilador de java (versión 3.8.0), y los nuevos plugins que hemos visto en clase para esta sesión.

En el directorio Plantillas-P06B/ejercicio1 encontraréis los ficheros que vamos a usar en este ejercicio:

- ❖ la implementación de las clases sobre las que realizaremos las pruebas: *ppss.ClienteDAO*, *ppss.Ciiente* y *ppss.IClienteDAO*,
- ❖ la implementación de dos tests de integración (*ClienteDAO_IT.java*), más una clase adicional para indicar a DbUnit que vamos a trabajar con una base de datos MySQL y no nos muestre ningún *warning* al acceder a la base de datos (*MijdbcDatabaseTester.java*). Esta clase adicional estará en el mismo directorio que los tests.
- ❖ ficheros xml con los datos iniciales de la BD (*cliente-init.xml*), y resultado esperado de uno de los tests (*cliente-esperado.xml*),
- ❖ fichero con el script sql para restaurar el esquema y las tablas de la base de datos (*create-table-customer.sql*)

Tendrás que crear la carpeta **resources** en *src/test/*. Para que IntelliJ la "vea" como la carpeta de recursos de maven, selecciona "**Mark directory as →Test Resources Root**" desde el menú contextual de la carpeta "*src/test/resources*".

Además, crea un nuevo *DataSource* asociado al servidor de bases de datos MySQL desde la ventana Database.

Se pide:

- A) Organiza el código proporcionado en el proyecto Maven que has creado según hemos visto en clase. Revisa el pom para que contenga las dependencias y plugins correctos, y añade, además, la siguiente librería de *logging*. Solamente la incluimos para que no nos aparezcan advertencias (*warnings*) al construir el proyecto con Maven.

```
<!-- Si no la incluimos veremos un warning al ejecutar los tests -->
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-simple</artifactId>
  <version>1.7.30</version>
</dependency>
```

Nota: Asegúrate de que la cadena de conexión del **plugin sql** NO contenga el esquema de nuestra base de datos DBUNIT, ya que precisamente el plugin sql es el que se encargará de crearlo cada vez que construyamos el proyecto. Por lo tanto, la cadena de conexión que usa el plugin sql debe ser: *jdbc:mysql://localhost:3306/?useSSL=false*. Cuando uses la cadena de conexión en el código del proyecto ésta SÍ deberá incluir el esquema DBUNIT.

La clase que vamos a usar que representa el driver para acceder a la base de datos es: *com.mysql.cj.jdbc.Driver*. Recuerda que tenemos que acceder a la base de datos (y por lo tanto usar este driver) tanto en *src/main* como en *src/test*.

Aunque proporcionamos ya el código de dos tests de integración, debes tener claro el uso del API *dbunit* y qué es lo que estamos probando en los tests.

- B) Utiliza la ventana Maven para ejecutar los tests de integración. Debes asegurarte previamente de que tanto la cadena de conexión, como el login y el password sean correctos, tanto en el *pom.xml*, como en el código de pruebas y en el código fuente. Observa la consola que muestra la salida de la construcción del proyecto y comprueba que se ejecutan todas las goals que hemos indicado en clase. Fíjate bien en los ficheros y artefactos generados.

Verás que aparecen varios *warnings*. Para que esto no ocurra debemos indicar a DbUnit el tipo de base de datos al que estamos accediendo. Usa la clase *MysqlDatabaseTester* proporcionada, en lugar de la clase *JdbcDatabaseTester* que hemos explicado en clase.

Puedes comprobar en la ventana Database de IntelliJ que se ha creado el esquema DBUNIT con una tabla *cliente*. Para ello tendrás que "refrescar" la vista Database, pulsando sobre el tercer icono empezando por la izquierda (icono *Refresh*).

- C) Una vez ejecutados los tests, cambia los datos de prueba del método *test_insert()* y realiza las modificaciones necesarias adicionales para que los tests sigan en "verde". Puedes probar a realizar cambios en los datos de entrada del otro test (*test_delete()*) para familiarizarte con el código.
- D) Implementa dos tests adicionales. Un test para actualizar los datos de un cliente (*testUpdate()*) y otro para recuperar los datos de un cliente (*testRetrieve()*). Utiliza los siguientes casos de prueba:

Tabla cliente inicial	Datos del cliente a modificar	Resultado esperado (tabla)
id =1 nombre="John" apellido="Smith" direccion = "1 Main Street" ciudad = "Anycity"	id =1 nombre= "John" apellido= "Smith" direccion = "Other Street" ciudad = "NewCity"	id =1 nombre = "John" apellido= "Smith" direccion = "Other Street" ciudad = "NewCity"

Tabla cliente inicial	id del cliente a recuperar	Resultado esperado
id =1 nombre = "John" apellido = "Smith" direccion = "1 Main Street" ciudad = "Anycity"	id =1	id =1 nombre = "John" apellido = "Smith" direccion = "1 Main Street" ciudad = "Anycity"

🔗 Ejercicio 2: proyecto multimódulo matriculacion

Vamos a añadir un segundo módulo a nuestro proyecto IntelliJ *dbunit*. Usaremos el proyecto multimódulo "*matriculacion*" de la práctica anterior (P06A).

Copia, desde un terminal, el proyecto maven multimódulo "*matriculacion*" de la práctica anterior como subdirectorío de la carpeta-*dbunit* (proyecto IntelliJ).

Ahora añadiremos el proyecto *matriculación* como un nuevo módulo a nuestro proyecto IntelliJ *dbunit* (**File→New→Module from Existing Sources...**):

- Seleccionamos la carpeta *matriculacion* que acabamos de copiar.
- En la siguiente pantalla seleccionamos: **Import module from external model→Maven**

IntelliJ ha importado el proyecto *matriculacion* como un módulo, pero no es "consciente" de que es un proyecto multimódulo, y que por lo tanto contiene otros módulos. No es algo importante, puesto que no dependemos de IntelliJ para construir el proyecto. Pero si no indicamos a IntelliJ que cada subcarpeta que contiene un fichero pom.xml es otro proyecto maven, es posible que IntelliJ comience a identificar errores que realmente no lo son. Y además, "perdemos" la utilidad de la vista Project.

Ve a File → Project Structure → Modules. Verás que en la lista de módulos IntelliJ sólo contempla dos: *dbunitexample* y *matriculacion*.

Vamos a añadir los módulos que faltan. Para ello, desde + -> Import Module, seleccionaremos, uno a uno, los proyectos maven *matriculacion-comun*, *matriculacion-dao*, *matriculacion-proxy* y *matriculacion-bo*, indicando que son proyectos maven.

Nos queda una última cosa por hacer: marcar los directorios de fuentes, de pruebas y de recursos, para que IntelliJ los muestre convenientemente en la vista projects.

Para ello, seleccionamos uno de los módulos de la lista de módulos (desde File → Project Structure → Modules) , por ejemplo, *matriculacion-dao*, y tenemos que marcar las carpetas *src/main/java*, *src/test/java*, y *src/main/resources* como **sources**, **Tests** y **Resources**, respectivamente.

Esto lo tenemos que hacer con los cuatro módulos que hemos añadido.

La **Figura 1** muestra la lista de módulos y el marcado de directorios para el proyecto *matriculacion-dao*

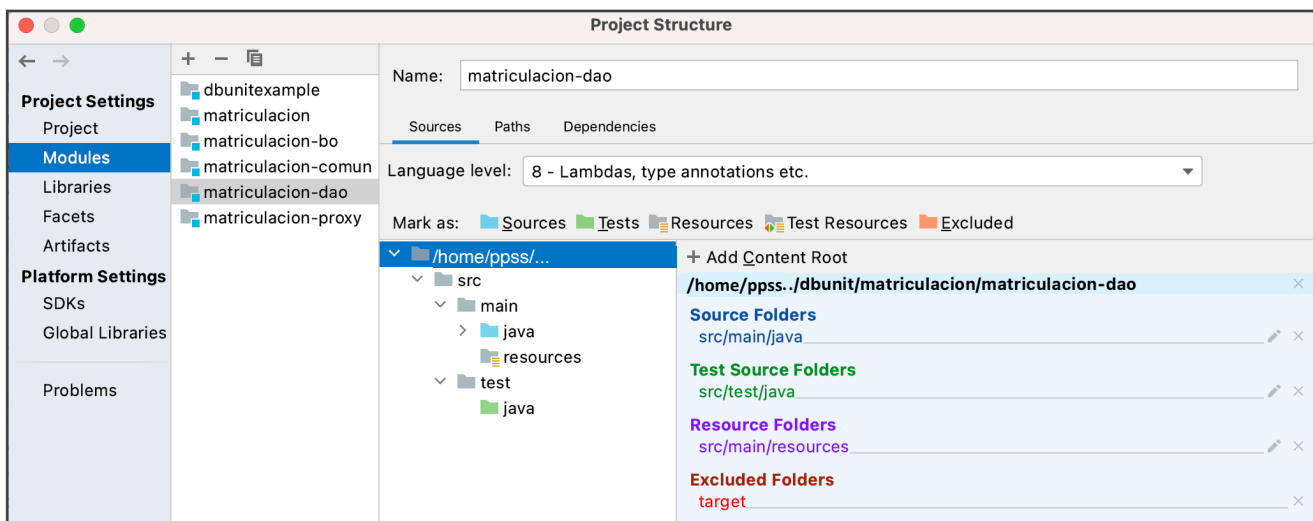
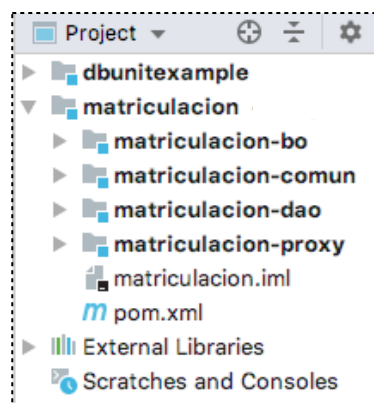


Figura 1. Lista de módulos del proyecto *dbunit*. Los módulos *matriculacion-xxx* son proyectos maven

Finalmente debemos ver nuestro proyecto *matriculacion* como un módulo en la vista Project de IntelliJ (ver imagen de la derecha)



Vista Project del proyecto IntelliJ dbunit

En el módulo *matriculacion-dao*, crea la carpeta **resources** en *src/test/*. Para que IntelliJ la "vea" como la carpeta de recursos de maven, selecciona "**Mark directory as → Test Resources Root**" desde el menú contextual de la carpeta "*src/test/resources*".

En el directorio Plantillas-P06B/ejercicio2 encontraréis los ficheros que vamos a usar en este ejercicio:

- ❖ fichero con el *script* sql para restaurar el esquema y las tablas de la base de datos (*matriculacion.sql*)
- ❖ fichero xml con un *dataset* ejemplo (*dataset.xml*) y fichero dtd con la gramática de los ficheros XML para nuestro esquema de base de datos (*matriculacion.dtd*)

Se pide:

- A) Dado que hay una relación de herencia entre *matriculación* y sus módulos agregados, vamos a modificar el **pom de matriculacion**, para añadir aquellos elementos comunes a los submódulos. Concretamente, estos deben heredar lo siguiente: la librería para usar *junit5*, y los *plugins compiler, surefire y failsafe*. Deberías tener claro para qué sirven cada uno de ellos.
- B) Incluye en el **pom del módulo matriculacion-dao** las librerías *dbunit, slf4j-simple*, y *mysql-connector-java* (igual que en el ejercicio anterior). Debes añadir también el plugin *sql-maven-plugin*. El *script* sql será el fichero *matriculacion.sql* que encontrarás en el directorio de plantillas (deberás copiarlo en el directorio *src/test/resources/sql*). Acuérdate de marcar la carpeta *resources* como "**Test Resources Root**":
- C) En la clase **FuenteDatosJDBC** (en *src/main/java* del módulo *matriculacion-dao*) se configura la conexión con la BD. Debes cambiar el driver *hsqldb* por el driver *jdbc* (*com.mysql.cj.jdbc.Driver*), así como los parámetros del método *getConnection*, es decir, la cadena de conexión, login y password (debes usar los valores que ya hemos usado para el ejercicio anterior).
- D) Copia el fichero **matriculacion.dtd** del directorio de plantillas en *src/test/resources* (del módulo *matriculacion-dao*). El fichero **dataset.xml** tiene un ejemplo de cómo definir nuestros datasets en formato xml. Puedes copiar también dicho fichero en *src/test/resources*.
- E) Queremos implementar tests de integración para los métodos *AlumnoDAO.addAlumno()* y *AlumnoDAO.delAlumno()*. Para ello vamos a necesitar crear los ficheros **tabla2.xml, tabla3.xml, y tabla4.xml**, que contienen los **datasets iniciales** y **datasets esperados** necesarios para implementar los casos de prueba de la **tabla 1** (dichos datasets se definen en las **tablas 2, 3 y 4**). Crea los ficheros xml correspondientes en la carpeta *src/test/resources*, con la opción *New→File* (desde el directorio *src/test/resources*), indicando el nombre del fichero: por ejemplo *tabla2.xml*. Puedes copiar el contenido de *dataset.xml* y editar los datos convenientemente.

Tabla 2. Base de datos de entrada. Contenido tabla ALUMNOS

NIF	Nombre	Dirección	E-mail	Fecha nacimiento
11111111A	Alfonso Ramirez Ruiz	Rambla, 22	alfonso@ppss.ua.es	1982-02-22
22222222B	Laura Martinez Perez	Maisonnave, 5	laura@ppss.ua.es	1980-02-22

Tabla 3. Base de datos esperada como salida en testA1

NIF	Nombre	Dirección	E-mail	Fecha nacimiento
11111111A	Alfonso Ramirez Ruiz	Rambla, 22	alfonso@ppss.ua.es	1982-02-22
22222222B	Laura Martinez Perez	Maisonnave, 5	laura@ppss.ua.es	1980-02-22
33333333C	Elena Aguirre Juarez			1985-02-22

Tabla 4. Base de datos esperada como salida en testB1

NIF	Nombre	Dirección	E-mail	Fecha nacimiento
22222222B	Laura Martinez Perez	Maisonnave, 5	laura@ppss.ua.es	1980-02-22

F) **Implementa**, usando *DbUnit*, los casos de prueba de la tabla 1 en una clase *AlumnoDAOIT*. Cada caso de prueba debe llevar el nombre indicado en la columna ID de la tabla 1.

Para hacer las pruebas, tendremos que utilizar un objeto que implemente la interfaz *IAlumnoDAO*. Así, por ejemplo para cada test Ax de la tabla 1, utilizaremos una sentencia del tipo:

```
new FactoriaDAO().getAlumnoDAO().addAlumno(alumno);
```

Para especificar la fecha, debes utilizar la clase *LocalDate*. Usa el constructor *LocalDate.of()* tal y como ya hemos hecho en prácticas anteriores.

Nota: en los drivers, deberías usar la clase *MiJdbcDatabaseTester* (igual que hemos hecho en el ejercicio anterior) para evitar los *warnings* al ejecutar los tests.

Tabla 1. Casos de prueba para *AlumnoDAO*

ID	Método a probar	Entrada	Salida Esperada
testA1	void addAlumno (AlumnoTO p)	p.nif = "33333333C" p.nombre = "Elena Aguirre Juarez" p.fechaNac = 1985-02-22	Tabla 3
testA2	void addAlumno (AlumnoTO p)	p.nif = "11111111A" p.nombre = "Alfonso Ramirez Ruiz" p.fechaNac = 1982-02-22	DAOException (1)
testA3	void addAlumno (AlumnoTO p)	p.nif = "44444444D" p.nombre = null p.fechaNac = 1982-02-22	DAOException (1)
testA4	void addAlumno (AlumnoTO p)	p = null	DAOException (2)
testA5	void addAlumno (AlumnoTO p)	p.nif = null p.nombre = "Pedro Garcia Lopez" p.fechaNac = 1982-02-22	DAOException (1)
testB1	void delAlumno (String nif)	nif = "11111111A"	Tabla 4
testB2	void delAlumno (String nif)	nif = "33333333C"	DAOException (3)

DAOException(1) = mensaje "Error al conectar con BD"

DAOException(2)= mensaje "Alumno nulo"

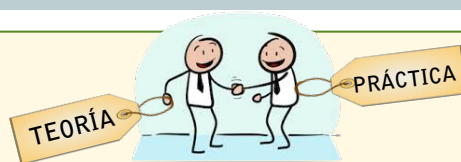
DAOException(3)= mensaje "No se ha borrado ningun alumno"

- G) **Ejecuta los tests** anteriores teniendo en cuenta que si algún test falla debemos interrumpir la construcción del proyecto. El test "testA4" falla. Repara el error en `src/main` de forma que devuelva el resultado correcto.
- Nota:** para ver el nuevo esquema creado, no es suficiente con refrescar el DataSource de la ventana Database. Edita el DataSource, y en la pestaña "**Schemas**", marca el nuevo esquema (matriculacion).
- H) Añade 3 **tests unitarios** en una clase `AlumnoDAOTest`. NO es necesario que los implementes, pueden contener simplemente un `assertTrue(true)`. Ejecuta sólo los tests unitarios usando el comando `mvn test` y comprueba que efectivamente sólo se lanzan los tests unitarios
- I) Añade también tests unitarios y de integración en los proyectos *matriculacion-bo* y *matriculacion-proxy*. Luego ejecuta `mvn verify` desde el proyecto padre. Anota la secuencia en la que se ejecutan los diferentes tipos de tests de todos los módulos. Deberías saber que estrategia de integración estamos siguiendo.
- J) Finalmente añade un nuevo **elemento de configuración**, dentro del proyecto *matriculacion* con el comando maven "`mvn verify -Dgroups="Integracion-fase1"`". Dado que los plugins *failsafe* y *surefire* comparten la variable *groups*, etiqueta tanto los tests unitarios como los de integración del proyecto *matriculacion-dao* con la etiqueta "*Integracion-fase1*". Ejecútala y comprueba que ahora sólo se ejecutan los tests unitarios y de integración del proyecto *matriculacion-dao*.

Resumen



¿Qué **conceptos** y **cuestiones** me deben quedar CLAROS después de hacer la práctica?



PRUEBAS DE INTEGRACIÓN

- Nuestro SUT estará formado por un conjunto de unidades. El objetivo principal es detectar defectos en las INTERFACES de dicho conjunto de unidades. Recuerda que dichas unidades ya habrán sido probadas individualmente
- Son pruebas dinámicas (requieren la ejecución de código), y también son pruebas de verificación (buscamos encontrar defectos en el código). Se realizan de forma INCREMENTAL siguiendo una ESTRATEGIA de integración, la cual determinará EL ORDEN en el que se deben seleccionar las unidades a integrar
- Las pruebas de integración se realizan de forma incremental, añadiendo cada vez un determinado número de unidades al conjunto, hasta que al final tengamos probadas TODAS las unidades integradas. En cada una de las "fases" tendremos que REPETIR TODAS las pruebas anteriores. A este proceso de "repetición" se le denomina PRUEBAS DE REGRESIÓN. Las pruebas de regresión provocan que las últimas unidades que se añaden al conjunto sean las MENOS probadas. Este hecho debemos tenerlo en cuenta a la hora de tomar una decisión por una estrategia de integración u otra, además del tipo de proyecto que estemos desarrollando (interfaz compleja o no, lógica de negocio compleja o no, tamaño del proyecto, riesgos,...)
- Recuerda que no podemos integrar dos unidades si no hay ninguna relación entre ellas, ya que el objetivo es encontrar errores en las interfaces de las unidades (en la interconexión entre ellas). Por eso es indispensable tener en cuenta el diseño de nuestra aplicación a la hora de determinar la estrategia de integración (qué componentes van a integrarse cada vez).

DISEÑO DE PRUEBAS DE INTEGRACIÓN

- Se usan técnicas de caja negra. Básicamente se seleccionan los comportamientos a probar (en cada una de las fases de integración) siguiendo unas guías generales en función del tipo de interfaces que usemos en nuestra aplicación.
- Recuerda que si nuestro test de integración da como resultado "failure" buscaremos la causa del error en las interfaces, aunque es posible que en este nivel de pruebas, salgan a la luz errores no detectados durante las pruebas unitarias (ya que NO podemos hacer pruebas exhaustivas, por lo tanto, nunca podremos demostrar que el código probado no tiene más defectos de los que hemos sido capaces de encontrar).

AUTOMATIZACIÓN DE LAS PRUEBAS DE INTEGRACIÓN CON UNA BD

- Usaremos la librería dbunit para automatizar las pruebas de integración con una BD. Esta librería es necesaria para controlar el estado de la BD antes de las pruebas, y comprobar el estado resultante después de ellas..
- Los tests de integración deben ejecutarse después de realizar todas la pruebas unitarias. Necesitamos disponer de todos los .class de nuestras unidades (antes de decidir un ORDEN de integración), por lo que los tests requerirán del empaquetado en un .jar de todos los .class de nuestro código.
- Los tests de integración son ejecutados por el plugin failsafe. Debemos añadirlo al pom, asociando la goal integration-test a la fase con el mismo nombre.
- El plugin sql permite ejecutar scripts sql. Es interesante para incluir acciones sobre la BD durante el proceso de construcción del proyecto. Por ejemplo, "recrear" las tablas de nuestra BD en cada ejecución, e inicializar dichas tablas con ciertos datos. La goal "execute" es la que se encarga de ejecutar el script, que situaremos físicamente en la carpeta "resources" de nuestro proyecto maven. Necesitaremos configurar el driver con los dato de acceso a la BD, y también asociar la goal a alguna fase previa (por ejemplo pre-integration-test) a la fase en la que se ejecutan los tests de integración.
- Es habitual que el código a integrar esté distribuido en varios proyectos. Los proyectos maven multimódulo nos permiten agrupar y trabajar con múltiples proyectos maven. En este caso nuestro proyecto maven multimódulo contendrá varios subproyectos maven.
- El proyecto maven multimódulo (que agrupa al resto de subproyectos), únicamente contiene el fichero de configuración pom.xml, es decir, NO tiene código (carpeta src), simplemente sirve para agrupar todo el código de nuestra aplicación en una única carpeta. El pom del proyecto multimódulo permite reducir duplicaciones, ya que se puede aplicar el mismo comando a todos los submódulos (misma configuración a todos los submódulos (relación de agregación, y/o aplicar la misma configuración a los submódulos (relación de herencia)).
- Si usamos un proyecto maven multimódulo,, el mecanismo reactor ejecuta el comando maven en todos los submódulos ordenándolos previamente atendiendo a las dependencias entre ellos. Por lo tanto, si ejecutamos las pruebas de integración desde el proyecto multimódulo, estaremos integrando en orden ascendente

P07- Pruebas de aceptación: Selenium IDE

Os recordamos que el plazo para entregar los ejercicios de esta práctica (P07) termina justo ANTES de comenzar la siguiente sesión de prácticas (P08) en el laboratorio (los grupos de los lunes, por lo tanto, el próximo lunes, los de los martes, el próximo martes, ...). Pasado este plazo los profesores no revisarán vuestro trabajo sobre P07 (de forma individual), con independencia de que se indiquen en clase las soluciones que os permitirán saber si las vuestras son correctas.

Pruebas de aceptación de aplicaciones Web

El objetivo de esta práctica es automatizar pruebas de aceptación de pruebas emergentes funcionales sobre una aplicación Web, para lo que utilizaremos la herramienta Selenium IDE, desde el navegador Chrome.

Tanto para pruebas de aceptación, como para pruebas del sistema, los datos de entrada de nuestros casos de prueba están formados por una **secuencia de entradas** al sistema (eventos del sistema). Dado que vamos a realizar pruebas sobre una aplicación web, cada dato de entrada de nuestra tabla de casos de prueba es del tipo "teclear S", "pulsar con botón derecho sobre Y", "hacer doble click sobre K",... y que el resultado esperado viene dado por la **secuencia de acciones** que se desencadenan al tener lugar cada uno de los eventos: por ejemplo: cargar la página X, enviar el texto tecleado al servidor, mostrar por pantalla el valor X,... Es decir, cambia algo el "aspecto" de la tabla, por así decirlo, pero sigue siendo una tabla de casos de prueba (conjunto de entradas concretas, y resultado esperado (también concreto)).

Otra observación más: puesto que el resultado esperado no es un único "valor", sino que está formado por la secuencia de salidas obtenidas de todas y cada una de las acciones que se desencadenan al producirse los eventos, nuestro *driver* normalmente tendrá varios "asserts" (por ejemplo, lo habitual es que cada vez que se cargue una nueva página (o parte de ella, si utilizamos ajax) verifiquemos que los datos que se han cargado y se han mostrado en el navegador son los correctos.

Implementaremos los *drivers* para nuestras pruebas con *scripts* de comandos Selenese (que posteriormente guardaremos en formato json). Tal y como hemos visto en clase, Selenium IDE nos permite "capturar" las acciones del usuario desde el navegador para generar de forma automática las instrucciones (código) de nuestros *drivers*, los cuales pueden ejecutarse de forma automática tantas veces como sea necesario. Esta forma de uso de la herramienta ("grabando" las acciones del usuario sobre el navegador) es lo que se conoce como el patrón *record-playback*.

Recuerda también que, aunque la práctica sea muy "guiada", debes tener claro lo que estás haciendo en cada momento.

Bitbucket

El trabajo de esta sesión también debes subirlo a *Bitbucket*. Todo el trabajo de esta práctica deberá estar en el directorio **P07-Selenium-IDE** dentro de tu espacio de trabajo, es decir, dentro de tu carpeta: ppss-2021-Gx-apellido1-apellido2.

Navegador Chrome y Selenium IDE

En la máquina virtual tenéis instalado el navegador Firefox con la extensión Selenium IDE. Hemos constatado que Selenium IDE + Firefox no funcionan bien en la máquina virtual (sin embargo Firefox+Selenium IDE, con las mismas versiones usadas en la máquina virtual pero bajo *macOS* funcionan perfectamente).

Vamos a instalar Chrome en la máquina virtual, puesto que la combinación Chrome+Selenium IDE no plantea ningún problema.

Desde un terminal, teclea los siguientes cuatro comandos (el símbolo ">" representa el prompt, no hay que copiarlo):

```
> wget -q -O - https://dl.google.com/linux/linux_signing_key.pub | sudo apt-key add -  
> sudo sh -c 'echo "deb [arch=amd64] http://dl.google.com/linux/chrome/deb/ stable  
main" >> /etc/apt/sources.list.d/google-chrome.list'  
> sudo apt update  
> sudo apt install google-chrome-stable
```

Ejecutamos Chrome desde: **Inicio → Internet → Google Chrome**

Finalmente instalamos Selenium IDE como una extensión del navegador Chrome, desde.

<https://chrome.google.com/webstore/detail/selenium-ide/mooikfkahbdckldjndioackbalphokd>

Si quieres acceder a Chrome desde un icono en el escritorio haz lo siguiente (desde el terminal):

(1) Primer copiamos el lanzador en el escritorio con:

```
cp /usr/share/applications/google-chrome.desktop $HOME/Desktop/
```

(2) A continuación, desde el menú contextual del icono copiado en el escritorio marcamos "Confiar en este ejecutable".:

Aplicación web para los ejercicios de esta sesión

Utilizaremos una aplicación Web Java denominada **JPetStore** (<http://mybatis.github.io/spring/sample.html>) sobre la que haremos las pruebas. Se trata de una versión simplificada de la demo de Sun denominada *Java PetStore*.

La aplicación a probar la tenéis descargada en vuestro *\$HOME* en la máquina virtual, en la carpeta **jpetstore-6**.

Nota: se podría descargar con el siguiente comando:

```
> git clone https://github.com/mybatis/jpetstore-6.git
```

Para familiarizarte con el código puedes abrir el proyecto maven "jpetstore-6" desde IntelliJ. El directorio de fuentes está estructurado en 4 paquetes:

- ❖ **domain:** contiene las clases que representan los objetos del dominio del negocio, con sus getters y setters,
- ❖ **mapper:** contiene las interfaces para "mapear" los objetos java con los datos persistentes,
- ❖ **service:** contiene las clases con la lógica de negocio de la aplicación,
- ❖ **web.actions:** contiene las acciones, es decir, la lógica de la capa de presentación de la aplicación.

La aplicación utiliza el patrón MVC (Modelo-Vista-Controlador) con tres capas. (Puedes consultar <http://www.mybatis.org/jpetstore-6/> para una descripción más detallada):

- ❖ **vista** (capa de presentación): formada por ficheros jsp y ActionBeans,
- ❖ **controlador** (capa de negocio): formada por objetos del dominio y servicios, y
- ❖ **modelo** (capa de datos): formada por interfaces de mapeo con datos persistentes

En carpeta jpetstore-6 el código no es exactamente el original, ya que hemos hecho las siguientes modificaciones:

- ❖ **fichero jpetstore.css:** En la línea 174 hemos borrado dos llaves que no tienen nada en su interior.
- ❖ **fichero AccountActionBean.java:** hemos cambiado la línea 119, "authenticated=true", por "authenticated=false"

Construcción y despliegue de la aplicación web

Las aplicaciones web se empaquetan en ficheros **.war**, y dicho artefacto tiene que ser desplegado en un servidor (un servidor web o un servidor de aplicaciones). Una vez que nuestra aplicación web esté en ejecución en el servidor, podemos acceder a ella a través del navegador, utilizando la url en la que ha sido desplegada nuestra aplicación.

Maven Build profiles

(se ejecutan con:

mvn -P profileID)

El pom de nuestra aplicación web contiene varios “*build profiles*” (etiquetas `<profile>`, anidadas en la etiqueta `<profiles>`). Maven usa los “*build profile*” com una forma de asegurar la “portabilidad” de nuestras construcciones. Un *build profile* usa los elementos definidos en el pom, y puede añadir nuevos elementos y/o modificar los existentes.

Por ejemplo, en el pom usamos el plugin “*cargo*” para poner en marcha un servidor web, concretamente “tomcat” y desplegar ahí nuestra aplicación. Si queremos cambiar el contenedor para desplegar nuestra aplicación web, podemos usar uno de los “profiles” creados, o añadir uno nuevo. Dado que en la máquina virtual, en la carpeta \$HOME/descargas tenéis un zip con el servidor de aplicaciones **wildfly 21.0.1**, vamos a usar este contenedor para desplegar nuestra aplicación web, y vamos a añadir un nuevo profile a los ya existentes en nuestro pom. Concretamente añadiremos el *build profile* siguiente::

```
<profile>
  <id>wildfly21</id>
  <properties>
    <wildfly.major-version>21</wildfly.major-version>
    <wildfly.version>21.0.1.Final</wildfly.version>
    <cargo.maven.containerId>wildfly${wildfly.major-version}x</cargo.maven.containerId>
    <cargo.maven.containerUrl>http://download.jboss.org/wildfly/${wildfly.version}/wildfly-
    ${wildfly.version}.zip</cargo.maven.containerUrl>
  </properties>
</profile>
```

Hemos añadido la siguiente línea en la configuración del plugin *cargo*. El plugin se descarga el contenedor web elegido en nuestro directorio de descargas (/home/ppss/Descargas). Como ya hemos copiado el zip en ese directorio, no será necesario descargarlo, simplemente lo descomprimirá en target/cargo cuando construyamos el sistema:

```
<plugin>
  <groupId>org.codehaus.cargo</groupId>
  <artifactId>cargo-maven2-plugin</artifactId>
  <version>${cargo-maven2-plugin.version}</version>
  ...
  <zipUrlInstaller>
    <url>${cargo.maven.containerUrl}</url>
    <!-- añadimos la siguiente línea -->
    <downloadDir>${env.HOME}/Descargas</downloadDir>
  </zipUrlInstaller>
  ...
</plugin>
```

Para **empaquetar** y **desplegar** nuestra aplicación web (desde el **TERMINAL**, y la carpeta \$HOME/jpetstore-6), usaremos el comando:

> mvn package cargo:run -P wildfly21 (1)

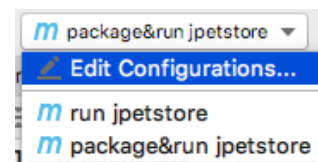
Dado que no vamos a editar nuestra aplicación web, las siguientes veces que necesitemos ejecutar la aplicación podemos hacerlo con:

> mvn cargo:run -P wildfly21 (2)

También podemos **empaquetar** y **desplegar** la aplicación desde **INTELLIJ**:

- a) Usando elementos de configuración.

Crearemos dos elementos de configuración maven con los nombres "package&run jpetstore" al que asociaremos el comando (1) y "run jpetstore", que tendrá asociado el comando (2)



- b) Desde la ventana Maven de IntelliJ.

Desde el elemento Profiles, desmarcamos el profile *tomcat90* (que es el que está activo por defecto) y marcamos *wildfly21*

A continuación podemos ejecutar la fase *package*, y arrancar y desplegar el war generado a través del plugin *cargo*.

Una vez que hemos arrancado el servidor de aplicaciones y desplegado el war, ya puedes **ejecutar** la aplicación web desde el navegador Chrome, accediendo a la URL:

<http://localhost:8080/jpetstore>

Cuando arrancamos wildfly pondremos en marcha dos procesos: la consola de administración de *wildfly* (en el puerto 9990), y el servidor web que contiene nuestra aplicación *jpetstore*, que estará escuchando en el puerto 8080-

Si has arrancado el servidor desde IntelliJ, puedes detenerlo usando el botón **cuadrado rojo** del panel de ejecución.

Si cierras IntelliJ y te has dejado el servidor "en marcha", puedes abortar el proceso posteriormente desde el terminal usando el comando `lsof`: **> lsof -i:8080**

Este comando nos muestra la información de qué proceso está ocupando ese puerto. Si hay algún proceso escuchando en ese puerto, nos mostrará, entre otras cosas, su PID.

Sabiendo el PID podemos abortar el proceso con **> kill -9 pidProceso**

NOTA: Dado que IntelliJ consume bastantes recursos de la máquina y no vamos a necesitar editar el código de nuestra aplicación web, os aconsejamos que en sucesivas ejecuciones uséis directamente el terminal. Y ejecutéis el comando (2) anterior desde la carpeta *jpetstore-6* (es decir, desde el directorio en el que se encuentra el pom del proyecto maven). Para parar el servidor puedes detener el proceso pulsando *Ctrl-C* desde el terminal

Ejercicios

Accede a la aplicación web y familiarízate con ella antes de comenzar a crear nuestros tests de aceptación.

La pantalla principal nos muestra la bienvenida a la tienda de animales. Una vez que entres en la tienda puedes mostrar una página de ayuda pulsando sobre el enlace "?" en la parte superior central de la página. Pinchando sobre el logo de la parte superior izquierda vuelves a la página principal de la tienda, en donde encontrarás 5 catálogos de diferentes animales, desde los que podrás seleccionar los que te interesen para añadirlos a tu carrito de la compra y luego poder hacer efectiva dicha compra si eres usuario de la tienda.

Vamos a implementar algunos tests de pruebas de aceptación de propiedades emergentes funcionales con Selenium IDE sobre la aplicación web.

👉 Ejercicio 1: Primer Test Case Selenium (Caso de prueba 1)

Vamos a realizar una primera **grabación de una secuencia de acciones** para familiarizarnos con Selenium IDE. Abre la herramienta pulsando sobre el icono de la barra de herramientas de Chrome, y selecciona "Record a new test in a new project":

- Nombre del proyecto: *DriversSeleniumIDE*.
- Base URL: *http://localhost:8080/jpetstore*
- ... y comenzamos a grabar

Antes de continuar, y para que veas en todo momento lo que está generando la herramienta, redimensiona las ventanas y asegúrate de que son totalmente visibles tanto la de grabación (ventana Chrome con el mensaje "Selenium IDE is recording"), como la herramienta Selenium IDE.

El **caso de prueba** consiste en el siguiente escenario:

1. Entra en la tienda
2. Verifica que el título de la página es "JPetStore Demo". Para ello con botón derecho desde cualquier sitio de la página selecciona el comando selenese "verify Title" (desde el menú contextual)
3. Haz click sobre la imagen del pez
4. Pulsa sobre el ítem del catálogo con identificador FI-SW-02
5. Verifica que el texto "Toothless Tiger Shark" está presente en la página (comando "verify Text")
6. Pulsa sobre "Return to Fish", seguidamente sobre "Return to Main Menu", y verifica que el texto "Sign In" está presente en la página

"Graba" con la herramienta el código del driver que implementa el caso de prueba anterior. Observa cómo se van generando automáticamente los comandos selenese a medida que interaccionamos con nuestra aplicación.

Detén la grabación, y pon el nombre "**Caso de prueba 1**" al driver implementado.

La implementación del driver debe quedar así:

Puedes observar que, tal y como hemos explicado en clase, cada comando requiere uno o dos parámetros.

Si el comando requiere un "locator", éste será el primer parámetro. Un "locator" representa un elemento html. Así, por ejemplo, el comando **click** tiene como parámetro el locator **linkText**, asociado a un hipervínculo html.

Observa también que las cadenas de caracteres NO llevan "comillas"

Driver "Caso de prueba 1"		
1	open	/jpetstore/
2	set window size	873x765
3	click	linkText=Enter the Store
4	verify title	JPetStore Demo
5	click	css=area:nth-child(2)
6	click	linkText=FI-SW-02
7	click	css=h2
8	verify text	css=h2 Tiger Shark
9	click	linkText=Return to FISH
10	click	linkText=Return to Main Menu
11	verify text	linkText=Sign In Sign In

Fíjate también en que Selenium añade alguna acción que realmente no hemos ejecutado: por ejemplo el comando 2 para redimensionar la ventana del navegador. Esta línea la añadirá Selenium en todos nuestros tests. Podríamos borrarla, aunque por comodidad no lo vamos a hacer. También puedes ver que cuando hemos insertado el comando verify, la herramienta también ha grabado el "click" que hemos hecho en la página (comando 7) para abrir el menú contextual y elegir el comando selenese. Vamos a dejarlos porque de hecho, son las acciones que realmente hemos necesitado hacer sobre la página. No los incluimos de forma explícita en los casos de prueba, pero se añaden automáticamente. Puedes probar a "comentar" estos los comandos que *selenese* nos ha añadido y que no son estrictamente los pasos indicados en el escenario de prueba. Para "comentar" un comando usaremos el botón "/" a la derecha del campo de texto Command. Este botón "**habilita/deshabilita**" la **ejecución del comando** correspondiente. En concreto, prueba a comentar los comandos 2 y 7 y verás que el test sigue funcionando correctamente. También puedes borrarlos usando la tecla "del".

Guarda el proyecto en un fichero con nombre "DriversSeleniumIDE" en el directorio *ppss-2021-Gx.../P07-Selenium-IDE*. El fichero generado tendrá extensión .side, pero es un fichero de texto en formato JSON.

Ahora ejecuta el test que acabas de crear (segundo icono que tiene la forma de triángulo). Prueba a cambiar la velocidad de ejecución de forma que puedas ver perfectamente la ejecución de todos y cada uno de los comandos selenese de nuestro driver.

🔗 Ejercicio 2: Test Case *compraNewUser*

Vamos a crear un nuevo test en nuestro proyecto Selenium, con el nombre "**test-compraNewUser**". En este caso vamos a usar un escenario en el que un usuario que todavía NO está registrado en la página quiere hacer una compra de varios productos. En este caso, después de añadir los productos al carrito y proceder a hacer efectiva la compra, ésta no se completará si el usuario no se ha creado una cuenta en la tienda. Después de crear la cuenta, se podrá proceder a la compra de los productos almacenados en el carrito (éstos se mantienen en el carrito durante el proceso de registro del nuevo cliente).

Como ya hemos indicado en este test actuaremos como un usuario no registrado que quiere hacer una compra de dos perros y un gato. Para ello pulsamos el botón de grabación y accedemos a nuestra aplicación desde el navegador (URL: <http://localhost:8080/jpetstore>).

El **escenario** elegido para el **caso de prueba** es el siguiente:

1. Entramos en la tienda y pulsamos con botón izquierdo del ratón sobre la imagen del perro
2. Nos aseguraremos de que estamos en la página correcta verificando que el texto "Dogs" se muestra en la página
3. Vamos a **comprar un Bulldog**. Verificaremos (igual que hemos hecho antes) que el texto "Bulldog" aparece en la página
4. Pulsamos sobre el enlace K9-BD-01 y verificamos que estamos en la página correcta verificando que el texto "Bulldog" que aparece sobre la tabla.
5. Queremos comprar un **macho adulto**. Verificamos que en la página aparece el texto "Male Adult Bulldog". A continuación pinchamos sobre "Add to cart" de la primera fila de la tabla. Verificamos que estamos en la página correcta comprobando que el texto "Shopping Cart" aparece en la nueva página cargada. Queremos comprar dos unidades, por lo que editaremos el campo de texto y pondremos un 2 (no es necesario que pulsemos "return" después de poner el 2). A continuación seleccionamos la opción "Update Cart" y verificaremos que aparece el texto "Sub Total:\$37,00".
6. Ahora vamos a **comprar un gato**. Para lo cual pinchamos sobre el enlace "Cats" en la parte superior de la página. Verificamos que estamos en la página correcta comprobando que aparece el texto "Cats". Antes de seguir, vamos a comprobar que efectivamente en el carrito tenemos los 2 bulldogs, para lo cual pinchamos sobre el icono del carrito de la compra de la parte superior de la página. Verificamos que efectivamente aparece el texto "Male Adult Bulldog" y el sub total es el mismo de antes. Volvemos de nuevo al enlace "Cats", y procedemos a comprar un ejemplar de persa. Para ello verificamos el texto "Persian", y pinchamos sobre FL-DLH-02. Verifica que el texto "Adult Female Persian" está presente en la página y añade al carrito una hembra adulta. Ahora nuestro carrito debe contener los productos "Male Adult Bulldog" y "Adult Female Persian", y el nuevo sub total debe ser de 130,50 dólares.
7. Antes de proceder a hacer efectiva la compra, pensamos que igual no es buena idea la compra de dos perros y un gato. **Decidimos no comprar el gato**, por lo que eliminamos este ítem. Ahora vamos a comprobar que el identificador EST-16 NO aparece en la página. Esto no lo podemos hacer de forma automática, sino que tendremos que introducir el comando manualmente. Se trata del comando "*verify not text*". No detengas la grabación.
8. Selecciona la última línea del editor de texto, que está vacía y aparece sombreada en azul en su parte superior. A continuación, en el campo "Command" busca y selecciona el comando "verify not text". En el campo de texto Value escribiremos EST-16. Para escribir el valor del campo target, pulsamos el botón a la izquierda de la lupa (para activar la búsqueda del elemento en el navegador), nos situamos con el ratón sobre la tabla en el navegador, cuando TODA la tabla aparezca resaltada, pulsamos con el botón izquierdo del ratón, y el cuadro de texto *target* se rellenará con el locator de la tabla.

9. Haz click sobre la siguiente línea en blanco del editor de comandos selenese, y continúa la grabación verificando que en la página el sub total vuelve a ser de 37 dólares. Hasta ahora nuestro script de pruebas debe tener el siguiente “aspecto” (No detengas la grabación):

Driver "CompraNewUser"		
1	open	http://localhost:8080/jpetstore/
2	set window size	647x497
3	click	linkText=Enter the Store
4	click	css=area:nth-child(3)
5	click	css=tr:nth-child(2) > td:nth-child(2)
6	verify text	css=tr:nth-child(2) > td:nth-child(2) Bulldog
7	click	linkText=K9-BD-01
8	click	css=h2
9	verify text	css=h2 Bulldog
10	click	css=tr:nth-child(2) > td:nth-child(3)
11	verify text	css=tr:nth-child(2) > td:nth-child(3) Male Adult Bulldog
12	click	linkText=Add to Cart
13	click	css=h2
14	verify text	css=h2 Shopping Cart
15	click	name=EST-6
16	type	name=EST-6 2
17	click	name=updateCartQuantities
18	click	css=tr:nth-child(3) > td:nth-child(1)
19	verify text	css=tr:nth-child(3) > td:nth-child(1) Sub Total: \$37,00
20	click	css=a:nth-child(7) > img
21	click	css=h2
22	verify text	css=h2 Cats
23	click	name=img_cart
24	click	css=td:nth-child(3)
25	verify text	css=td:nth-child(3) Male Adult Bulldog
26	click	css=tr:nth-child(3) > td:nth-child(1)
27	verify text	css=tr:nth-child(3) > td:nth-child(1) Sub Total: \$37,00
28	click	css=a:nth-child(7) > img
29	click	css=tr:nth-child(3) > td:nth-child(2)
30	verify text	css=tr:nth-child(3) > td:nth-child(2) Persian
31	click	linkText=FL-DLH-02
32	click	css=tr:nth-child(3) > td:nth-child(3)
33	verify text	css=tr:nth-child(3) > td:nth-child(3) Adult Male Persian
34	click	linkText=Add to Cart
35	click	css=tr:nth-child(2) > td:nth-child(3)
36	verify text	css=tr:nth-child(2) > td:nth-child(3) Male Adult Bulldog
37	click	css=tr:nth-child(3) > td:nth-child(3)
38	verify text	css=tr:nth-child(3) > td:nth-child(3) Adult Female Persian
39	click	css=tr:nth-child(4) > td:nth-child(1)
40	verify text	css=tr:nth-child(4) > td:nth-child(1) Sub Total: \$130,50
41	click	css=tr:nth-child(3) .Button
42	verify not text	css=table Adult Female Persian
43	verify text	css=tr:nth-child(3) > td:nth-child(1) Sub Total: \$37,00

10. Ahora procederemos a **realizar la compra**, pinchando sobre “Proceed to Checkout”. Verificamos que en la nueva página nos aparece el mensaje: “You must sign on before attempting...”. Vamos a probar a introducir el login “z” y el password “z”. Verificaremos que en la página aparece el texto: “Invalid username or password...”. Como somos un usuario nuevo, tendremos que **registrarnos** primero para poder hacer la compra, por lo que pinchamos sobre “Register now”. Nos aparecerá una página con el texto “User Information”. Rellenamos los campos. Por ejemplo podemos poner como User ID y password el carácter “z”. Rellenaremos todos los campos de “Account

Information" con la letra "z" (por ejemplo. Si quieres, puedes poner cualquier cadena de caracteres). Recuerda rellenar TODOS los campos. Dejaremos sin rellenar el apartado "Profile Information" como aparece por defecto. Finalmente pincharemos sobre "Save Account Information".

IMPORTANTE!!!: Al introducir el nombre de usuario, se graba como valor de target el locator "id" del campo de texto etiquetado como "User ID" (compruébalo). El valor de este atributo cambiará cada vez que se recargue la página, por lo que vamos a tener que usar otro "locator". Pulsa sobre el desplegable de "Target" y elige el segundo locator de la lista (name=username). De no hacerlo así, si repetimos el test, éste fallará. Puedes hacer los cambios sin detener la grabación. Recuerda que **siempre que introduzcas el valor del campo User ID**, se debe usar el **locator name** en lugar del **locator id**.

Selenium IDE, cuando usamos un *locator* en el campo *target*, siempre guarda todos los *locators* alternativos que podríamos usar con ese elemento, por eso nos aparecen más opciones cuando pulsamos la lista desplegable de Target.

11. Después de crear la cuenta verificamos que el texto "Sign In" aparece en la pantalla. Después volvemos a nuestro **carrito de la compra** pulsando sobre el icono del carrito de la parte superior de la pantalla. Verificaremos que aparece el texto "Shopping Cart" y que el subtotal sigue siendo de \$37,00. Ahora pulsamos sobre el botón "Proceed to Checkout", **introducimos el login y password "z", "z"** y entramos. Volvemos a seleccionar el carrito de la compra y comprobamos que el subtotal sigue siendo de 37,00 dólares.
12. Volvemos a pulsar sobre "Proceed to checkout", sobre "continue" y confirmamos. A continuación verificaremos que el nombre del usuario de la dirección de facturación es el correcto ("z"), y que la primera dirección de envío es la correcta. ("z"). Finalmente cerramos la sesión pulsando sobre "Sign Out" y accedemos al carrito de la compra para verificar que el subtotal ahora es de 0 euros. A continuación mostramos la secuencia de comandos que hemos añadido (por simplicidad hemos quitado de la secuencia los "clicks" previos a la inserción y verificación de texto):

Driver "CompraNewUser"			
44	click	linkText=Proceed to Checkout	
46	verify text	css=li	You must sign on ...
48	type	name=username	z
50	type	name=password	z
51	click	name=signon	
53	verify text	css=li	Invalid username ...
54	click	linkText=Register Now!	
55	type	name=username	z
57	type	name=password	z
59	type	name=repeatedPassword	z
...
79	type	name=account.country	z
80	click	name=newAccount	
81	verify text	linkText=Sign In	Sign In
82	click	name=img_cart	
84	verify text	css=h2	Shopping Cart
85	click	css=tr:nth-child(3) > td:nth-child(1)	
86	click	name=img_cart	
88	verify text	css=tr:nth-child(3) > td:nth-child(1)	Sub Total: \$37,00
89	click	linkText=Proceed to Checkout	
91	type	name=username	z
93	type	name=password	z
94	click	name=signon	
95	click	name=img_cart	
97	verify text	css=tr:nth-child(3) > td:nth-child(1)	Sub Total: \$37,00
98	click	linkText=Proceed to Checkout	

Driver "CompraNewUser"			
99	click	name=newOrder	
101	verify text	css=tr:nth-child(3) > td:nth-child(2)	z
103	verify text	css=tr:nth-child(5) > td:nth-child(2)	z
105	verify text	css=tr:nth-child(12) > td:nth-child(2)	z
106	click	linkText=Sign Out	
107	click	name=img_cart	
109	verify text	css=tr:nth-child(3) > td:nth-child(1)	Sub Total: \$0,00

Antes de ejecutar el test tendremos que “borrar” el usuario que hemos creado durante la creación del script de pruebas (driver). Puesto que la aplicación utiliza una base de datos en memoria (HSQLDB), una forma de “limpiar” la base de datos es reiniciar la aplicación. Para ello bastaría con parar y volver a arrancar el servidor de aplicaciones Wildfly. Podemos hacerlo desde IntelliJ o desde un terminal.

Una vez que volvamos a arrancar el servidor, ya podemos ejecutar nuestro test desde Selenium IDE. Puedes poner la velocidad más baja de ejecución para poder ir viendo claramente los pasos (también puedes pausar la ejecución en cualquier momento).

Si intentas repetir el test sin reiniciar el servidor puedes comprobar que el test fallará puesto que estaremos intentando darnos de alta dos veces en el sistema.

🔗 Ejercicio 3: Test Case *compraOldUser*

Ahora vamos a crear otro script de prueba (driver) con el nombre “**test-compra-old-user**”. En este caso se trata de un usuario previamente registrado, por ejemplo, el usuario z que acabamos de crear, que quiere comprar 500 ejemplares de iguana verde adulta (el identificador del producto es: RP-LI-02, y el identificador del ítem es: EST-13). Se trata de comprobar que, después de hacer la compra, deben quedar 500 ejemplares menos en stock.

Para ello tendrás que incluir en el script de pruebas el acceso a la página en la que se muestra la información del stock inicial de un ejemplar, recordar esta cantidad para luego restarle 500 unidades, y volver a navegar por dicha página para comprobar que efectivamente el nuevo valor ha disminuido en 500 unidades. Para poder programar esto, vamos a tener que utilizar **variables** para almacenar los valores que necesitemos y operar con ellos, ya que vamos a trabajar con contenidos de la página html que son dinámicos (no sabemos a priori cuál será el stock, y dependiendo de la cantidad que queramos comprar, este stock tendrá un valor que tampoco conocemos a priori). En los siguientes apartados indicamos que comandos podéis utilizar para trabajar con la información dinámica que genera la página.

Nuestro **caso de prueba** consistirá en el siguiente **escenario** (puedes grabar parte de él, y más adelante explicamos cómo programar los pasos 5 y 9. Recuerda que en cualquier momento podemos añadir/modificar/borrar cualquier comando selenese del editor):

1. Entramos en la tienda y nos logueamos como el usuario ‘z’ con password ‘z’. Verificamos que estamos en la página correcta (debe aparecer el texto “Welcome z”).
2. Pulsamos con botón izquierdo del ratón sobre la imagen de los reptiles
3. Seleccionamos el producto asociado a la iguana
4. Accedemos al identificador del ítem (EST-13)
5. (Explicaremos este paso a continuación). Guardamos el texto con la información número de ejemplares en stock en una variable. Extraemos de dicha cadena de caracteres solamente el número de ejemplares disponibles, y almacenamos dicho valor en una segunda variable.
6. Añadimos al carrito el producto seleccionado (Iguana verde adulta)
7. Cambiamos la cantidad a comprar, ya que deseamos comprar 500 unidades y actualizamos el carrito
8. Procedemos a realizar la compra (Botón "Proceed to Checkout"), y confirmamos los datos.

9. Volvemos a acceder a reptiles, a continuación a la iguana, y posteriormente a la iguana verde adulta. Verificamos que la cantidad de ejemplares restantes es la correcta (tiene que haber 500 unidades menos disponibles que antes de realizar la compra)
10. Cerramos la sesión (y verificamos que en la página aparece el texto “Sign In”).

Para programar el paso 5 necesitas usar el comando Selenese **store text**. Dicho comando almacena el texto de un elemento de nuestra página en una variable, por lo tanto requiere dos parámetros. Puedes insertar el comando “store text” y activar el localizador de elementos del campo “Target” para buscar el valor del locator adecuado. Podemos usar también el comando **echo** para “imprimir” en la pantalla de Log el valor de la variable del comando anterior. Para referenciar el valor de una variable, pongamos por ejemplo var1 se utiliza el nombre de la variable entre llaves y precedida de un “\$”, es decir \${var1}.

Nota: el comando store text hay que introducirlo manualmente

store text	css=tr:nth-child(5) > td	stockText
echo	Texto con el stock de iguanas	
echo	\${stockText}	

La variable *stockText* contiene el texto “10000 in stock”, pero necesitamos almacenar solamente la cantidad (10000), en lugar de todo el texto, para ello puedes utilizar el comando Selenese **execute script**. Como ya hemos indicado en clase, este comando ejecuta un *script Javascript*. Para extraer un valor numérico de una cadena de caracteres podemos usar el siguiente *script*:

```
var res = String(${stockText}).match(/\\d+\\.?\\d*/g);
return res;
```

La función *String* convierte un valor (en este caso el valor de nuestra variable *stockText*) en una cadena de caracteres. A continuación aplicamos sobre dicho string la función *match()*, que devuelve la subcadena que coincida con el patrón (expresión regular) que se pasa como parámetro. Finalmente usamos la sentencia *return* para devolver el resultado final.

execute script	var res = String(\${stockText}).match(/\\d+\\.?\\d*/g); return res;	stockNumber
echo	Valor de stock (solo numeros)	
echo	\${stockNumber}	

Usaremos una tercera variable para almacenar la diferencia entre el valor del stock inicial (*stockNumber*) y las 500 unidades que queremos comprar. Verificaremos que el valor resultante de la variable es el correcto (es decir, que ahora hay 500 unidades menos en stock de las que había antes de realizar la compra). Para obtener la diferencia podemos usar de nuevo el comando **execute script**.

execute script	return \${stockNumber} - 500	stockEsperado
echo	Valor esperado	
echo	\${stockEsperado}	

Para verificar que se ha actualizado correctamente el stock después de la compra, usaremos el comando **assert**. Dicho comando usa como primer parámetro el nombre de una variable (sin llaves), que contiene el valor esperado, y como segundo parámetro el valor que queremos comparar con el esperado.

assert	stockEsperado	\${stockNumberUpdated}
--------	---------------	------------------------

En este caso, *\${stockNumberUpdated}* es el valor numérico del stock que consultamos después de hacer la compra.

🔗 Ejercicio 4: Test Case *controlFlow*

Ahora vamos a crear otro script de prueba (driver) con el nombre “**controlFlow**”. La idea es practicar con alguno de los comandos que permiten alterar la secuencia de ejecución del driver. Por ejemplo, el comando **if**. Este comando requiere un argumento que pueda evaluarse a true o false.

El caso de prueba en consiste en el siguiente **requerimiento**:

1. Entramos en la tienda y después de verificar que aparece el texto "Sign In", pulsamos sobre el enlace e introducimos el login "ppss", y password "ppss".
2. Si el usuario no tiene cuenta en la tienda (más adelante explicamos cómo saber esto), entonces creamos el nuevo usuario, con login y password "ppss", y puedes rellenar el resto de campos con el valor "a". A continuación nos logueamos en la tienda de nuevo, y ahora nos debe aparecer el mensaje "Welcome a!".
3. Después de entrar con éxito en la tienda, verificamos que en la página aparece el mensaje "Welcome a!" y a continuación saldremos de la tienda (opción "Sign Out").

Para averiguar si el usuario tiene cuenta en la tienda o no, tenemos que usar la información del código de las páginas web a las que somos redirigidos en el caso de que el usuario ya tenga la cuenta creada o no. Si te fijas, ambas páginas son diferentes, y además, no hay ningún elemento (html) de la página que tenga el mismo locator pero diferentes valores en ambas.

En este caso, vamos a usar el comando **store xpath count**. Este comando requiere dos parámetros, en el primero indicaremos una ruta relativa, en el segundo almacenaremos el número de elementos de la página que contienen ese *path*. Aprovecharemos el hecho de que la página que muestra cuando el usuario no está registrado contiene 2 formularios, mientras que la página que muestra cuando el usuario ya está registrado y consigue entrar al sistema sólo tiene 1 formulario. La ruta *xpath* que vamos a usar será, por lo tanto, ***xpath=//form***.

A continuación mostramos parte del contenido de tu driver (omitimos alguno de los clicks y pasos que ya hemos explicado antes).

En este caso vamos a usar el operador "==" de Javascript.

El operador "==" realiza una igualdad estricta, eso significa que tienen que coincidir los valores y los tipos.

Driver "ControlFlow"		
open	http://localhost:8080/jpetstore/	
set window size	807x869	
click	linkText=Enter the Store	
click	id=MenuContent	
verify text	linkText=Sign In	Sign In
click	linkText=Sign In	
type	name=username	ppss
type	name=password	ppss
click	name=signon	
store xpath count	xpath=//form	formulario
echo	\${formulario}	
if	\${formulario}==2	
echo	usuario no válido	
click	linkText=Register Now!	
type	name=username	ppss
type	name=password	ppss
type	name=repeatedPassword	ppss
type	name=account.firstName	a
...	...	
click	name=newAccount	
click	linkText=Sign In	
type	name=username	ppss
type	name=password	ppss
click	name=signon	
end		
verify text	id=WelcomeContent	Welcome a!
click	linkText=Sign Out	
verify text	linkText=Sign In	Sign In

Ejecuta el driver antes de crear el usuario ppss y repite el test para comprobar que funciona en ambos casos.

Resumen



¿Qué **conceptos** y **cuestiones** me deben quedar CLAROS después de hacer la práctica?



PRUEBAS DEL SISTEMA Y ACEPTACIÓN

- Las pruebas del sistema son pruebas de VERIFICACIÓN, mientras que las pruebas de aceptación son pruebas de VALIDACIÓN. En ambos casos nuestro SUT estará formado por todo el código de nuestra aplicación.
- Las pruebas del sistema se diseñan desde el punto de vista del desarrollador mientras que las pruebas de aceptación se diseñan desde el punto de vista del usuario final. En ambos casos se deben tener en cuenta varios entradas y resultados esperados "intermedios".
- Tanto las pruebas del sistema como las de aceptación se basan en las propiedades emergentes. Dadas dichas propiedades emergentes, el objetivo de las pruebas del sistema es encontrar defectos en las funcionalidades de dicho sistema, mientras que el objetivo de las pruebas de aceptación es comprobar que se satisfacen los criterios de aceptación.

DISEÑO DE PRUEBAS DE ACEPTACIÓN

- En ambos casos se usan técnicas de caja negra.
- En una prueba del sistema la selección de comportamientos a probar se hace desde un punto de vista técnico (se tienen en cuenta cómo se ha implementado los componentes implicados en las funcionalidades probadas).
- En una prueba de aceptación el diseño se realiza pensando siempre en el uso "real" de nuestra aplicación por el usuario o usuarios finales. (no tenemos en cuenta qué componentes están implicados).

AUTOMATIZACIÓN DE LAS PRUEBAS DE ACEPTACIÓN CON SELENIUM IDE

- Seleccionamos diferentes escenarios de uso de nuestra aplicación. Selenium IDE NO es un lenguaje de programación, es una herramienta que nos permite generar scripts (de forma automática o manual), que podemos ejecutar desde un navegador, por lo tanto podremos usarla si nuestra aplicación tiene una interfaz web.
- Aunque Selenium IDE no es un lenguaje de programación, en las últimas versiones se han incorporado comandos para controlar el flujo de ejecución de nuestros scripts (incluyendo por ejemplo acciones condicionales y bucles).
- No es posible integrar Selenium IDE en una herramienta de construcción de proyectos, lo que constituye una limitación frente a otras aproximaciones.
- En los ejercicios propuestos se han trabajado una serie de comandos, que es necesario conocer y saber aplicar.
- Recuerda que esta práctica, aunque es muy guiada y se os proporciona la solución, requiere un trabajo personal para poder asimilar todos los conceptos trabajados en clase.

P08- Pruebas de aceptación: Selenium WebDriver

Os recordamos que el plazo para entregar los ejercicios de esta práctica (P08) termina justo ANTES de comenzar la siguiente sesión de prácticas (P09) en el laboratorio (los grupos de los lunes, por lo tanto, el próximo lunes, los de los martes, el próximo martes, ...). Pasado este plazo los profesores no revisarán vuestro trabajo sobre P08 (de forma individual), con independencia de que se indiquen en clase las soluciones que os permitirán saber si las vuestras son correctas.

Pruebas de aceptación de aplicaciones Web

El objetivo de esta práctica es automatizar pruebas de aceptación de pruebas emergentes funcionales sobre una aplicación Web. Utilizaremos la librería Selenium WebDriver, desde el navegador Chrome.

Tal y como hemos explicado en clase, usaremos un proyecto Maven que contendrá únicamente nuestros drivers, puesto que no disponemos del código fuente de la aplicación sobre las que haremos las pruebas de aceptación. Al igual que en la sesión anterior, proporcionamos los casos de prueba obtenidos a partir de un escenario de la aplicación a probar. Recuerda que nuestro objetivo concreto es validar la funcionalidad del sistema.

Bitbucket

El trabajo de esta sesión también debes subirlo a *Bitbucket*. Todo el trabajo de esta práctica deberá estar en el directorio **P08-WebDriver** dentro de tu espacio de trabajo, es decir, dentro de tu carpeta: ppss-2021-Gx-apellido1-apellido2.

ChromeDriver

La librería webdriver interactúa con el navegador a través un driver. Como vamos a ejecutar nuestros tests a través de Chrome necesitamos usar el driver **chromedriver**.

Vamos a descargarlo desde: <https://chromedriver.storage.googleapis.com/index.html>

Necesitamos la versión que coincida con la versión del navegador. Si la versión del navegador es, por ejemplo, 89.0.4389.114, nos descargaremos la versión 89.0.4389.23, para linux, es decir, el fichero: **chromedriver_linux64.zip** de la carpeta **89.0.4389.23**.

El fichero descargado lo copiamos en nuestro \$HOME, en la carpeta **chromedriver**, y lo descomprimos.

A continuación vamos a añadir la ruta del driver en nuestro PATH, para que esté accesible desde cualquier directorio. Para ello, editamos el fichero oculto **.profile** que se encuentra en nuestro \$HOME, y añadimos la línea (al final del fichero):

```
export PATH=$PATH:/home/ppss/chromedriver
```

Guardamos los cambios, cerramos la sesión, y volvemos a entrar para que los cambios en el fichero **.profile** tengan efecto.

El driver necesita usar la librería libconf 2.4. La instalamos con el comando:

```
> sudo apt-get install libgconf-2-4
```

Finalmente podemos probar el driver desde un terminal tecleando el nombre del ejecutable:

```
> chromedriver
```


Ahora ya estamos en disposición de usar el driver desde nuestro código, a partir de una instancia de tipo `ChromeDriver`:

```
WebDriver driver = new ChromeDriver();
```

 (opción 1)

De forma alternativa, podríamos copiar el driver (fichero `chromedriver`), por ejemplo en la carpeta `src/test/resources/drivers` de nuestro proyecto maven. En ese caso, usaríamos las sentencias:

```
System.setProperty("webdriver.chrome.driver",  
    "./src/test/resources/drivers/chromedriver");
```

 (opción 2)

```
WebDriver driver = new ChromeDriver();
```

Las dos opciones son válidas. Con la opción 2 conseguimos que nuestro código sea más portable, pero puedes usar la opción 1 puesto que hemos actualizado el PATH del sistema a través del fichero `.profile`.

Cookies

Las cookies son datos almacenados en ficheros de texto en el ordenador. Cuando un servidor web envía una página a un navegador, la "conexión" termina, y el servidor "olvida" cualquier cosa sobre el usuario.

Las cookies se inventaron para resolver el problema de "cómo recordar información sobre el usuario", de forma que cuando un usuario visita una página web, se almacena cierta información en una cookie, que se enviará al servidor, de forma que dicho servidor sabrá que la petición proviene del mismo usuario.

Por lo tanto, las cookies constituyen un mecanismo para mantener el estado en una aplicación Web. Es decir, la idea es permitir que una aplicación web tenga la capacidad de interactuar con un determinado usuario, diferenciándolo del resto. Si no mantenemos el estado, cada vez que un usuario accede a una página web, el servidor no sabrá si se trata del mismo usuario o si cada petición es de un usuario diferente. Como ejemplo: es como cuando dejamos la ropa en la tintorería, y el dependiente nos da un ticket, de forma que cuando vayamos a recogerla, sabrán que hemos ido antes a dejarla, y qué ropa venimos a recoger. Pues bien, ese ticket es similar a una cookie.

Cada cookie se asocia con una serie de propiedades: nombre, valor, dominio, ruta (path), fecha de expiración, y si segura o no.

Cuando validamos por ejemplo una aplicación de venta on-line, necesitaremos automatizar escenarios de prueba como hacer un pedido, ver el carrito de compra, proporcionar los datos de pago, confirmar el pedido, etc. Si no almacenamos las cookies, necesitaremos loguearnos en el sistema cada vez que ejecutemos cualquiera de los escenarios anteriores, lo cual incrementará el tiempo de ejecución de nuestros tests.

Una posible solución es almacenar las cookies en un fichero, y posteriormente recuperarlas y guardarlas en el navegador. De esta forma podremos "saltarnos" el proceso de login en nuestro test ya que el navegador ya tendrá esta información.

En la carpeta **Plantillas-P08**, hemos proporcionado una clase ***Cookies***, con 3 métodos, para poder:

- almacenar en un fichero de texto en el directorio target, las cookies generadas cuando los logueamos en el sistema,
- leer la información sobre las cookies almacenada en el fichero y guardarla en el navegador
- imprimir por pantalla las cookies almacenadas en nuestro navegador.

Usaremos esta clase en el tercer ejercicio de esta práctica.

Ejercicios

Vamos a implementar nuestros tests de pruebas de aceptación para una aplicación Web denominada Guru Bank, a la que accederemos desde Chrome usando la url <http://demo.guru99.com/V5>.

Una vez en la página principal, debéis **crearos una cuenta** de administrador para usar la aplicación (siguiendo las instrucciones que se muestran en la parte inferior de dicha página). Recuerda que, tal y como se indica, el usuario y password asignados tienen una validez de 20 días, por lo que si necesitas usar la aplicación pasado ese tiempo deberás crear otra cuenta (y actualizar dicha información en el código de tus tests).

Para los ejercicios de esta sesión crea, en la carpeta *ppss-2021-Gx-apellido1-apellido2/P08-WebDriver/* un proyecto maven, con groupId = **ppss**, y artifactId= **guru99Bank**.

Lo PRIMERO que tienes que hacer es configurar correctamente el pom. Debes incluir las propiedades, dependencias y plugins necesarios para implementar tests unitarios con webdriver. Recuerda que para Maven serán tests unitarios (y se ejecutarán a través del plugin surefire), pero realmente son tests de aceptación, tal y como ya hemos explicado en clase.

Si estuviésemos haciendo las pruebas de aceptación sobre código desarrollado por nosotros, en *src/main* tendríamos el código fuente a probar, y en *src/test* tendríamos tanto los tests unitarios (ejecutados con surefire), como el resto de tests (ejecutados con failsafe).

Observaciones sobre esperas implícitas y/o explícitas

Para sincronizar la carga de las páginas con la ejecución de nuestros drivers, usaremos un **wait implícito**. Recuerda que en este caso establecemos el mismo temporizador para todos los *webelements* de las páginas. Sólo se usa una vez, antes de ejecutar cada test. Este código webdriver lo incluiremos en nuestro test (aunque usemos el patrón Page Object) ya que no se verá afectado por posibles cambios de las páginas html a las que accede dicho test. El valor del temporizador será en segundos, y dependerá de la máquina en la que estéis ejecutando los tests. Podéis probar inicialmente con 5 segundos y ajustarlo a un valor mayor o menor si es necesario.

También puedes usar un **wait explícito**, en cuyo caso sólo afectará al elemento al que asociemos el temporizador.

Ejemplo de **wait implícito**:

```
driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);
```

Ejemplo de **wait explícito** para una ventana de alerta:

```
WebDriverWait wait = new WebDriverWait(driver, 10);  
wait.until(ExpectedConditions.alertIsPresent());
```

NOTA: Recuerda que aconsejamos, para realizar este tipo de pruebas, que intentes tener el menor número de aplicaciones abiertas de forma innecesaria, ya que esto puede "ralentizar" la carga de las páginas en el navegador, dificultando así el proceso de testing (generando errores debido a que no se cargan las páginas con la suficiente rapidez).

👉 Ejercicio 1: Tests Login

Vamos a crear el paquete ***ejercicio1.sinPageObject***, en el que implementaremos dos casos de prueba sin usar el patrón Page Object.

Los casos de prueba a implementar son los siguientes (en una clase **TestLogin**):

- Un primer caso de prueba (con el nombre ***test_Login_Correct()***) en el que accederemos a la aplicación bancaria con un nombre de usuario y contraseña correctos y en pantalla aparecerá el correspondiente mensaje de bienvenida (podéis usar el mensaje completo o parte de él para verificar el resultado).
- Un segundo caso de prueba, al que llamaremos ***test_Login_Incorrect()*** en el que accederemos a la aplicación bancaria con un nombre de usuario y contraseña incorrectos, y debemos verificar que se muestra el mensaje "User is not valid" en un *alert Box*.

Para implementar el driver necesitas usar la clase *Alert*, que representa un elemento *Alert Box*.

Alert Box

Un *Alert Box* no es más que un pequeño "recuadro" que aparece en la pantalla que proporciona algún tipo de información o aviso sobre alguna operación que intentamos realizar, pudiendo solicitarnos algún tipo de permiso para realizar dicha operación

A continuación mostramos un ejemplo de algunos métodos que podemos usar:

```
//Operaciones sobre ventanas de alerta
//cambiamos el foco a la ventana de alerta
Alert alert = driver.switchTo().alert();
//podemos mostrar el mensaje de la ventana
String mensaje = alert.getText();
//podemos pulsar sobre el botón OK (si lo hubiese)
alert.accept();
//podemos pulsar sobre el botón Cancel (si lo hubiese)
alert.dismiss();
//podemos teclear algún texto (si procede)
alert.sendKeys("user");
```

También puedes consultar aquí un ejemplo de uso de *Alert Box* con WebDriver:

<https://www.guru99.com/alert-popup-handling-selenium.html>

Crea un elemento de configuración Maven desde IntelliJ (ponle como nombre: ***Test Login***) para invocar al comando:

```
mvn test -Dtest=ejercicio1.sinPageObject.TestLogin
```

Simplemente tienes que rellenar el campo "Command Line" omitiendo "mvn". Recuerda también que, para no "perder" la configuración al subir nuestro proyecto a Bitbucket, tendrás que marcar "Store as Project File" (en la parte superior derecha de la ventana), y guardarlo, por ejemplo, en la carpeta ***src/test/resources/configurations/***

Nota: Si al ejecutar la *configuration* desde IntelliJ te aparece una ventana de alerta con el mensaje:

'Login' is not allowed to run in parallel. Would you like to stop the running one?

Simplemente ignora el mensaje seleccionando "Cancel".

⇒ Ejercicio 2: Tests Login usando Page Objects

Vamos a crear el paquete ***ejercicio2.conPO***, en el que implementaremos los dos casos de prueba del ejercicio anterior usando el patrón Page Object. No usaremos la clase *PageFactory*. Recuerda que, además de los drivers, tienes que implementar la/s Page Object de las que dependen los tests, y que éstas estarán en *src/main*, tal y como hemos explicado en clase.

Los casos de prueba se implementarán en una clase ***TestLogin***, y los nombres de cada caso de prueba serán los mismos que en el ejercicio anterior: *test_Login_Correct()* y *test_Login_Incorrect()*.

En este caso necesitaréis implementar dos *page objects*: *LoginPage* y *ManagerPage*.

Para ejecutar los tests crea un elemento de configuración Maven desde IntelliJ (ponle como nombre = ***Test LoginPO***) con el comando:

```
mvn test -Dtest=ejercicio2.conPO.TestLogin
```

Recuerda marcar "Store as Project File" para guardar la configuración en la carpeta *test/resources/configurations*.

⇒ Ejercicio 3: Tests NewClient

Vamos a crear el paquete ***ejercicio3.conPOyPFact***, en el que implementaremos dos nuevos casos de prueba usando el patrón *Page Object*, junto con la clase *PageFactory*.

Necesitarás implementar las Page Objects necesarias en *src/main*, pero en el mismo paquete que nuestros drivers, en este caso serán: *ManagerPage* y *NewCustomerPage*

No vamos a necesitar la clase *LoginPage* ya que vamos a ejecutar los tests de forma que nos "saltemos" el paso de loguearnos en el sistema. Para ello usaremos el método *Cookies.storeCookiesToFile(login, password)* invocándolo UNA sóla vez y antes de ejecutar cualquier test.

Posteriormente y antes de ejecutar cada test, tendremos que guardar las cookies en el navegador invocando al método *Cookies.loadCookiesFromFile(driver)*. Esto lo haremos antes de acceder a la página inicial, que en este caso, no será la página de login, sino que directamente será la página inicial del Manager (<http://demo.guru99.com/V5/manager/Managerhomepage.php>)

La clase *Cookies* estará en *src/main* junto con las *Page Objects*.

Los casos de prueba que vamos a implementar son los siguientes (en la clase ***TestNewClientCookies***):

- ***testTestNewClientOk()***: después de haber accedido al banco con nuestras credenciales (este paso no será necesario ya que estaremos identificados por el sistema gracias a las cookies que hemos guardado en el el fichero), daremos de alta a un nuevo cliente, comprobaremos que el proceso termina correctamente (gracias a un mensaje en la pantalla del navegador), y finalmente regresaremos a la pantalla inicial del Manager (lo cual tendremos también que comprobar).

Como regla general, SIEMPRE que cambiemos de pantalla en el navegador, debemos incluir el *Asssert* correspondiente para asegurarnos de que estamos en la página correcta.

IMPORTANTE!!

Cuando creamos un nuevo cliente, la aplicación le asigna de forma automática un identificador único. Dicho valor será necesario para realizar cualquier operación con el cliente (por ejemplo editar sus datos, borrarlo...). Por lo tanto, deberás almacenar dicho valor (como un atributo en la clase que contiene los tests) para poder consultarlo desde todos los tests que necesiten trabajar con dicho cliente. También puedes mostrar dicho valor por pantalla durante la ejecución del test.

Para recordar los datos del nuevo cliente, vamos a realizar una captura de pantalla durante la ejecución del test, una vez que enviemos los datos del nuevo cliente al servidor a través del botón del formulario correspondiente.

Para generar una captura de pantalla y guardarla en un fichero (puedes usar el nombre que consideres oportuno) podemos hacerlo con:

```
File scrFile = ((TakesScreenshot) driver).getScreenshotAs(OutputType.FILE);  
File destFile = new File("./target/"+filename+".png");  
Files.copy(scrFile.toPath(),destFile.toPath(),StandardCopyOption.REPLACE_EXISTING);
```

Para poder reutilizar estas líneas si fuese necesario hacer más capturas de pantalla, hemos proporcionado (en la carpeta de plantillas) el método `ScreenShot.captura(driver,filename)` con dicho código.

La clase `ScreenShot` deberá estar en `src/main` junto con las *Page Objects*

Los datos concretos para el nuevo cliente, pueden ser éstos:

- Nombre del cliente: tu login del correo electrónico de la ua
- Género: "m" (o "f")
- Fecha de nacimiento: tu fecha de nacimiento en formato "aaaa-mm-dd" (*Ver nota)
- Dirección: "Calle x"
- Ciudad: "Alicante", (para el campo State puedes poner el mismo)
- Pin: "123456" (puedes poner cualquier otra secuencia de 6 dígitos)
- Número de móvil: "99999999"
- e-mail: tu email institucional
- password: "123456", (o cualquier otra secuencia de dígitos)

Una vez introducidos los datos pulsaremos sobre el botón "Submit" (asegúrate de que haces scroll de la pantalla para que el botón submit esté visible, lo explicamos a continuación).

El resultado del test será "pass" si hemos conseguido crear el nuevo cliente. Para ello podemos comprobar que en la nueva página aparece el texto (o una parte de él) "Customer Registered Successfully!!!".

NOTA: Para introducir la fecha usaremos tres veces el método `sendKeys` con el día, el mes y el año, respectivamente. Ten en cuenta que el formato para el día será "dd", para el mes "mm" y para el año "yyyy". Por ejemplo, "06", "05", "2003"

Cómo realizar "scroll" de la ventana del navegador

Cuando vamos a introducir los datos del nuevo cliente que queremos crear, observa que no "caben" en la pantalla todos los cuadros de texto, y que para introducir todos los datos, necesitaremos hacer "scroll" en la ventana del navegador (de no hacerlo así obtendremos un error, puesto que no podemos interactuar con elementos de la página que no están visibles).

Para ello necesitaremos usar el método javascript `scrollIntoView()`. A continuación mostramos un ejemplo en el que hacemos scroll del contenido de la ventana hasta que sea visible el webelement al que hemos denominado "submit" y que representa el botón para enviar el formulario con los datos del cliente.

```
...
//Si no hacemos esto, el botón submit no está a la vista y
//no podremos enviar los datos del formulario
JavascriptExecutor js = (JavascriptExecutor) driver;
//This will scroll the page till the element is found
js.executeScript("arguments[0].scrollIntoView();", submit);
//ahora ya está visible el botón en la página y ya podemos hacer click sobre él
submit.click();
...
```

Puedes consultar aquí un ejemplo de uso de *Scroll* con *WebDriver*:

<https://www.guru99.com/scroll-up-down-selenium-webdriver.html>

Otras observaciones a tener en cuenta:

- ➔ Recuerda que si usamos el patrón Page Object nuestros tests NO deben contener código *Webdriver*.
- ➔ Recuerda que debes verificar, cada vez que cambiemos de página, que estamos en la página correcta, para ello puedes utilizar el título de la página o alguna información de la misma, por ejemplo el login del administrador (en el caso de la página resultante de hacer login).

- ➔ El campo e-mail del cliente es único para cualquier cliente. Si intentamos dar de alta un cliente con un e-mail ya registrado, la aplicación nos mostrará un mensaje indicando que no se puede repetir el valor de dicho campo.
- ➔ Adicionalmente, para poder repetir la ejecución del test debemos “eliminar” el cliente que acabamos de crear después de ejecutar con éxito el test. Para ello tendrás que crear una nueva *page object*, que puedes llamar “DeleteCustomerPage” y que representa la página de nuestra aplicación para borrar un cliente. Una vez implementada la nueva *page object*, modifica el test para borrar el nuevo cliente creado (tendrás que utilizar el identificador del cliente para poder borrarlo), y además aceptar los mensajes de dos alertas que nos aparecerán para confirmar que queremos borrar dicha información.
- ➔ El borrado del cliente puedes hacerlo en un método privado que invocarás desde el método anotado con `@AfterEach`.
- **testTestNewClientDuplicate()**: en este segundo test, se trata de crear un nuevo cliente usando un e-mail que ya existe. En este caso, después de introducir los datos del cliente y pulsar el botón “Submit” nos debe aparecer un mensaje de alerta con el mensaje “Email Address Already Exist !!”. Para garantizar que el cliente que introducimos como entrada ya existe, debes añadirlo previamente, y a continuación, repetir la operación para asegurarnos de que efectivamente se trata de un cliente repetido. Debemos comprobar que la aplicación nos muestra el mensaje anterior y que no nos deja insertar dos veces el mismo cliente. Al igual que antes, debes borrar los datos del cliente creado después de ejecutar el test para poder repetir su ejecución con las mismas condiciones.

Puedes crear un elemento de configuración Maven desde IntelliJ (nombre = **TestNewClientCookies**) con el comando:

```
mvn test -Dtest=TestNewClientCookies
```

Recuerda marcar “Store as Project File” para guardar la configuración en la carpeta *test/resources/configurations*.

Modo “headless”

Podemos ejecutar los tests sin necesidad de mostrar el navegador, con lo cual ahorraremos tiempo de ejecución. Para ello tenemos que configurar el driver en modo *headless*. La idea es cambiar el modo de ejecución de los tests desde el proceso de construcción. Para ello tendremos que:

- Modificar la configuración del plugin **surefire**. Vamos a añadir una “propiedad” en la etiqueta `<systemPropertyVariables>`. Llamaremos a esta propiedad “**chromeHeadless**” (es una elección personal, podemos elegir cualquier nombre que consideremos oportuno). El valor de la propiedad “**chromeHeadless**” tendrá como valor el de la `<property>` **headless.value** (de nuevo este nombre depende de nuestra elección personal).

```
<properties>
...
<headless.value>false</headless.value>
</properties>

<plugin>
...
<artifactId>maven-surefire-plugin</artifactId>
...
<configuration>
  <systemPropertyVariables>
    <chromeHeadless>${headless.value}</chromeHeadless>
  </systemPropertyVariables>
</configuration>
</plugin>
```

- En el código del test (en el método anotado con `@BeforeEach`) creamos la instancia de *ChromeDriver* activando el modo *headless*. Para ello usaremos el valor de la propiedad *chromeHeadless* que hemos definido en la configuración del plugin **surefire**.


```

ChromeOptions chromeOptions = new ChromeOptions();
//recuperamos el valor de la propiedad chromeHeadless definida en surefire
boolean headless = Boolean.parseBoolean(System.getProperty("chromeHeadless"));
//el método setHeadless() cambia la configuración de Chrome a modo headless
chromeOptions.setHeadless(headless);
//ahora creamos una instancia de ChromeDriver a partir de chromeOptions
driver = new ChromeDriver(chromeOptions);

```

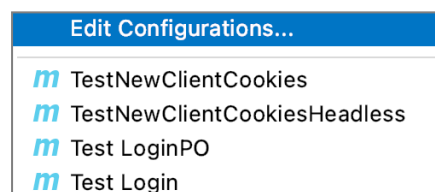
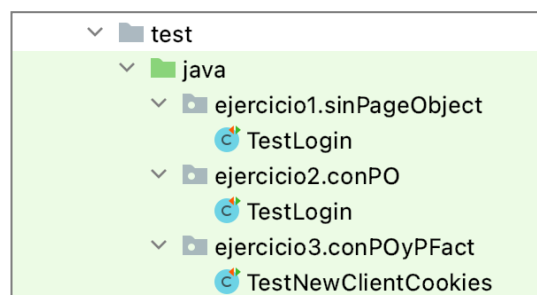
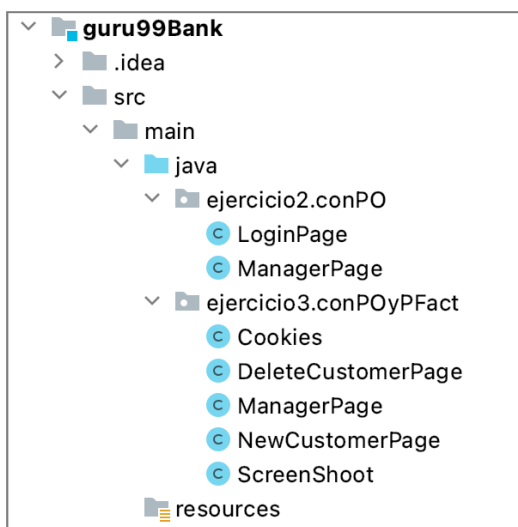
- Ahora podemos activar el modo headless usando el valor "true" para la *property* *headless.value* de nuestro *pom*.

```
mvn test -Dtest=TestNewClientCookies -Dheadless.value=true
```

Crea un elemento de configuración Maven desde IntelliJ (nombre = **TestNewClientCookiesHeadless**) con el comando anterior y pruébalo. Observarás que las capturas de pantalla se realizan con independencia de si estamos ejecutando nuestros tests en modo *headless* o no.

Recuerda marcar "Store as Project File" para guardar la configuración en la carpeta *test/resources/configurations*.

Finalmente, mostramos capturas de pantalla del proyecto maven para esta práctica, así como el conjunto de "configurations" que debéis crear::



Resumen



¿Qué **conceptos** y **cuestiones** me deben quedar CLAROS después de hacer la práctica?



AUTOMATIZACIÓN DE PRUEBAS DE ACEPTACIÓN

- Consideraremos el caso de que nuestro SUT sea una aplicación web, por lo que usaremos webdriver para implementar y ejecutar los casos de prueba. La aplicación estará desplegada en un servidor web o un servidor de aplicaciones, y nuestros tests accederán a nuestro SUT a través de webdriver, que será nuestro intermediario con el navegador (en este caso usaremos Chrome, junto con el driver correspondiente)
- Si usamos webdriver directamente en nuestros tests, éstos dependerán del código html de las páginas web de la aplicación a probar, y por lo tanto serán muy "sensibles" a cualquier cambio en el código html. Una forma de independizarlos de la interfaz web es usar el patrón PAGE OBJECT, de forma que nuestros tests NO contendrán código webdriver, independizándolos del código html. El código webdriver estará en las page objects que son las clases que dependen directamente del código html, a su vez nuestros tests dependerán de las page objects.
- Junto con el patrón Page Object, podemos usar la clase PageFactory para crear e inicializar los atributos de una Page Object. Los valores de atributos se inyectan en el test mediante la anotación @FindBy, y a través del localizador correspondiente. Esta inyección se realiza de forma "lazy", es decir, los valores se inyectan justo antes de ser usados.
- Con webdriver podemos manejar las alertas generadas por la aplicación a probar, introducir esperas (implícitas y explícitas), realizar scroll en la pantalla del navegador, agrupar elementos, capturar la pantalla del navegador y manejar cookies, entre otras cosas.
- El manejo de las cookies del navegador nos será útil para acortar la duración de los tests, ya que podremos evitar loguearnos en la aplicación para probar determinados escenarios.
- También podremos acortar los tiempos de ejecución de nuestros tests si los ejecutamos en modo headless. Podemos decidir si vamos a ejecutar o no nuestros tests en modo headless aprovechando la capacidad del plugin surefire y/o failsafe, de hacer llegar a nuestros tests ciertas propiedades definidas por el usuario. De esta forma podremos, cuando lancemos el proceso de construcción, ejecutar en ambos modos sin modificar el código.

P09- Pruebas de aceptación: JMeter

Os recordamos que el plazo para entregar los ejercicios de esta práctica (P09) termina justo ANTES de comenzar la siguiente sesión de prácticas (P08) en el laboratorio (los grupos de los lunes, por lo tanto, el próximo lunes, los de los martes, el próximo martes, ...). Pasado este plazo los profesores no revisarán vuestro trabajo sobre P09 (de forma individual), con independencia de que se indiquen en clase las soluciones que os permitirán saber si las vuestras son correctas.

Pruebas de aceptación de aplicaciones Web

El objetivo de esta práctica es automatizar pruebas de aceptación de pruebas emergentes NO funcionales sobre una aplicación Web. Utilizaremos la herramienta de escritorio **JMeter**.

Usaremos una aplicación web: JPetStore. Proporcionamos la secuencia de entradas de nuestro driver, que nos permitirán evaluar si nuestro sistema soporta una determinada carga de usuarios (pruebas de carga)

Bitbucket

El trabajo de esta sesión también debes subirlo a *Bitbucket*. Todo el trabajo de esta práctica deberá estar en el directorio **P09-JMeter** dentro de tu espacio de trabajo, es decir, dentro de tu carpeta: ppss-2021-Gx-apellido1-apellido2.

Aplicación JMeter

En la máquina virtual hemos instalado JMeter 5.4. Esta versión para linux tiene un "bug" de forma que al abrir un plan JMeter aparece una ventana emergente con mensaje "Unexpected Error". Al abrir la ventana de log (pulsando sobre el error a la derecha del icono con forma de triángulo) veremos que se ha producido una excepción debido a un puntero nulo al cargar el archivo. El archivo se ha cargado correctamente, y simplemente podemos "limpiar" la ventana de log pulsando sobre el icono "Clear all" y continuar trabajando normalmente.

Dado que es molesto tener que hacer esto cada vez que abramos un fichero .jmx, vamos a descargar la última versión (5.4.1), en la que han reparado este "bug".

Accede a la página de JMeter: https://jmeter.apache.org/download_jmeter.cgi y descarga el BINARIO **apache-jmeter-5.4.1.tgz**

Descomprímelo en tu \$HOME, verás que se crea la carpeta *apache-jmeter-5.4.1*

A continuación edita el fichero .profile de tu \$HOME, y edita la última línea:

```
export PATH=$PATH:/home/ppss/apache-jmeter-5.4.1/bin
```

Edita el fichero /home/ppss/Desktop/jmeter.desktop y edita las siguientes líneas:

```
Name=Jmeter 5.4.1
```

```
Icon=/home/ppss/apache-jmeter-5.4.1/docs/images/jmeter_square.png
```

Finalmente cierra la sesión para que los cambios en el .profile tengan efecto.

Vuelve a entrar y comprueba que al hacer doble click sobre el icono del escritorio accedes a la versión 5.4.1 de JMeter. Puedes borrar el directorio apache-jmeter-5.4 y todo su contenido.

Aplicación web para los ejercicios de esta sesión

Utilizaremos una aplicación Web Java JPetStore que ya conocemos de la práctica P07, en la cual realizamos pruebas de propiedades emergentes funcionales con Selenium IDE.

Os recordamos el comando para empaquetar y desplegar nuestra aplicación web (si ya la habías desplegado en la práctica anterior puedes omitir este comando, la aplicación seguirá estando "alojada" en el servidor de aplicaciones wildfly):

```
> mvn package cargo:run -P wildfly21
```

El comando para arrancar el servidor de aplicaciones y ejecutar nuestra aplicación es:

```
> mvn cargo:run -P wildfly21
```

Es importante NO tener abierto IntelliJ ni ningún otro proceso adicional que no sea absolutamente necesario, ya que interferirán en nuestro proceso de pruebas, desvirtuando el resultado obtenido.

Para ejecutar la aplicación usaremos la siguiente url (en esta práctica vamos a usar el navegador Firefox):

```
http://localhost:8080/jpetstore
```

Ejercicios

Para iniciar JMeter puedes usar el icono del escritorio o también ejecutando desde el terminal:

```
> jmeter
```

Nota1: Cuando arrancamos JMeter, por defecto la apariencia de la aplicación es en modo “*Darcula*”, con lo cual tendrá un aspecto bastaste oscuro. Si preferís cambiarlo a otro más claro, podéis hacerlo desde *Options*→*Look and Feel*, y seleccionar por ejemplo “*IntelliJ*”.

Nota2: En nuestros tests, vamos a trabajar con un usuario “z” que supondremos que tiene ya una cuenta en la tienda de animales. Por lo tanto, antes de implementar los drivers, debes crear dicha cuenta para el usuario “z”, puedes poner como contraseña “z”, y rellenar el resto de campos como consideres.

🔗🔗 Ejercicio 1: Uso de Proxies en JMeter

Antes de comenzar a implementar nuestros tests, vamos a explicar el **uso de “proxies” en JMeter** para averiguar los parámetros de una petición al servidor; cuando éstos no son visibles en la url correspondiente.

La creación de un plan de pruebas con JMeter puede presentar cierta dificultad cuando se ven implicadas *queries* y/o formularios complejos, **peticiones https POSTs**, así como peticiones javascript, en las que los parámetros que se envían por la red NO son visibles en la URL.

JMeter proporciona un **servidor HTTP proxy**, a través del cual podemos utilizar el navegador para realizar las pruebas, y JMeter “grabará” las peticiones http generadas, tal y como se enviarán al servidor, y creará los correspondientes HTTP *samplers*. Una vez que hayamos “guardado” las peticiones HTTP que necesitemos, podemos utilizarlas para crear un plan de pruebas.

Vamos a ver paso a paso cómo utilizar dicho servidor *proxy*:

A) JMeter “grabará” todas las acciones que realicemos en el navegador en un controlador de tipo “recording”, por lo que tendremos que incluir un controlador de este tipo en nuestro plan de pruebas. Lo primero que haremos será **añadir un grupo de hilos** en nuestro plan. Desde el menú

contextual del nodo "Test Plan", elegiremos *Add→Threads (Users)→Thread Group*. A continuación, desde el menú contextual del grupo de hilos que acabamos de añadir tendremos que seleccionar *Add→Logic Controller→Recording Controller*.

- B) Ahora vamos a **añadir un servidor proxy** para realizar peticiones HTTP. Desde el menú contextual del elemento *Test Plan*, seleccionando *Add→NonTest Elements→HTTP(S) Test Script Recorder*. En la configuración del *proxy* tendremos que asegurarnos de que el **puerto** no está ocupado con algún otro servicio en nuestra máquina local, (ver el valor del campo *Global Settings→Port* de la configuración del elemento *HTTP(S) Test Script Recorder*, por defecto tiene el valor 8888). En nuestro caso, los puertos 8080 y 9990 están ocupados por el servidor de aplicaciones Wildfly, por lo que en principio el puerto 8888 estará libre.

**Cómo averiguar
qué puertos están
siendo utilizados**

Para averiguar qué conexiones están activas y qué procesos y puertos están abiertos podemos usar el comando:

> **ss -tap** (<https://www.binarytides.com/linux-ss-command/>)

De forma alternativa, también puedes usar el comando:

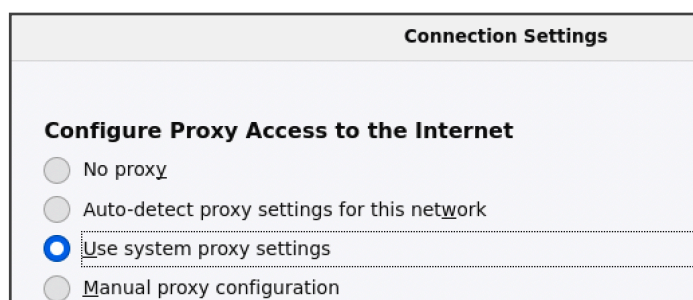
> **lsof -i tcp**

Por defecto, el servidor proxy (elemento *HTTP(S) Test Script Recorder*) interceptará todas las peticiones http dirigidas a nuestra aplicación y las "grabará" en un controlador de tipo **recording** (ver el valor del campo *Test plan content→Target Controller*, en la pestaña *Test Plan Creation* de la configuración del *HTTP(S) Test Script Recorder*). Ya hemos incluido el controlador de grabación en nuestro plan. Por defecto, JMeter grabará cualquier "cosa" que envíe o reciba el navegador, incluyendo páginas HTML, ficheros javascript, hojas de estilo css, imágenes, etc, la mayoría de las cuales no nos serán de mucha utilidad para nuestro plan de pruebas. Para "filtrar" el tipo de contenido que queremos (o no queremos) utilizaremos patrones URL a Incluir (o a Excluir), desde la pestaña **Request Filtering** de la configuración del Servidor Proxy.

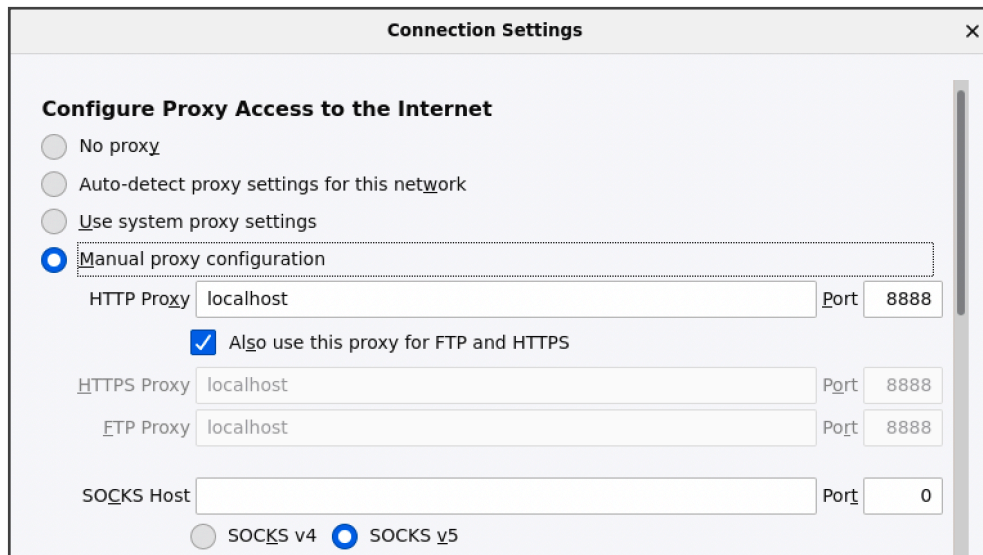
Por ejemplo, vamos a **excluir** del proceso de grabación las **imágenes** y las **hojas de estilo** (desde la pestaña *Request Filtering*, y más concretamente desde el elemento *URL Patterns to exclude*). Para ello tendremos que pulsar con el ratón sobre el botón *Add*, y a continuación hacer doble click en la nueva fila añadida (en color blanco). Tenemos que añadir dos líneas, y escribiremos las expresiones regulares **.gif* y **.css* como patrones URL a excluir (cada patrón en una línea diferente). NO hay que poner comillas al introducir cada patrón. Para más información sobre expresiones regulares podéis consultar: <http://rubular.com>.

- C) A continuación tendremos que **configurar el navegador** (en este caso Firefox) para utilizar el puerto en donde actuará el proxy JMeter. Para ello iremos al menú de "Preferencias→General→Network settings→Settings...". Desmarcamos la opción actual ("Use system proxy settings"), y marcamos "Manual proxy configuration". Como valor de Proxy HTTP pondremos **localhost**, y el puerto el **8888**. Acuérdate de marcar la casilla "Also use this proxy for FTP and HTTPS".

Las siguientes capturas de pantalla muestran la configuración Inicial de Firefox, y la configuración que debemos usar para utilizar el proxy JMeter.

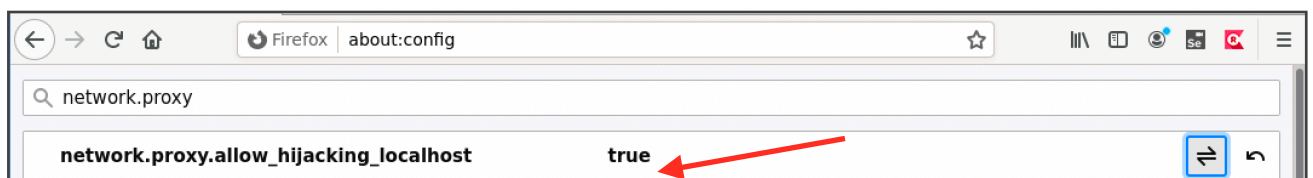


Configuración Inicial de Firefox



Configuración de Firefox para usar el proxy de JMeter

Finalmente, y dado que las conexiones con localhost o 127.0.0.1 nunca se establecen a través de un proxy, tendremos que cambiar el valor de la variable `network.proxy.allow_hijacking_localhost` a `true`, desde [about:config](#)



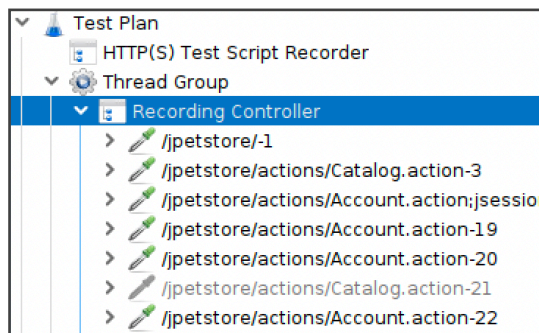
Valor de la propiedad `network.proxy.allow_hijacking_localhost` para usar el proxy de JMeter

Ahora ya estamos en disposición de usar el *proxy* de JMeter para ver cuáles son las peticiones http que tenemos que utilizar para “loguearnos” en el sistema. La petición de *login* es una petición http de tipo POST, y los parámetros de esta petición NO son visibles en el navegador, por lo que averiguaremos dichos valores a través del *proxy*.

- D) Una forma de asegurarnos de que el navegador está utilizando el proxy de JMeter es: estando “parado” el proxy intentar navegar desde Firefox (o acceder a nuestra aplicación web). Si el navegador sigue sirviendo las páginas es que NO está utilizando el proxy. Una vez que nos hemos asegurado de que hemos configurado adecuadamente Firefox para utilizar el proxy de JMeter, iremos al panel de configuración de nuestro *Test Script Recorder* y en el panel “State” veremos el botón “Start”. Al pulsar dicho botón **pondremos en funcionamiento el proxy** y podemos comenzar la grabación (no podremos arrancar el proxy si previamente no hemos añadido el controlador de grabación en nuestro plan).
- E) Al arrancar el proxy nos aparecerá una alerta indicando que se ha creado un certificado temporal en un directorio de JMeter (simplemente tenemos que pulsar sobre “OK” en dicha ventana). También veremos una ventana con el nombre “Recorder: Transactions Control” que permanecerá abierta hasta que detengamos el proxy. Ahora, cualquier acción que hagamos desde el navegador, quedará “registrada” en el controlador de grabación que hemos añadido en nuestro plan de pruebas. Vamos a realizar las siguientes acciones en el navegador:
- Entramos en la aplicación de la tienda de animales (<http://localhost:8080/jpetstore/actions/Catalog.action>), a continuación pinchamos sobre Sign In.
 - Introducimos las credenciales del usuario z que ya tenemos creado y pulsamos sobre “Login”. **Nota:** si no has creado antes el usuario tal y como te hemos indicado, hazlo ahora, aunque entonces se grabarán también esas acciones, las cuales no vamos a necesitar en nuestro driver. No te preocupes, porque podrás borrarlas.
 - Finalmente pulsamos sobre “Sign Out” en nuestra aplicación web y PARAMOS la grabación desde el proxy de JMeter.

Ahora podemos ver las peticiones http que han quedado grabadas en el “Controlador Grabación” de nuestro plan. Selecciona cada una de ellas y observa la ruta y parámetros de petición (**comprueba que cuando hacemos “login” en el sistema se envían 5 parámetros, y que es una petición POST**). Utilizaremos esta información en el siguiente ejercicio para confeccionar nuestro plan de pruebas.

Observarás que hay algunas peticiones que están deshabilitadas (aparecen en gris claro). Se trata de redirecciones realizadas por la aplicación, que también quedan grabadas, pero que no se tendrán en cuenta cuando ejecutemos el test.



Nuestro plan de pruebas JMeter cuando finalicemos el ejercicio tendrá un aspecto similar al mostrado en la imagen de la izquierda. En nuestro caso hemos grabado también las acciones de creación de un nuevo usuario.

Observaciones sobre el CONTROLADOR DE GRABACIÓN: Cuando ejecutemos el plan de pruebas, el “Recording Controller” no tiene ningún efecto sobre la lógica de ejecución de las acciones que contiene, es como el controlador “Simple”, es decir, actúa como un nodo que “agrupa” un conjunto de elementos.

Guarda el plan de pruebas que hemos creado, lo puedes hacer desde “File→Save”, o pulsando sobre el icono con forma de *disquette* y ponle el nombre **Plan-Ejercicio1-proxy.jmx**

Una vez terminado el ejercicio recuerda volver a la configuración inicial de red en Firefox para poder acceder a Internet sin utilizar el proxy de JMeter. Además tendrás que volver a poner a false el valor de la variable `network.proxy.allow_hijacking_localhost` a true, desde [about:config](#).

⇒ ⇒ Ejercicio 2:Driver y validación de carga

Vamos a crear un nuevo plan de pruebas en el que usaremos la información obtenida en el ejercicio anterior. Puedes duplicar el jmx del ejercicio anterior, y ponerle como nombre **Plan-Ejercicio2**. Queremos evaluar el rendimiento de nuestra aplicación y comprobar si es capaz de soportar una determinada carga de usuarios.

Llamaremos a nuestro plan de pruebas “**Plan JMeter**”. Lo primero que haremos será añadir un **grupo de hilos**. En este caso, ya lo tenemos del ejercicio anterior. Como ya hemos visto, JMeter trabaja simulando un conjunto de usuarios concurrentes realizando varias tareas sobre la aplicación. Cada usuario simulado se representa como un hilo. Por lo tanto, el número de hilos representa el número de usuarios simultáneos generados por el grupo de hilos. De momento dejaremos los valores por defecto: un hilo, un periodo de subida de 1 y un bucle con una única iteración. Borraremos el elemento *HTTP(S) Test Script Recorder* puesto que ya no nos hace falta. A continuación añadiremos los siguientes elementos:

A) Primero vamos a incluir dos **elementos de configuración** en nuestro plan de pruebas. Los elementos de configuración evitan que tengamos que introducir repetidamente alguna información de configuración de los *samplers*. Por ejemplo, si en el plan de pruebas incluimos varias llamadas a algunas páginas que están protegidas mediante una autenticación http básica, se podría compartir la información de usuario y password, para no tener que repetir estos datos para cada *sampler* dentro del plan de pruebas. Para pruebas de aplicaciones Web, el elemento de configuración más importante es “**HTTP Request Defaults**”. Lo añadimos desde el menú contextual del grupo de hilos (*Add→ Config Element→HTTP Request Defaults*). Estableceremos el nombre del servidor a **localhost**, y el número de puerto a **8080**.

Otro elemento de configuración muy útil es “**HTTP Cookie Manager**”. Éste almacena cookies y hace que estén disponibles para subsecuentes llamadas al mismo sitio, tal y como haría un navegador. Dado que cada hilo representa un usuario diferente, las cookies no se compartirán entre hilos. Lo añadiremos desde el menú contextual del grupo de hilos (*Add→ Config Element→HTTP Cookie Manager*). En este caso usaremos la configuración por defecto de este elemento.

Si nos equivocamos al crear un elemento, en cualquier momento podemos borrar/modificar/mover cualquier elemento del plan. Para **borrar** un elemento lo haremos desde el menú contextual de dicho elemento. Para **modificar** su configuración simplemente seleccionaremos dicho elemento. Para **mover** un elemento a otra posición dentro del plan, seleccionaremos y arrastraremos dicho elemento a su nueva posición.

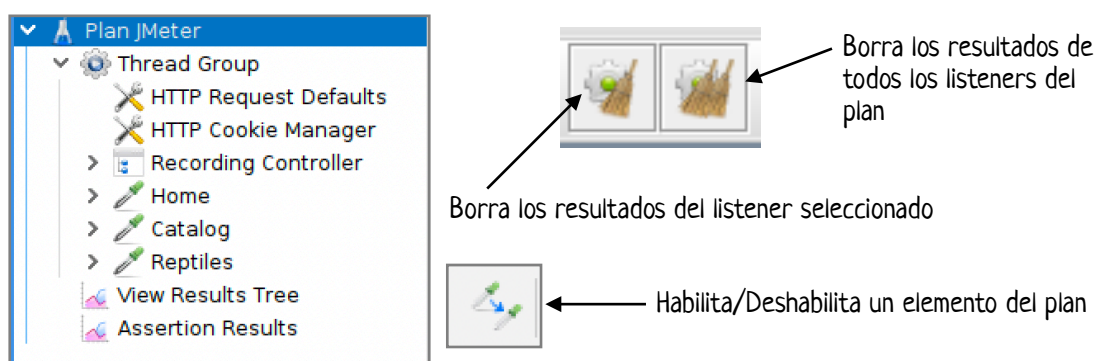
- B) Ahora vamos a añadir un **sampler** para hacer peticiones de páginas **http** a nuestro servidor de aplicaciones y así simular la acción de los usuarios (desde el menú contextual del grupo de hilos: **Add→Sampler→HTTP Request**). La URL inicial de la tienda es "http://localhost:8080/jpetstore". Se trata de una petición GET. El servidor y el puerto ya los hemos indicado en un elemento de configuración (localhost y 8080), por lo que no es necesario indicarlos. Después del contexto inicial (cuyo valor es *jpetstore*, y es donde se encuentra desplegada nuestra aplicación web), vendría el nombre del recurso, en este caso, la página html a la que queremos acceder. Observamos que no aparece esta información (no es necesaria ya que por defecto se accederá a index.html). Por lo tanto, lo único que haremos será indicar en el campo *Path* el valor **"/jpetstore/"** (no hay que poner las comillas). Vamos a hacer uso de varios *samplers* http, por lo que será conveniente indicar un nombre adecuado para poder diferenciarlos y hacer más "legible" nuestro código. Por ejemplo, para este primer *sampler* podemos especificar como nombre el valor **"Home"**
- C) Para asegurarnos de que estamos en la página correcta, vamos a añadir una **aserción** (desde el menú contextual de la petición HTTP Home: **Add→Assertions→Response Assertion**). Nos aseguraremos de marcar **"Text Response"**, y el patrón a probar contendrá la cadena **"Welcome to JPetStore 6"**. Recuerda que primero tienes que pinchar sobre el botón **"Add"**, y después hacer doble click en la fila en blanco añadida para editar su contenido. El texto se pone SIN comillas. Otra opción es copiar el texto anterior (con ctrl-C) y directamente seleccionar el botón **Add From Clipboard**.
- D) Ahora añadiremos **otra petición http** GET (desde el grupo de hilos), en este caso, puesto que la página mostrada nos muestra el catálogo de animales, podemos asignarle como nombre **"Catalog"**. Utiliza el navegador y pincha en el hipervínculo ("enter the Store") para ver la URL de la nueva petición. En este caso puedes ver que la ruta tendrá el valor **"/jpetstore/actions/Catalog.action"**. para verificar que entramos en la página correcta añadiremos una aserción igual que en el caso anterior con el patrón a probar **"Saltwater"**.
- E) Desde la página anterior, vamos a acceder al menú de Reptiles (por ejemplo pinchando sobre la imagen correspondiente). Tendremos, por lo tanto, que añadir otra **petición http** (le pondremos el nombre **"Reptiles"**). Podemos comprobar que la URL del navegador, es la siguiente: **"http://localhost:8080/jpetstore/actions/Catalog.action?viewCategory=&categoryId=REPTILES"** En este caso, vemos que la petición tiene dos parámetros (los parámetros se especifican con pares nombre=valor, separados por '&'): un primer parámetro con nombre *viewCategory* (que en este caso no tiene valor asociado, pero que es necesario incluir), y un segundo parámetro con nombre *categoryId* (cuyo valor es REPTILES). Por lo tanto, indicaremos, además de la **ruta** (*/jpetstore/actions/Catalog.action*), los **parámetros** anteriores con el valor correspondiente para *categoryId*. Acuérdate de que primero tienes que pulsar el botón añadir, y hacer doble click en las entradas añadidas para introducir los valores. No tienes que poner comillas). Añade una **aserción de respuesta textual** con el patrón **"Reptiles"**. Observa que en la configuración del elemento Response Assertion aparece al final: **"Custom Failure Message"**. Puedes incluir mensajes en todas la aserciones, los cuales se mostrarán si la aserción falla.

Nota: Como acabamos de ver, cuando utilizamos **samplers http**, para averiguar la "ruta" y parámetros de la petición URL podemos fijarnos en la ruta que aparece en el navegador al ejecutar la aplicación. Esta aproximación funciona bien cuando se trata de peticiones GET "simples" y parámetros "fáciles de manipular". Páginas que contengan formularios grandes pueden requerir docenas de parámetros. Además, si se utilizan peticiones HTTP POST (como por ejemplo el envío de login y password), los valores de los parámetros no aparecerán en la URL, por lo que tendremos que descubrirlos de alguna otra forma. Por ejemplo usando el **Proxy JMeter**, que grabará el caso de prueba por nosotros, tal y como hemos visto en el ejercicio 1, y que luego podremos modificar según nuestras necesidades.

- F) Antes de continuar vamos a **grabar nuestro plan** de pruebas (como medida de precaución es recomendable grabar a menudo). Lo podemos hacer desde “File→Save”. Si hemos grabado la creación del usuario “z” podemos borrarlos ya (o deshabilitarlos, desde el menú contextual de cada elemento. Asegúrate de que no borras todavía el sampler que contiene la petición POST, ya que la necesitaremos en breve). Dado que podemos ejecutar el plan en cualquier momento, vamos a hacerlo ahora, pero antes nos aseguraremos de que en las propiedades del grupo de hilos solamente se genera un hilo, con un periodo de subida de 1 segundo y una única vez. Pulsamos el icono con un triángulo verde (el primero de ellos).

Para poder “ver” lo que ocurre al ejecutar las pruebas es imprescindible añadir uno o varios “**Listeners**” que nos muestren (y/o graben los resultados de las pruebas en algún fichero). Utilizaremos, de momento, dos de ellos. Desde el menú contextual del plan de pruebas elegimos **Add→Listener→Assertion Results**. También añadiremos un segundo Listener: **View Results Tree**. El primero de ellos nos mostrará en pantalla las etiquetas de los samplers ejecutados, y los resultados de las aserciones de nuestro plan, para cada uno de los samplers. El segundo nos muestra las peticiones y respuestas de cada uno de los *samplers*.

Para **ejecutar el plan** tienes que ir al menú de JMeter y seleccionar **Run→Start** (también puedes utilizar el icono con forma de **triángulo verde**). En el extremo derecho de la barra de herramientas de JMeter verás un pequeño cuadrado. Durante la ejecución del plan, verás que el cuadrado se muestra de color verde. Cuando acaba la ejecución vuelve a estar gris. Selecciona cualquiera de los listeners que hemos añadido y observa los resultados. Si ejecutas el plan con un listener seleccionado, verás cómo se va actualizando durante la ejecución. Realiza varias ejecuciones para familiarizarte con la información que proporcionan estos dos listeners. (si todas las aserciones se evalúan a “true” verás que el listener correspondiente no muestra ninguna información. Prueba a cambiar la aserción para que se produzca un “fallo”, y podrás observar que el listener muestra el error producido). Verás que los listeners muestran los resultados de la ejecución “a continuación” de los resultados de ejecuciones anteriores. Podemos “limpiar” el panel de resultados pinchando sobre el icono correspondiente. A continuación mostramos el aspecto del plan de pruebas hasta este momento, así como la imagen del icono para “limpiar” los resultados de los listeners de ejecuciones anteriores.



Plan de pruebas hasta este momento

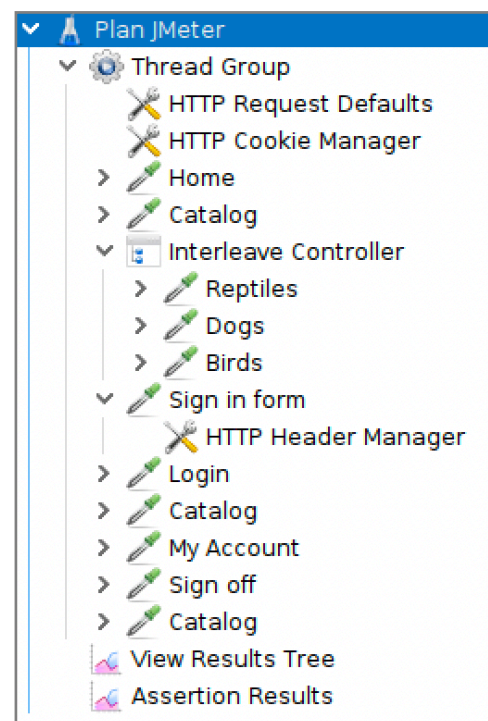
Antes de continuar, familiarízate con la información que proporciona cada uno de los listeners (mensajes http de petición y respuesta y resultados de las aserciones). Puedes probar a mover los listeners a otro nivel del plan (por ejemplo como hijos de algún sampler) y verás lo que ocurre. Observa también que la información mostrada por los listeners no se “resetea” entre dos ejecuciones consecutivas de forma automática.

- G) Ahora añadiremos **controladores lógicos** a nuestro plan de pruebas para “alterar” el comportamiento secuencial del plan. Vamos a incluir un “Controlador Interleave” desde el menú contextual del Grupo de Hilos: **Add→Logic Controller→Interleave Controller**. Este controlador “alterna” la ejecución de uno de sus hijos en cada iteración de cada uno de los hilos. Podéis consultar el funcionamiento de este elemento y de cualquier componente de JMeter en (http://jmeter.apache.org/usermanual/component_reference.html). Situiremos el controlador justo después del elemento Catalog, e incluiremos como nodos hijo tres peticiones http. Una de ellas es la que hemos denominado Reptiles. Las otras dos son el acceso a los submenús Dogs, y Birds, respectivamente. Puedes crearlos rápidamente “duplicando” el nodo Reptiles y editando los elementos correspondientes (acuérdate de modificar también las aserciones).

H) Vamos a añadir las **peticiones http** para “loguearnos” en el sistema (como hijos del grupo de hilos), y las llamaremos “**Sign in form**”, y “**Login**” respectivamente. La primera es la petición que realizamos cuando “pinchamos” sobre el enlace “Sign in” en el navegador, y la segunda corresponde con la introducción del login y password y pinchar sobre el enlace “Login” en el navegador Firefox. Estas acciones las tenemos “grabadas” en el controlador del ejercicio anterior. Identifícalas y cópialas (o muévelas) a continuación del controlador Interleave. Seguidamente introducimos una **petición para mostrar el catálogo** (podemos, por ejemplo duplicar el elemento “Catalog” que ya teníamos antes (fíjate que, en el navegador Firefox, después de “pinchar” sobre “Login”, se llama de nuevo a la página que muestra el catálogo general de especies). El nuevo elemento “Catalog” tendrá asociada una **aserción de Respuesta** con dos patrones: el patrón “Sign Out”, y el patrón “Welcome z”. (IMPORTANTE: JMeter es sensible a las mayúsculas/minúsculas, asegúrate de que el patrón coincide exactamente con el mensaje que aparece en la página).

Añade otra **petición Http** con el nombre “**My Account**” que es la petición que se genera cuando en el navegador “pinchamos” sobre “My Account”. Asociaremos una **aserción de respuesta** con el patrón “User Information”. Finalmente añadimos una última petición para salir de nuestra cuenta (utiliza la petición que tenemos grabada) a la que llamaremos “**Sign off**”, seguida de otra petición “**Catalog**”, esta vez con una aserción de respuesta con el patrón “Sign In”.

I) Vamos a **borrar el controlador Grabación**, ya que ya no nos hace falta, simplemente lo hemos utilizado para “averiguar” la configuración de las peticiones http que teníamos que realizar. Guardamos el plan y ejecutamos (Todas las peticiones y aserciones correspondientes tiene que aparecen “en verde”. Si no es así, corrige los posibles errores en tu plan). Podemos ver los resultados en los “listeners” *Assertion Results* y *View Results Tree*. El plan resultante debe tener el aspecto mostrado en la imagen de la derecha:



J) **Añadimos dos listeners más** (desde el nodo raíz de nuestro plan de pruebas): un **Aggregate Report**, que proporciona un resumen general de la prueba; y un **Graph Results**, que “dibuja” los tiempos registrados para cada una de las muestras. El **Aggregate Report** muestra una tabla con una fila para cada petición, mostrando el tiempo de respuesta, el número de peticiones, ratio de error, ... El gráfico de resultados muestra el rendimiento de la aplicación (throughput) como el número de peticiones por minuto gestionadas por el servidor.

Vuelve a ejecutar el plan y observa los resultados mostrados por los nuevos listeners que hemos añadido (Puedes comprobar de nuevo que los resultados obtenidos por los listeners no se “resetean” de forma automática). Reduce al máximo el número de aplicaciones “abiertas” en el ordenador cuando estés ejecutando JMeter, ya que afectarán a los datos obtenidos. En un caso de prueba “real” solamente deberíamos tener en ejecución la aplicación JMeter en una máquina.

- K) Vamos a introducir pausas para simulara un comportamiento más realista. Para ello añadiremos **timers**. Añadimos un **Uniform Random Timer** desde el menú contextual del grupo de hilos. configuramos un **Constant delay offset** de 1500 ms, y un **Random delay maximum** de 100ms. Dado que el proceso de login le llevará al usuario algo más de tiempo, añadiremos otro timer del mismo tipo como hijo del sampler login. En este caso el **delay** aleatorio seguirá siendo de 100 ms máximo, mientras que el **offset** constante será de 2000ms. Para comprobar el efecto de los timers, añadiremos también el listener View Results in Table, el cual nos proporciona, para cada muestra, el tiempo en el que comienza a ejecutarse.

Ejecuta de nuevo el plan (recuerda que primero debes "limpiar" los resultados de ejecuciones previas) y comprueba en dicho listener que cada muestra se lanza teniendo en cuenta las pausas introducidas por los timers (el tiempo de comienzo de una muestra será tiempo de respuesta (sample time) de la muestra anterior + el delay impuesto por el timer. Observa que el tiempo de ejecución de la muestra no cambia al introducir los "delays" de los timers, en cambio el **throughput** sí se ve afectado por estos retrasos, ya que se calcula como: $\text{numero_peticiones} / \text{tiempo_total_ejecución}$. El tiempo total de ejecución se calcula desde el inicio de la primera petición hasta el final de la última, y tiene en cuenta los tiempos de espera.

- L) Ahora vamos a suponer que está prevista una carga de 25 usuarios concurrentes. Nuestras especificaciones estipulan que con 25 usuarios concurrentes, el tiempo de respuesta para cada uno de ellos debe ser inferior a 2ms segundos por página. Vamos a cambiar los parámetros del grupo de hilos de nuestro plan para validar si nuestra aplicación cumple con los requisitos de rendimiento acordados con el cliente.

Es importante que entiendas lo que significa el término "concurrentes": que los 25 usuarios estén siendo atendidos "a la vez". Por ejemplo, si lanzamos 100 usuarios con un ramp-up de 1 segundo, esto no garantiza que se vayan a ejecutar de forma concurrente. Supongamos que cada usuario necesita 1 ms para ejecutar el plan. Después de un segundo, todos los usuarios se habrán ejecutado, pero no de forma concurrente, ya que cuando empieza el siguiente, el anterior ya ha terminado.

Podemos comprobar el número de hilos que se están ejecutando de forma concurrente observando la gráfica "Active threads Over Time". Esta gráfica no la proporciona ningún listener. Vamos a generarla a través de un comando desde el terminal.

Para ver mejor los resultados, vamos a cambiar la "granularidad" del eje de abcisas de la gráfica. Por defecto tiene un valor de 1 minuto, nosotros usaremos sólo 3 segundos. Para ello tienes que editar el fichero `/home/ppss/jmeter5.4.1/bin/user.properties`, y modificar el valor de la siguiente propiedad (la encontrarás en la línea 76 del fichero):

```
jmeter.reportgenerator.overall_granularity=3000
```

A continuación añadimos el listener **Simple Data Writer**, que guardará en un fichero los datos calculados por JMeter durante el ejecución del plan. Vamos a generar dicho fichero con el nombre **data.csv** en la misma carpeta que nuestro plan (tendrás que poner la ruta absoluta del fichero en el cuadro de texto *Filename* de la configuración del listener).

Cambia el número de hilos del plan a 25, y usa, por ejemplo, un ramp-up de 10 segundos. Ejecuta el plan, verás que se ha generado el fichero data.csv en tu carpeta de trabajo. Ahora crearemos la carpeta **reports**, que es donde vamos a generar las páginas html con los informes de jmeter.

Ejecuta el siguiente comando:

```
> jmeter -g data.csv -o reports/
```

En la carpeta reports, abre el fichero index.html en el navegador. La gráfica que queremos ver es Charts→Over Time→Active Threads Over Time. Tienes que poder ver en la gráfica cuando los 25 hilos se están ejecutando concurrentemente. Si en la gráfica el número de hilos concurrentes es inferior a 25 entonces tendrás que ajustar los parámetros ampliando el número de iteraciones, por ejemplo. Si en la gráfica puedes ver los 25 hilos ejecutándose concurrentemente, entonces sólo tienes que comprobar en la tabla del Aggregate Report que los todos los tiempos de respuesta tiempos de respuesta para cada sampler son inferiores a 2 ms.



¿Qué **conceptos** y **cuestiones** me deben quedar CLAROS después de hacer la práctica?



DISEÑO DE PRUEBAS DE ACEPTACIÓN DE PROPIEDADES EMERGENTES NO FUNCIONALES

- Los comportamientos a probar se seleccionan teniendo en cuenta la especificación (caja negra).
- En general, las propiedades emergentes no funcionales contribuyen a determinar el rendimiento (performance) de nuestra aplicación, en cuyo caso tendremos que tener en cuenta el "perfil operacional" de la misma, que refleja la frecuencia con la que un usuario usa normalmente los servicios del sistema.
- Es muy importante que las propiedades emergentes puedan cuantificarse, por lo que deberemos usar las métricas adecuadas que nos permitan validar dichas propiedades

AUTOMATIZACIÓN DE PRUEBAS DE ACEPTACIÓN

- Usaremos JMeter, una aplicación de escritorio que nos permitirá implementar nuestros drivers sin usar código java..
- Las "sentencias" de nuestros drivers NO son líneas de código escritas de forma secuencial, sino que usaremos una estructura jerárquica (árbol) en donde los nodos representan elementos de diferentes tipos (grupos de hilos, listeners, samplers, controllers...), que podremos configurar. El orden de ejecución de dichas "sentencias" dependerá de dónde estén situados en la jerarquía los diferentes tipos de elementos
- Los "resultados" de la ejecución de nuestros test JMeter, consisten en una serie de datos calculados a partir de ciertas métricas (número de muestras, tiempos de ejecución,...), que necesariamente estarán contenidas en los listeners que hayamos usado para la implementación de cada driver.
- Los resultados obtenidos por la herramienta JMeter no son suficientes para determinar la validez o no de nuestras pruebas. Será necesario un análisis posterior, que dependerá de la propiedad emergente que queramos validar, para poder cuantificarla.

P10- Análisis de pruebas: Cobertura

Cobertura

Un análisis de la cobertura de nuestras pruebas nos permite conocer la **extensión** de las mismas. En esta sesión utilizaremos la herramienta Jacoco para analizar tanto la cobertura de líneas como de condiciones de nuestros tests.

Esta herramienta se integra con Maven, de forma que podemos incluir el análisis de cobertura en la construcción de nuestro proyecto.

Bitbucket

El trabajo de esta sesión también debes subirlo a *Bitbucket*. Todo el trabajo de esta práctica deberá estar en el directorio **P10-Cobertura** dentro de tu espacio de trabajo, es decir, dentro de tu carpeta: `ppss-2021-Gx-apellido1-apellido2`.

Recuerda que si trabajas desde los ordenadores del laboratorio primero deberás configurar git y clonar tu repositorio de Bitbucket.

📁 Ejercicio 1: Proyecto cobertura

Crea un proyecto Maven (en la carpeta `ppss-2021-Gx-apellido1-apellido2/P10-Cobertura/`) con `groupId = ppss`, y `artifactId = cobertura`. El nombre del proyecto IntelliJ será **cobertura**.

Añade la siguiente clase al paquete "ejercicio1".

```
package ejercicio1;
public class MultipathExample {
    public int multiPath1(int a, int b, int c) {

        if (a > 5) {
            c += a;
        }
        if (b > 5) {
            c += b;
        }
        return c;
    }
}
```

- A) Calcula la complejidad ciclomática (CC) para el método `multiPathExample()`. Implementa un número mínimo de casos de prueba para conseguir una **cobertura del 100% de líneas y de condiciones** para el método `multiPath1()`.
- B) Obtén un **informe de cobertura** para el proyecto y familiarízate con el informe obtenido (el valor "n/a" significa "Not applicable"). Puedes abrir directamente el fichero ***index.html*** del informe en el navegador desde el menú contextual, seleccionando "Open in Browser".

Nota: Cuando abrimos un fichero html en el navegador desde IntelliJ, en ocasiones no se visualiza correctamente. Si eso ocurre, ábrelo directamente desde el Gestor de Archivos.

- C) Añade el siguiente caso de prueba al conjunto (`a=7, b=7, c=7, resultado esperado = 7`). Deberías tener claro lo que ocurre al intentar generar de nuevo el informe de cobertura y cómo solucionar el problema.

- D) Recuerda que siempre tienes que ejecutar *mvn clean*, antes de generar un nuevo informe. Cambia el caso de prueba del apartado anterior por ((a=3, b=6, c=2, resultado esperado = 8). Ahora añade el siguiente método a la clase *MultipathExample* y genera de nuevo el informe. Añade los casos de prueba necesarios (utilizando un test parametrizado) para conseguir una cobertura del 100% de condiciones y decisiones y vuelve a generar el informe. Observa las diferencias. Debes tener claro cómo obtiene JaCoCo el valor de CC para el nuevo método que has añadido

```
public int multiPath2(int a, int b, int c )
{
    if ((a > 5) && (b < 5)) {
        b += a;
    }
    if (c > 5) {
        c += b;
    }
    return c;
}
```

- E) Añade el siguiente método a la clase *MultipathExample* y genera de nuevo el informe. Añade los casos de prueba necesarios (utilizando un test parametrizado) para conseguir una cobertura del 100% de condiciones y decisiones y vuelve a generar el informe. Observa las diferencias y justifica el valor de CC para el nuevo método que has añadido

```
public int multiPath3(int a, int b, int c )
{
    if ((a > 5) & (b < 5)) {
        b += a;
    }
    if (c > 5) {
        c += b;
    }
    return c;
}
```

🔗🔗 Ejercicio 2: Informes de cobertura

En el proyecto anterior, vamos a crear un nuevo paquete "ejercicio2", al que deberás añadir las clases de la carpeta/plantillas/ejercicio2. Cada fichero debes añadirlo donde corresponda.

- A) Obtén los informes de cobertura de nuestros tests unitarios y de integración, teniendo en cuenta las nuevas clases (tendrás que modificar convenientemente el pom del proyecto).
- B) Queremos "chequear" que se alcanzan ciertos niveles de cobertura con nuestras pruebas unitarias. Modifica el pom para que se compruebe de forma automática lo siguiente:
- ➡ A nivel de proyecto, queremos conseguir una CC con un valor mínimo del 90%, una cobertura de instrucciones mínima del 80%, y que no haya ninguna clase que no se haya probado.
 - ➡ A nivel de clase, queremos conseguir una cobertura de líneas del 75%
- C) Obtén de nuevo los informes de cobertura, y comprueba que el proceso de construcción falla, porque se incumplen dos de las reglas (<rule>) que hemos impuesto. Con respecto al nivel de proyecto, modifica la regla que no permite completar la construcción, cambiando el valor del contador correspondiente. Con respecto a nivel de clase, no queremos cambiar la regla, de forma que para que la construcción se lleve a cabo con éxito tendrás que excluir una de las clases del paquete ejercicio2 anidando en la <rule> correspondiente el elemento <exclude>, como se muestra a continuación:

```
<rule>
  <element>...</element>
  <excludes>
    <exclude>ejercicio2.NombreDeLaClase</exclude>
  </excludes>
  ...
</rule>
```

⇒ Ejercicio 3: Proyecto Matriculacion

Para este ejercicio usaremos el proyecto multimódulo **matriculacion** de la práctica P06B.

Para ello copia tu solución, es decir, la carpeta *matriculacion* (y todo su contenido) en el directorio de esta práctica (*ppss-2021-Gx-apellido1-apellido2/P10-Cobertura/*).

Puedes borrar el fichero *matriculacion.iml* del proyecto (carpeta "matriculacion"), ya que no nos hará falta. A continuación simplemente abre el proyecto **matriculacion** desde IntelliJ (seleccionando la carpeta que acabas de copiar).

Se pide:

- A) Modifica convenientemente el pom del módulo **matriculacion-dao** para obtener un informe de cobertura para dicho proyecto. Genera el informe a través del correspondiente comando maven.
- B) Observa los valores obtenidos a nivel de proyecto y paquete. Tienes que tener claro cómo se obtienen dichos valores. Fíjate también en los valores de CC obtenidos a nivel de paquete y clase.
- C) El proyecto *matriculacion-dao* también ejecuta las clases de *matriculacion-comun*, sin embargo no aparecen en el informe. Deberías tener claro el por qué no aparecen.
- D) Dado que tenemos un proyecto multimódulo, vamos a usar la goal *jacoco:report-aggregate* para generar un informe para las dependencias de cada módulo, aprovechándonos del mecanismo reactor de maven. (ver <https://www.jacoco.org/jacoco/trunk/doc/report-aggregate-mojo.html>)

Para ello tendrás que comentar el plugin jacoco del proyecto *matriculacion-dao*, e incluirlo en el pom del proyecto *matriculacion*. En lugar de usar las goals *report* y *report-integration*, debes usar la goal *report-aggregate*. Si observas la documentación del enlace, verás que dicha goal no está asociada por defecto a ninguna fase de maven. Debes tener claro qué ocurrirá si no la asociamos a ninguna fase, así como a qué fase deberemos asociarla.

- E) Ejecuta el comando maven correspondiente para obtener el informe agregado de cobertura para el proyecto multimódulo. Averigua dónde se genera dicho informe de cobertura. Observa las diferencias con el informe que hemos obtenido anteriormente.

Resumen



¿Qué **conceptos** y **cuestiones** me deben quedar CLAROS después de hacer la práctica?



NIVELES DE COBERTURA

- La cobertura es una métrica que mide la extensión de nuestras pruebas. Existen diferentes variantes de esta métrica, que se pueden clasificar por niveles, de menos a más cobertura. Es importante entender cada uno de los niveles.
- El cálculo de esta métrica forma parte del análisis de pruebas, que se realiza después de su ejecución.

HERRAMIENTA JaCoCo

- JaCoCo es una herramienta que permite analizar la cobertura de nuestras pruebas, calculando los valores de varios "contadores" (JaCoCo counters), como son: líneas de código, instrucciones, complejidad ciclomática, módulos, clases,... También se pueden calcular los valores a nivel de proyecto, paquete, clases y métodos.
- JaCoCo puede usarse integrado con maven a través del plugin correspondiente. Es posible realizar una instrumentación de las clases on-the-fly, o de forma off-line. En nuestro caso usaremos la primera de las opciones.
- JaCoCo genera informes de cobertura tanto para los tests unitarios como para los tests de integración. Y en cualquier caso, se pueden establecer diferentes "reglas" para establecer diferentes niveles de cobertura dependiendo de los valores de los contadores, de forma que si no se cumplen las restricciones especificadas, el proceso de construcción no terminará con éxito.
- De igual forma, para proyectos multimódulo, se pueden generar informes "agregados", de forma que se tengan en cuenta todos y cada uno de los módulos del proyecto y sus dependencias entre ellos.