

Sumario

GIT.....	2
Que es GIT.....	2
Descargar GIT.....	2
Crear un repositorio nuevo.....	2
Hacer checkout a un repositorio.....	2
Flujo de trabajo.....	3
add & commit.....	3
Envío de cambios.....	3
Ramas.....	4
Actualiza & fusiona.....	5
Etiquetas.....	5
Reemplaza cambios locales.....	5
Datos útiles.....	5
GITHUB.....	6
¿Qué es GitHub?.....	6
Aprendiendo a usar GitHub.....	6
Crear una cuenta.....	6
Manejo de repositorios.....	8
¿Cómo crear un repositorio?.....	8
Crear un proyecto.....	9
Subir proyecto.....	10
Colaborar en un proyecto ajeno.....	10
Maven.....	11
Introducción.....	11
Instalación de Maven.....	11
Creando proyectos y los arquetipos.....	12
El fichero pom.xml.....	12
Como compilar, empaquetar,	13
El repositorio de Maven.....	14
Creando el proyecto Web.....	15
Definiendo dependencias.....	16
Pruebas Unitarias JUNIT.....	16
Introducción a JUnit.....	16
Un ejemplo sencillo.....	17
Implementación de los casos de prueba.....	18
Pruebas con lanzamiento de excepciones.....	20
Ejecución de pruebas.....	21
TDD Test Driven Development (Desarrollo guiado por pruebas).....	22
Definición.....	22
Fixtures.....	23
Objetos mock.....	25

GIT

Que es GIT

Git es uno de los sistemas de control de versiones más populares y del que más estamos oyendo hablar últimamente. Gracias a su potencia y versatilidad muchos grandes proyectos de software libre están migrando sus **repositorios a Git**. Cada vez más es más importante saber usar Git, tanto a nivel personal como laboral. Parte del éxito de este sistema de repositorios es GitHub. Los repositorios en GIT son distribuidos.



Descargar GIT

<https://git-scm.com/>

Crear un repositorio nuevo

Crea un directorio nuevo, ábrelo y ejecuta

```
git init
```

para crear un nuevo repositorio de git.

Hacer checkout a un repositorio

Crea una copia local del repositorio ejecutando

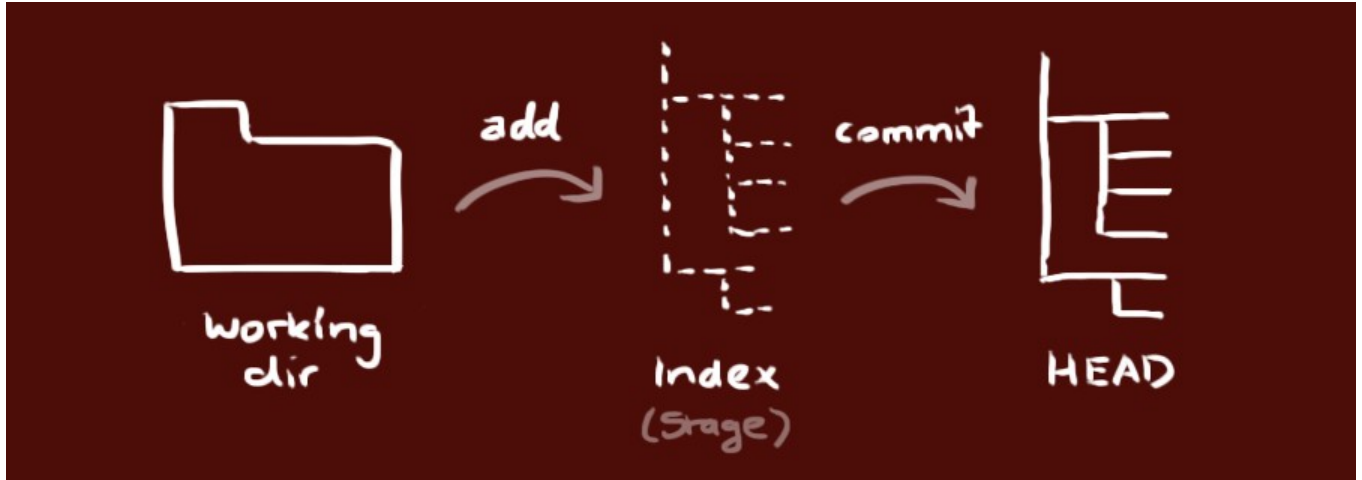
```
git clone /path/to/repository
```

Si utilizas un servidor remoto, ejecuta

```
git clone username@host:/path/to/repository
```

Flujo de trabajo

Tu repositorio local esta compuesto por tres "árboles" administrados por git. El primero es tu **Directorio de trabajo** que contiene los archivos, el segundo es el **Index** que actua como una zona intermedia, y el último es el **HEAD** que apunta al último commit realizado.



add & commit

Puedes registrar cambios (añadirlos al **Index**) usando

```
git add <filename>
git add .
```

Este es el primer paso en el flujo de trabajo básico. Para hacer commit a estos cambios usa

```
git commit -m "Commit message"
```

Ahora el archivo esta incluído en el **HEAD**, pero aún no en tu repositorio remoto.

Envío de cambios

Tus cambios están ahora en el **HEAD** de tu copia local. Para enviar estos cambios a tu repositorio remoto ejecuta

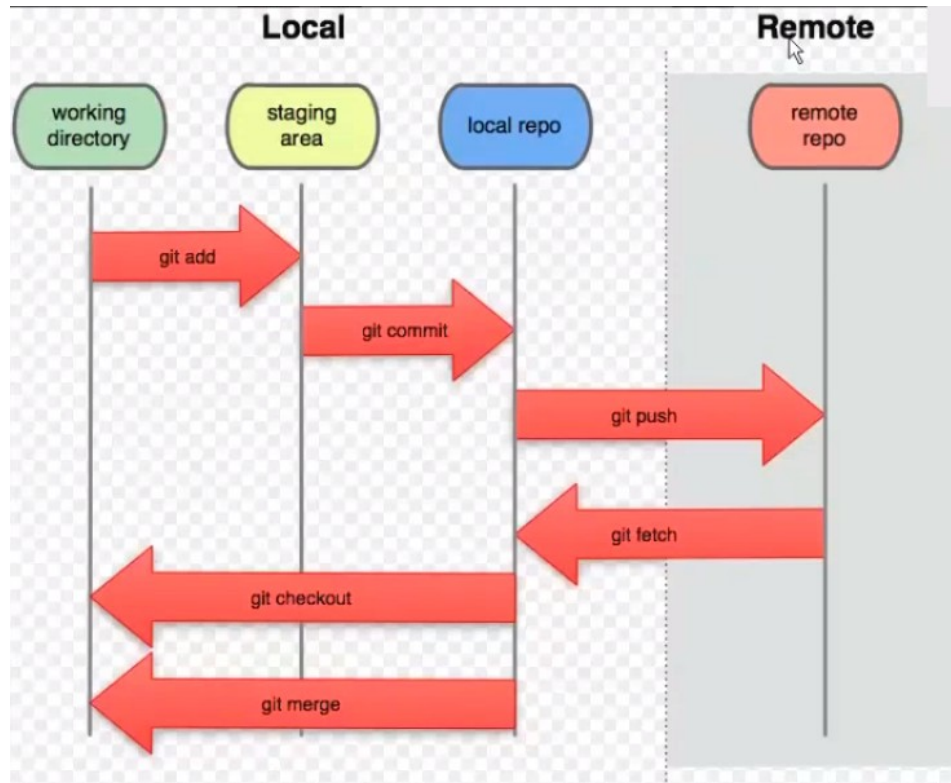
```
git push origin master
```

Reemplaza *master* por la rama a la que quieres enviar tus cambios.

Si no has clonado un repositorio ya existente y quieres conectar tu repositorio local a un repositorio remoto, usa

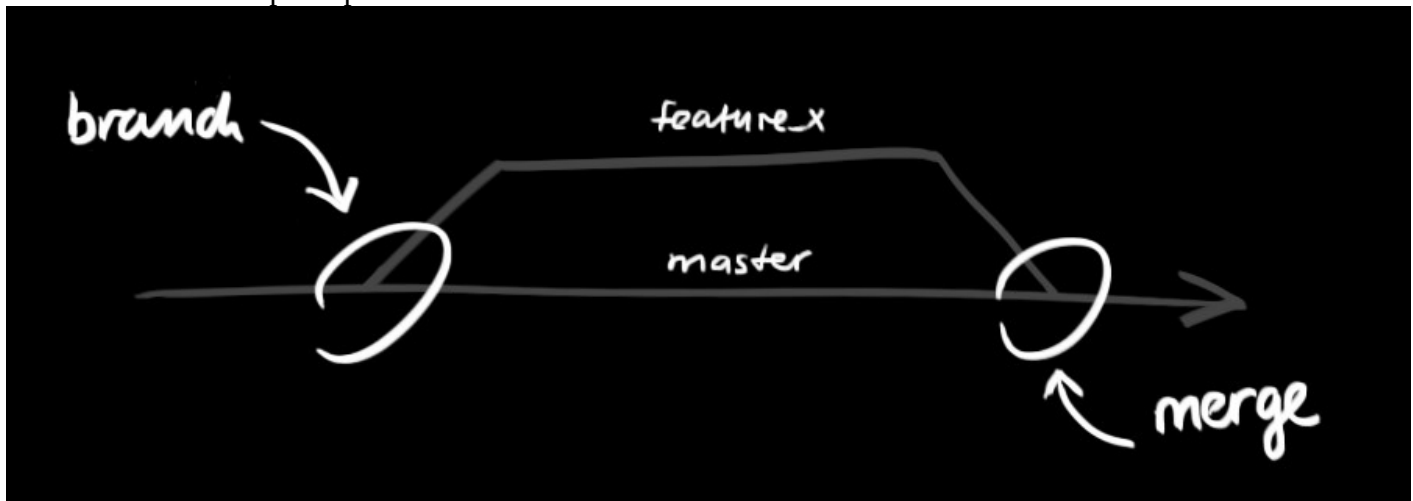
```
git remote add origin <server>
```

Ahora podrás subir tus cambios al repositorio remoto seleccionado.



Ramas

Las ramas son utilizadas para desarrollar funcionalidades aisladas unas de otras. La rama *master* es la rama "por defecto" cuando creas un repositorio. Crea nuevas ramas durante el desarrollo y fusiionalas a la rama principal cuando termines.



Crea una nueva rama llamada "feature_x" y cámbiate a ella usando

```
git checkout -b feature_x
```

vuelve a la rama principal

```
git checkout master
```

y borra la rama

```
git branch -d feature_x
```

Una rama nueva *no estará disponible para los demás* a menos que subas (push) la rama a tu

repositorio remoto

```
git push origin <branch>
```

Actualiza & fusiona

Para actualizar tu repositorio local al commit más nuevo, ejecuta

```
git pull
```

en tu directorio de trabajo para *bajar* y *fusionar* los cambios remotos.

Para fusionar otra rama a tu rama activa (por ejemplo master), utiliza

```
git merge <branch>
```

en ambos casos git intentará fusionar automáticamente los cambios. Desafortunadamente, no siempre será posible y se podrán producir *conflictos*. Tú eres responsable de fusionar esos *conflictos* manualmente al editar los archivos mostrados por git. Después de modificarlos, necesitas marcarlos como fusionados con

```
git add <filename>
```

Antes de fusionar los cambios, puedes revisarlos usando

```
git diff <source_branch> <target_branch>
```

Etiquetas

Se recomienda crear etiquetas para cada nueva versión publicada de un software. Este concepto no es nuevo, ya que estaba disponible en SVN. Puedes crear una nueva etiqueta llamada *1.0.0* ejecutando

```
git tag 1.0.0 1b2e1d63ff
```

1b2e1d63ff se refiere a los 10 caracteres del commit id al cual quieres referirte con tu etiqueta.

Puedes obtener el commit id con

```
git log
```

también puedes usar menos caracteres que el commit id, pero debe ser un valor único.

Reemplaza cambios locales

En caso de que hagas algo mal (lo que seguramente nunca suceda ;) puedes reemplazar cambios locales usando el comando

```
git checkout -- <filename>
```

Este comando reemplaza los cambios en tu directorio de trabajo con el último contenido de HEAD.

Los cambios que ya han sido agregados al Index, así como también los nuevos archivos, se mantendrán sin cambio.

Por otro lado, si quieres deshacer todos los cambios locales y commits, puedes traer la última versión del servidor y apuntar a tu copia local principal de esta forma

```
git fetch origin
```

```
git reset --hard origin/master
```

Datos útiles

Interfaz gráfica por defecto

```
gitk
```

Colores especiales para la consola

```
git config color.ui true
```

Mostrar sólo una línea por cada commit en la traza

```
git config format.pretty oneline
```

Agregar archivos de forma interactiva
`git add -i`

GITHUB

¿Qué es GitHub?

GitHub es una plataforma de **desarrollo colaborativo de software** para alojar proyectos utilizando el sistema de control de versiones [Git](#).

GitHub aloja tu repositorio de código y te brinda **herramientas** muy útiles para el **trabajo en equipo**, dentro de un proyecto.

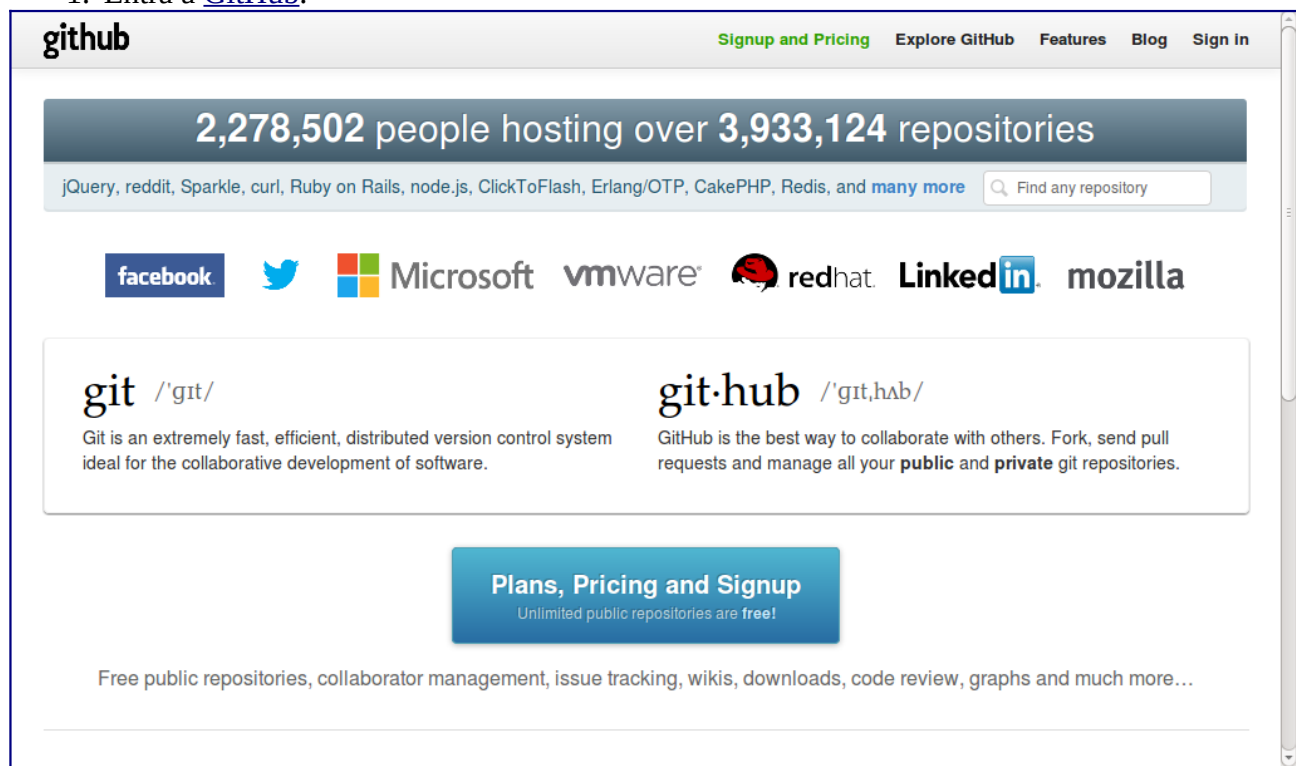
Además de eso, puedes **contribuir a mejorar el software de los demás**. Para poder alcanzar esta meta, GitHub provee de funcionalidades para hacer un **fork** y solicitar **pulls**.

Aprendiendo a usar GitHub

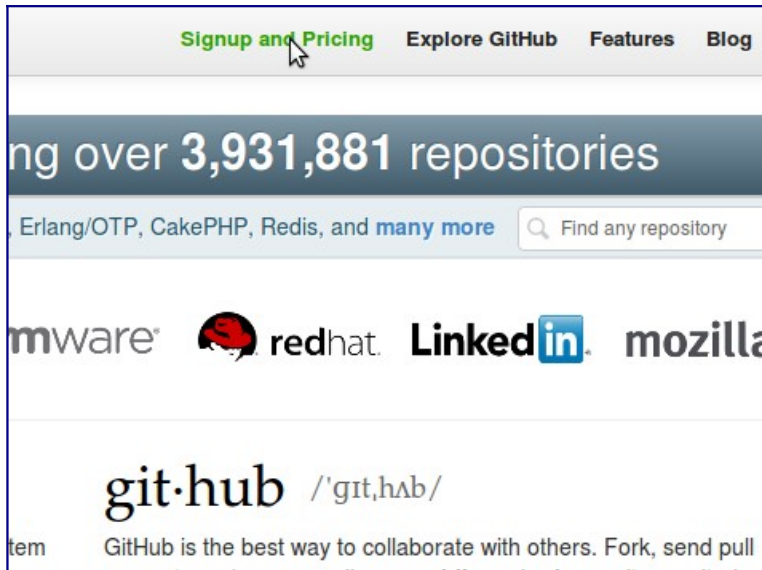
Crear una cuenta

Para crear una cuenta GitHub vamos a seguir los siguientes pasos:

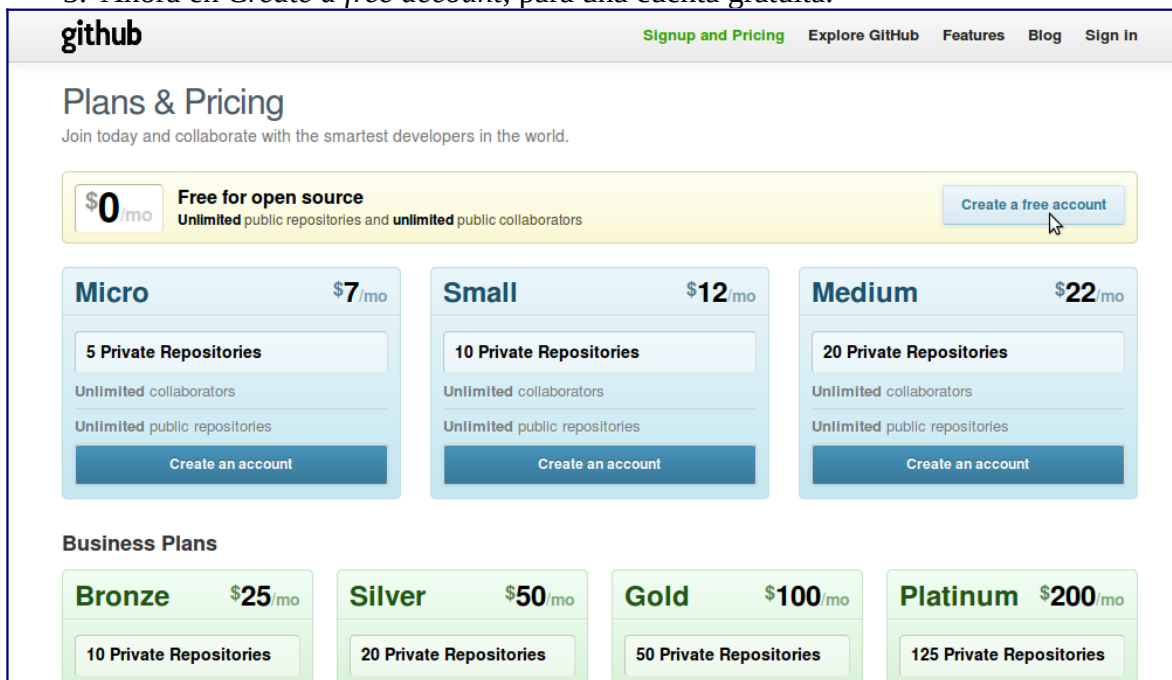
1. Entrá a [GitHub](#).



2. Hací clic en *Singnup and Pricing*, de la barra de herramientas de la página.



3. Ahora en *Create a free account*, para una cuenta gratuita.



4. Finalmente, solo habrá que llenar un pequeño formulario con los siguientes datos:

- Nombre de usuario.
- Dirección email.
- Contraseña.
- Confirmar contraseña.

Create your free personal account

Username

Email Address

We promise we won't share your email with anyone.

Password

Must contain one lowercase letter, one number, and be at least 7 characters long.

Confirm Password

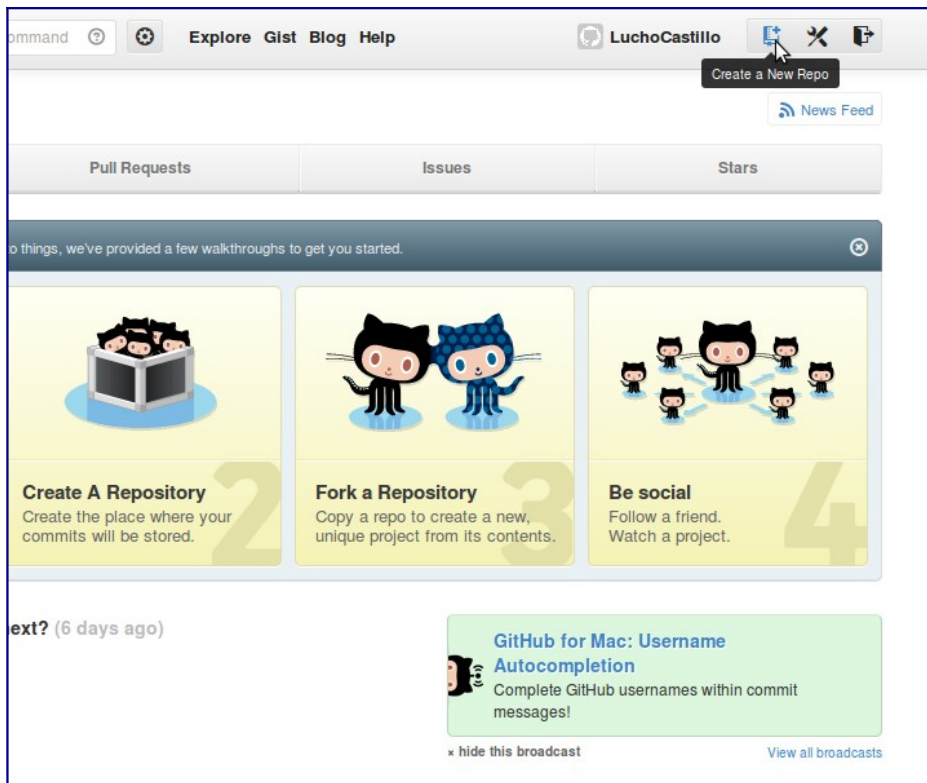
By clicking on "Create an account" below, you are agreeing to the [Terms of Service](#) and the [Privacy Policy](#).

Create an account

Manejo de repositorios

¿Cómo crear un repositorio?



Para crear un repositorio en GitHub, solo hay que seleccionar el botón “*Create a New Repo*”, de la barra de herramientas, habiendo entrado a [GitHub](#) con tu cuenta:



Ahora habrá que llenar dos datos:

1. Nombre del repositorio
2. Descripción del repositorio (opcional)


Owner Repository name


 **LuchoCastillo** Repository 

Great repository names are short and memorable. Need inspiration? How about **furry-octo-nemesis**.

Description (optional)

Soy un repositorio :)

☒ **Public** 
Anyone can see this repository. You choose who can commit.

☐ **Private** 
You choose who can see and commit to this repository.

☐ **Initialize this repository with a README**
This will allow you to `git clone` the repository immediately.

Add .gitignore: **None**

Create repository

¡Listo! Repositorio creado, ahora lo vas a poder ver en tu perfil.

Crear un proyecto

Al crear el repositorio, inmediatamente nos va a llevar a él. Como nuestro proyecto no tiene nada en su interior, no nos va a mostrar más que una ayuda para subir archivos y proyectos.

Para crear un proyecto desde cero, habrá que comenzar creando los archivos del mismo y luego subiéndolos a la página.

En el primer recuadro de la ayuda, verás una serie de comandos para el terminal.

```
touch README.md
git init
git add README.md
git commit -m "comentario"
git remote add origin https://github.com/LuchoCastillo/Repositorio.git
git push -u origin master
```

Antes de seguirlos, tendrás que instalar git:
sudo apt-get install git

Subir proyecto

Para subir un proyecto ya realizado a GitHub, habrá que seguir exactamente los mismos pasos, ya que para iniciar un proyecto nuevo, habrá que subir los archivos creados y luego modificarlos en el editor de la página.

En este caso, los archivos junto con su contenido, ya están hechos. Solo hay que subirlos y editarlos si surge algún inconveniente.

Para subir un archivo hay que ubicarse en la carpeta del repositorio y seguir estos pasos:

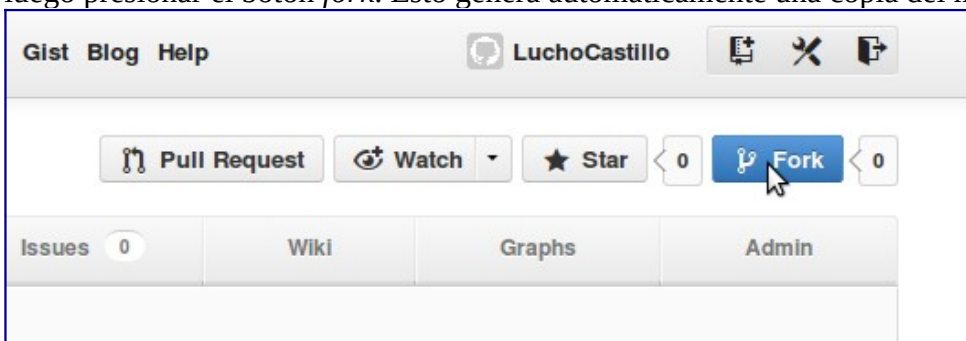
```
git add archivo
git commit -m "comentario"
git push
```

Es importante realizar los 3, ya que si no se ingresa un comentario, no se realiza el cambio.

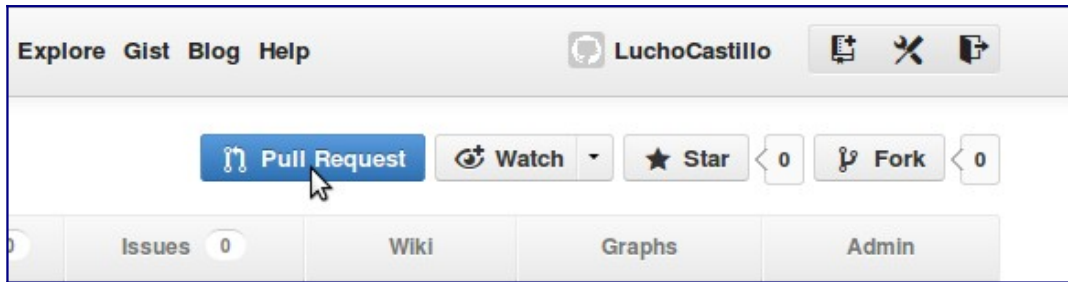
Para adquirir un poco más de práctica y conocer más comandos de `git`, pueden ingresar a este [tutorial](#) bastante práctico.

Colaborar en un proyecto ajeno

Para colaborar en un proyecto ajeno simplemente basta con buscarlo dentro de los repositorios, y luego presionar el botón *fork*. Esto genera automáticamente una copia del mismo en tu perfil.



Al terminar tus modificaciones podrás presionar *Pull Request* para enviárselo al creador del mismo.



Maven

Introducción

Maven (<http://maven.apache.org>) es una herramienta para la gestión de proyectos de software, que se basa en el concepto de POM (Project Object Model). Es decir, con Maven vamos a poder compilar, empaquetar, generar documentación, pasar los test, preparar las builds, ... Ahora muchos pensaréis que eso ya los sabemos hacer con Ant (<http://ant.apache.org>), pero no debemos confundirnos, ya que Maven y Ant son cosas totalmente diferentes. Algunas de las principales ventajas de Maven frente a Ant, podrían ser:

- Maven se basa en patrones y en estándares. Esto permite a los desarrolladores moverse entre proyectos y no necesitan aprender como compilar o empaquetar. Esto mejora el mantenimiento y la reusabilidad.
- Mientras que con Ant escribimos una serie de tareas, y unas dependencias entre estas tareas; con el POM hacemos una descripción del proyecto. Es decir, decimos de que se compone nuestro proyecto (nombre, versión, librerías de las que depende, ...), y Maven se encargará de hacer todas las tareas por nosotros.
- Maven hace la gestión de librerías, incluso teniendo en cuenta las dependencias transitivas. Es decir, si A depende de B y B depende de C, es que A depende de C. Esto quiere decir que cuando empaquetemos A, Maven se encargará de añadir tanto B como C en el paquete.

En cualquier caso no tienen por que ser herramientas enfrentadas, sino que pueden ser complementarias. Usando cada una según nos interese.

Instalación de Maven

Debemos descargarnos el paquete de Maven (por ejemplo maven-2.0.4-bin.tar.bz2) de <http://maven.apache.org/download.html#installation> (es recomendable elegir un mirror para la descarga).

Una vez descargado, tenemos que descomprimirlo en nuestro ordenador. Por ejemplo, suponiendo que nos hemos descargado el paquete de Maven en /download, podemos hacer:

```
$ cd /opt $ tar -xjf /download/maven-2.0.4-bin.tar.bz2
$ cd /opt
$ tar -xjf /download/maven-2.0.4-bin.tar.bz2
```

Esto nos creará el directorio /opt/maven-2.0.4.

Ahora basta con añadir el directorio /opt/maven-2.0.4/bin al PATH del sistema:

```
export PATH=$PATH:/opt/maven-2.0.4/bin
```

Para comprobar que está correctamente instalado podemos probar a ejecutar:
\$ mvn --version

Si tenemos algún problema podemos comprobar si la variable JAVA_HOME apunta correctamente a la localización donde está instalada nuestra JDK.

Creando proyectos y los arquetipos

Podríamos decir que un “arquetipo” para Maven es una plantilla. Es decir, gracias a un arquetipo Maven es capaz de generar una estructura de directorios y ficheros.

Con los arquetipos se acaba “el miedo al folio en blanco” a la hora de empezar un proyecto, ya que basta con decirle a Maven que tipo de proyecto queremos y nos creará la estructura base.

Por ejemplo, para crear nuestro proyecto java basta con hacer:

```
$ mvn archetype:create -DgroupId=com.autentia.demoapp
-DartifactId=autentiaNegocio
```

Donde groupId es el identificador único de la organización o grupo que crea el proyecto (se podría decir que es el identificador de la aplicación), y artifactId es el identificador único del artefacto principal de este proyecto (se podría decir que es el identificador del módulo dentro de la aplicación), es decir, este será el nombre del jar.

En el comando no ha hecho falta decir que queremos construir un proyecto java, ya que esta es la opción por defecto.

El resultado de ejecutar este comando es la siguiente estructura de directorios y ficheros:

```
autentiaNegocio
|-- pom.xml
`-- src
    |-- main
    |   |-- java
    |   |   |-- com
    |   |   |   |-- autentia
    |   |   |   |   |-- demoapp
    |   |   |   |       |-- App.java
    |-- test
    |   |-- java
    |   |   |-- com
    |   |   |   |-- autentia
    |   |   |   |   |-- demoapp
    |   |   |   |       |-- AppTest.java
```

Esta estructura de directorios es estándar, siendo la misma en todos los proyectos. Maven nos permite cambiar esta estructura, pero no sería recomendable ya que el hecho de que sea estándar permite a los desarrolladores moverse entre proyectos con mayor comodidad, ya que siempre sabrán donde encontrar las cosas. Para más información sobre la estructura de directorios se puede consultar <http://maven.apache.org/guides/introduction/introduction-to-the-standard-directory-layout.html>

El fichero pom.xml

El fichero pom.xml es donde vamos a describir nuestro proyecto. Vamos a echar un vistazo al fichero pom.xml que nos ha generado Maven:

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.autentia.demoapp</groupId>
  <artifactId>autentiaNegocio</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>Maven Quick Start Archetype</name>
  <url>http://maven.apache.org/</url>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

Con packaging se indica el tipo de empaquetado que hay que hacer con el proyecto. Podemos usar jar, war, ear, pom.

Con version se indica la versión del proyecto con la que estamos trabajando. Al indicar SNAPSHOT se quiere decir que es una versión evolutiva, es decir que estamos trabajando para obtener al versión 1.0.

También podemos ver como dentro de dependencies se describen las dependencias del proyecto. Ahora tenemos una dependencia de junit para poder compilar y ejecutar los test. Dentro de la descripción de las dependencias es interesante destacar el elemento scope que indica que tipo de librería se trata. Podemos distinguir:

- **compile** – es el valor por defecto. Se utiliza en todos los casos (compilar, ejecutar, ...).
- **provided** – también se utiliza en todos los casos, pero se espera que el jar sea suministrado por la JDK o el contenedor. Es decir, no se incluirá al empaquetar el proyecto, ni en el repositorio.
- **runtime** – no se utiliza para compilar, pero si es necesario para ejecutar.
- **test** – Sólo se utiliza para compilar o ejecutar los test.
- **system** – es similar a provided, pero eres tu el que tiene que suministrar el jar. No se incluirá al empaquetar el proyecto, ni en el repositorio.

Para saber más se puede consultar <http://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html>

Como compilar, empaquetar, ...

Ahora mismo, sin haber escrito ni una línea ya podemos hacer todas las tareas habituales (esto con Ant no sería tan fácil);):

- \$ mvn compile – compila el proyecto y deja el resultado en target/classes
- \$ mvn test – compila los test y los ejecuta
- \$ mvn package – empaqueta el proyecto y lo dejará en target/autentiaNegocio-1.0-SNAPSHOT.jar
- \$ mvn install – guarda el proyecto en el repositorio
- \$ mvn clean – borra el directorio de salida (target)
- ...

Estas tareas son estándar, por lo que un desarrollador puede saltar de un proyecto a otro y siempre sabrá compilar, empaquetar, ...

Maven define un ciclo de vida por lo que al ejecutar un objetivo, antes se ejecutan los sus antecesores en el ciclo de vida. Por ejemplo, si ejecutamos package, antes se ejecutará compile y test.

Para saber más sobre el ciclo de vida de Maven se puede leer

<http://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html>

El repositorio de Maven

Maven guarda todas las dependencias y proyectos en un único repositorio.

Este repositorio está situado en <USER_HOME>/m2/repository (aunque Maven nos permite cambiar esta localización por defecto).

<https://mvnrepository.com/>

The screenshot shows the Maven Repository website. The browser address bar displays <https://mvnrepository.com/artifact/mysql/mysql-connector-java>. The page title is "MySQL Connector/J" and the description is "JDBC Type 4 driver for MySQL". The page shows the artifact is used by 2,526 artifacts. A table lists the versions available in the Central repository, with 67 versions in total. The table includes columns for Version, Repository, Usages, and Date.

Version	Repository	Usages	Date
8.0.11	Central	17	Apr, 2018
8.0.9-rc	Central	0	Jan, 2018
8.0.8-dmr	Central	12	Sep, 2017
8.0.7-dmr	Central	6	Jun, 2017
6.0.6	Central	127	Feb, 2017
6.0.5	Central	51	Oct, 2016
6.0.4	Central	17	Aug, 2016
6.0.3	Central	21	Jun, 2016
6.0.2	Central	12	Mar, 2016
5.1.46	Central	39	Feb, 2018

También existe un repositorio remoto desde el cual se descargan los diferentes jars según los vamos necesitando. Esto ya lo habréis notado al ejecutar los primeros comando de Maven; seguro que habéis visto como Maven se ponía a descargar varias cosas (ojo, si salís a Internet a través de un proxy, lo tendréis que configurar en Maven <http://maven.apache.org/guides/mini/guide-proxies.html>).

Este repositorio remoto también podemos cambiarlo, e incluso podemos tener más de uno. Sería muy interesante que dentro de nuestra organización tuviéramos un repositorio remoto donde publicar todos nuestros proyectos.

Para saber más puedes leer <http://maven.apache.org/guides/introduction/introduction-to-repositories.html>

En general esta idea es bastante buena porque no hace falta que tengamos los jar duplicados en el sistema de control de versiones, dentro de cada proyecto. Sobre todo con los jar de terceros, ya que ocupan espacio y es algo que no tiene utilidad que versionemos (lo interesante es saber en cada versión de nuestro proyecto que dependencias tenía pero no versionar la dependencia en sí). Además el repositorio de Maven guardar las diferentes versiones de cada proyecto o dependencia, con lo cual podemos actualizar una dependencia de un proyecto a una versión superior, sin que afecte al resto de nuestros proyectos (que seguirán usando la versión anterior).

Creando el proyecto Web

Basándonos en los arquetipos, basta con ejecutar:

```
$ mvn archetype:create -DgroupId=com.autentia.demoapp -DartifactId=autentiaWeb
-DarchetypeArtifactId=maven-archetype-webapp
```

Como antes, indicamos el groupId y el artifactId, pero esta vez también indicamos el archetypeArtifactId. En este último atributo, con el valor maven-archetype-webapp, estamos indicando que queremos usar la plantilla de aplicaciones web.

A parte de los arquetipos que nos proporciona Maven, es interesante saber que podemos crear nuestros propios arquetipos: <http://maven.apache.org/guides/mini/guide-creating-archetypes.html>
Echemos un vistazo al pom.xml generado:

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.autentia.demoapp</groupId>
  <artifactId>autentiaWeb</artifactId>
  <packaging>war</packaging>
  <version>1.0-SNAPSHOT</version>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
  <build>
    <finalName>autentiaWeb</finalName>
  </build>
</project>
```

Destacaremos como en este caso el elemento packaging es igual a war, y la aparición del elemento finalName.

Con finalName estamos indicando el nombre del archivo que se genera al empaquetar, en este caso el nombre del archivo war. En el proyecto anterior no usamos este elemento porque queríamos el

comportamiento por defecto: el nombre del archivo se genera como el artifactId + version. Pero este comportamiento no nos suele interesar en un archivo war, porque sino al desplegar la aplicación habría que poner la versión de la aplicación en la URL para acceder a ella.

Definiendo dependencias

Hasta ahora no hemos escrito ni una sola línea, bueno ya es hora de ponerse a trabajar y hacer algo 😊

Vamos a indicar que el proyecto autentiaNegocio depende del driver de MySQL. Para ello basta con editar el fichero autentiaNegocio/pom.xml y añadir:

```
...
<dependencies>
...
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>5.0.3</version>
        <scope>runtime</scope>
    </dependency>
</dependencies>
...
</project>
```

Nótese como se indica scope runtime, es decir este jar sólo se tendrá en cuenta a la hora de ejecutar, pero no se usará para compilar.

Pruebas Unitarias JUNIT

Introducción a JUnit

Cuando probamos un programa, lo ejecutamos con unos datos de entrada (casos de prueba) para verificar que el funcionamiento cumple los requisitos esperados. Definimos **prueba unitaria** como la prueba de uno de los módulos que componen un programa.

En los últimos años se han desarrollado un conjunto de herramientas que facilitan la elaboración de pruebas unitarias en diferentes lenguajes. Dicho conjunto se denomina *XUnit*. De entre dicho conjunto, **JUnit** es la herramienta utilizada para realizar pruebas unitarias en Java.

El concepto fundamental en estas herramientas es el **caso de prueba** (*test case*), y la **suite** de prueba (*test suite*). Los casos de prueba son clases o módulos que disponen de métodos para probar los métodos de una clase o módulo concreta/o. Así, para cada clase que quisiéramos probar definiríamos su correspondiente clase de caso de prueba. Mediante las suites podemos organizar los casos de prueba, de forma que cada suite agrupa los casos de prueba de módulos que están funcionalmente relacionados.

Las pruebas que se van construyendo se estructuran así en forma de árbol, de modo que las hojas son los casos de prueba, y podemos ejecutar cualquier subárbol (suite).

De esta forma, construimos programas que sirven para probar nuestros módulos, y que podremos ejecutar de forma automática. A medida que la aplicación vaya avanzando, se dispondrá de un conjunto importante de casos de prueba, que servirá para hacer pruebas de regresión. Eso es importante, puesto que cuando cambiamos un módulo que ya ha sido probado, el cambio puede haber afectado a otros módulos, y sería necesario volver a ejecutar las pruebas para verificar que todo sigue funcionando.

Aplicando lo anterior a Java, JUnit es un conjunto de clases opensource que nos permiten probar nuestras aplicaciones Java. Podemos encontrar información actualizada de JUnit en

<http://www.junit.org>

Encontraremos una distribución de JUnit, en la que habrá un fichero JAR, **junit.jar**, que contendrá las clases que deberemos tener en el CLASSPATH a la hora de implementar y ejecutar los casos de prueba.

Un ejemplo sencillo

Supongamos que tenemos una clase EmpleadoBR con las reglas de negocio aplicables a los empleados de una tienda. En esta clase encontramos los siguientes métodos con sus respectivas especificaciones:

Método	Especificación
float calculaSalarioBruto(TipoEmpleado tipo, float ventasMes, float horasExtra)	El salario base será 1000 euros si el empleado es de tipo TipoEmpleado.vendedor, y de 1500 euros si es de tipo TipoVendedor.encargado. A esta cantidad se le sumará una prima de 100 euros si ventasMes es mayor o igual que 1000 euros, y de 200 euros si fuese al menos de 1500 euros. Por último, cada hora extra se pagará a 20 euros. Si tipo es null, o ventasMes o horasExtra toman valores negativos el método lanzará una excepción de tipo BRException.
float calculaSalarioNeto(float salarioBruto)	Si el salario bruto es menor de 1000 euros, no se aplicará ninguna retención. Para salarios a partir de 1000 euros, y menores de 1500 euros se les aplicará un 16%, y a los salarios a partir de 1500 euros se les aplicará un 18%. El método nos devolverá salarioBruto * (1-retencion), o BRException si el salario es menor que cero.

A partir de dichas especificaciones podemos diseñar un conjunto de casos de prueba siguiendo métodos como el método de pruebas de particiones, también conocido como caja negra. Si en lugar de contar con la especificación, contásemos con el código del método a probar también podríamos diseñar a partir de él un conjunto de casos de prueba utilizando otro tipo de métodos (en este caso se podría utilizar el método de caja blanca). No vamos a entrar en el estudio de estos métodos de prueba, sino que nos centraremos en el estudio de la herramienta JUnit. Por lo tanto, supondremos que después de aplicar un método de pruebas hemos obtenido los siguientes casos de prueba:

Método a probar	Entrada	Salida esperada
calculaSalarioNeto	2000	1640
calculaSalarioNeto	1500	1230
calculaSalarioNeto	1499.99	1259.9916
calculaSalarioNeto	1250	1050

calculaSalarioNeto	1000	840
calculaSalarioNeto	999.99	999.99
calculaSalarioNeto	500	500
calculaSalarioNeto	0	0
calculaSalarioNeto	-1	BRException
calculaSalarioBruto	vendedor, 2000 euros, 8h	1360
calculaSalarioBruto	vendedor, 1500 euros, 3h	1260
calculaSalarioBruto	vendedor, 1499.99 euros, 0h	1100
calculaSalarioBruto	encargado, 1250 euros, 8h	1760
calculaSalarioBruto	encargado, 1000 euros, 0h	1600
calculaSalarioBruto	encargado, 999.99 euros, 3h	1560
calculaSalarioBruto	encargado, 500 euros, 0h	1500
calculaSalarioBruto	encargado, 0 euros, 8h	1660
calculaSalarioBruto	vendedor, -1 euros, 8h	BRException
calculaSalarioBruto	vendedor, 1500 euros, -1h	BRException
calculaSalarioBruto	null, 1500 euros, 8h	BRException

Nota:

Los casos de prueba se pueden diseñar e implementar antes de haber implementado el método a probar. De hecho, es recomendable hacerlo así, ya que de esta forma las pruebas comprobarán si el método implementado se ajusta a las especificaciones que se dieron en un principio. Evidentemente, esto no se podrá hacer en las pruebas que diseñemos siguiendo el método de caja blanca. También resulta conveniente que sean personas distintas las que se encargan de las pruebas y de la implementación del método, por el mismo motivo comentado anteriormente. A continuación veremos como implementar estas pruebas utilizando JUnit 4.

Implementación de los casos de prueba

Vamos a utilizar JUnit para probar los métodos anteriores. Para ello deberemos crear una serie de clases en las que implementaremos las pruebas diseñadas. Esta implementación consistirá básicamente en invocar el método que está siendo probado pasándole los parámetros de entrada establecidos para cada caso de prueba, y comprobar si la salida real coincide con la salida esperada. Esto en principio lo podríamos hacer sin necesidad de utilizar JUnit, pero el utilizar esta herramienta nos va a ser de gran utilidad ya que nos proporciona un *framework* que nos obligará a implementar las pruebas en un formato estándar que podrá ser reutilizable y entendible por cualquiera que conozca la librería. El aplicar este *framework* también nos ayudará a tener una batería de pruebas ordenada, que pueda ser ejecutada fácilmente y que nos muestre los resultados de forma clara mediante una interfaz gráfica que proporciona la herramienta. Esto nos ayudará a realizar pruebas de regresión, es decir, ejecutar la misma batería de pruebas en varios momentos del desarrollo, para así asegurarnos de que lo que nos había funcionado antes siga funcionando bien. Para implementar las pruebas en JUnit utilizaremos dos elementos básicos:

- Por un lado, marcaremos con la anotación `@Test` los métodos que queramos que JUnit ejecute. Estos serán los métodos en los que implementemos nuestras pruebas. En estos métodos llamaremos al método probado y comprobaremos si el resultado obtenido es igual al esperado.
- Para comprobar si el resultado obtenido coincide con el esperado utilizaremos los métodos `assert` de la librería JUnit. Estos son una serie de métodos estáticos de la clase `Assert` (para simplificar el código podríamos hacer un `import` estático de dicha clase), todos ellos con el prefijo `assert-`. Existen multitud de variantes de estos métodos, según el tipo de datos que estemos comprobando (`assertTrue`, `assertFalse`, `assertEquals`, `assertNull`, etc). Las llamadas a estos métodos servirán para que JUnit sepa qué pruebas han tenido éxito y cuáles no.

Cuando ejecutemos nuestras pruebas con JUnit, se nos mostrará un informe con el número de pruebas exitosas y fallidas, y un detalle desglosado por casos de prueba. Para los casos de prueba que hayan fallado, nos indicará además el valor que se ha obtenido y el que se esperaba.

Además de estos elementos básicos anteriores, a la hora de implementar las pruebas con JUnit deberemos seguir una serie de buenas prácticas que se detallan a continuación:

- La clase de pruebas se llamará igual que la clase a probar, pero con el sufijo `-Test`. Por ejemplo, si queremos probar la clase `MiClase`, la clase de pruebas se llamará `MiClaseTest`.
- La clase de pruebas se ubicará en el mismo paquete en el que estaba la clase probada. Si `MiClase` está en el paquete `es.ua.jtech.lja`, `MiClaseTest` pertenecerá a ese mismo paquete. De esta forma nos aseguramos tener acceso a todos los miembros de tipo protegido y paquete de la clase a probar.
- Mezclar clases reales de la aplicación con clases que sólo nos servirán para realizar las pruebas durante el desarrollo no es nada recomendable, pero no queremos renunciar a poner la clase de pruebas en el mismo paquete que la clase probada. Para solucionar este problema lo que se hará es crear las clases de prueba en un directorio de fuentes diferente. Si los fuentes de la aplicación se encuentran normalmente en un directorio llamado `src`, los fuentes de pruebas irían en un directorio llamado `test`.
- Los métodos de prueba (los que están anotados con `@Test`), tendrán como nombre el mismo nombre que el del método probado, pero con prefijo `test-`. Por ejemplo, para probar `miMetodo` tendríamos un método de prueba llamado `testMiMetodo`.
- Aunque dentro de un método de prueba podemos poner tantos `assert` como queramos, es recomendable crear un método de prueba diferente por cada caso de prueba que tengamos. Por ejemplo, si para `miMetodo` hemos diseñado tres casos de prueba, podríamos tener tres métodos de prueba distintos: `testMiMetodo1`, `testMiMetodo2`, y `testMiMetodo3`. De esta forma, cuando se presenten los resultados de las pruebas podremos ver exactamente qué caso de prueba es el que ha fallado.

```
public class EmpleadoBRTest {
    @Test
    public void testCalculaSalarioBruto1() {
        float resultadoReal = EmpleadoBR.calculaSalarioBruto(
            TipoEmpleado.vendedor, 2000.0f, 8.0f);
        float resultadoEsperado = 1360.0f;
        assertEquals(resultadoEsperado, resultadoReal, 0.01);
    }

    @Test
    public void testCalculaSalarioBruto2() {
        float resultadoReal = EmpleadoBR.calculaSalarioBruto(
            TipoEmpleado.vendedor, 1500.0f, 3.0f);
        float resultadoEsperado = 1260.0f;
        assertEquals(resultadoEsperado, resultadoReal, 0.01);
    }
}
```

```

    }

    @Test
    public void testCalculaSalarioNeto1() {
        float resultadoReal = EmpleadoBR.calculaSalarioNeto(2000.0f);
        float resultadoEsperado = 1640.0f;
        assertEquals(resultadoEsperado, resultadoReal, 0.01);
    }

    @Test
    public void testCalculaSalarioNeto2() {
        float resultadoReal = EmpleadoBR.calculaSalarioNeto(1500.0f);
        float resultadoEsperado = 1230.0f;
        assertEquals(resultadoEsperado, resultadoReal, 0.01);
    }
}

```

En general la construcción de pruebas sigue siempre estos mismos patrones: llamar al método probado y comprobar si la salida real coincide con la esperada utilizando los métodos `assert`.

Pruebas con lanzamiento de excepciones

En algunos casos de prueba, lo que se espera como salida no es que el método nos devuelva un determinado valor, sino que se produzca una excepción. Para comprobar con JUnit que la excepción se ha lanzado podemos optar por dos métodos diferentes. El más sencillo de ellos es indicar la excepción esperada en la anotación `@Test`:

```

@Test(expected=BRException.class)
public void testCalculaSalarioNeto9() {
    EmpleadoBR.calculaSalarioNeto(-1.0f);
}

```

Otra posibilidad es utilizar el método `fail` de JUnit, que nos permite indicar que hay un fallo en la prueba. En este caso lo que haríamos sería llamar al método probado dentro de un bloque `try-catch` que capture la excepción esperada. Si al llamar al método no saltase la excepción, llamaríamos a `fail` para indicar que no se ha comportado como debería según su especificación.

```

@Test
public void testCalculaSalarioNeto9() {
    try {
        EmpleadoBR.calculaSalarioNeto(-1.0f);
        fail("Se esperaba excepcion BRException");
    } catch (BRException e) {}
}

```

Cuando el método que estemos probando pueda lanzar una excepción de tipo *checked* que debamos capturar de forma obligatoria en JUnit, también podemos utilizar `fail` dentro del bloque `catch` para notificar del fallo en caso de que se lance la excepción de forma no esperada:

```

public void testCalculaSalarioBruto1() {
    float resultadoReal;
    try {
        resultadoReal = EmpleadoBR.calculaSalarioBruto(
            TipoEmpleado.vendedor, 2000.0f, 8.0f);
        float resultadoEsperado = 1360.0f;
        assertEquals(resultadoEsperado, resultadoReal, 0.01);
    } catch (BRException e) {

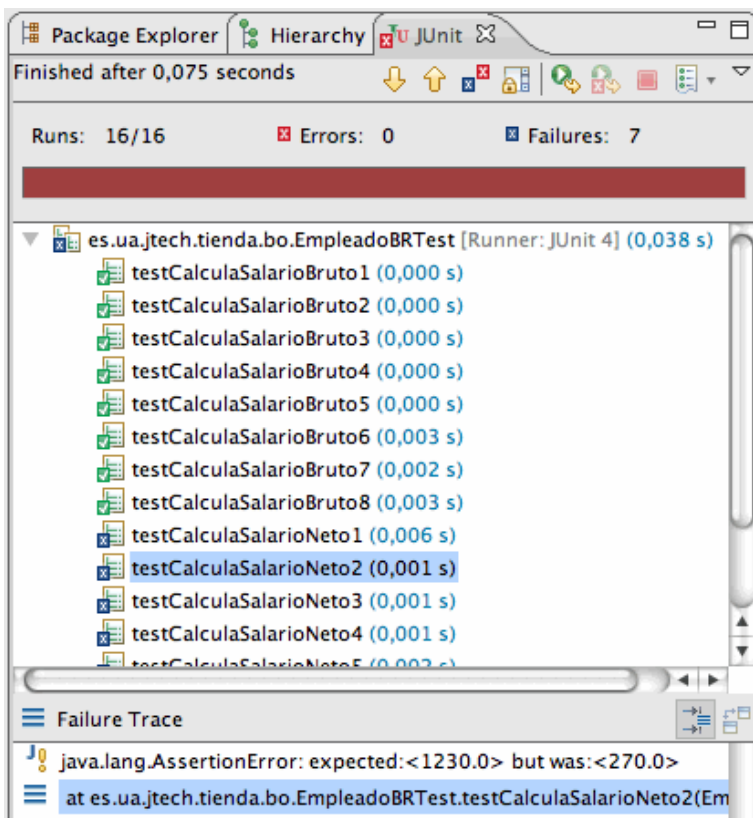
```

```
fail("Lanzada excepcion no esperada BRException");
}
}
```

Si el método probado lanzase una excepción de tipo *unchecked* no esperada, y no la capturásemos, JUnit en lugar de marcarlo como fallo lo marcará como error.

Ejecución de pruebas

Cuando tengamos definida la clase de prueba que queramos ejecutar, y la clase a probar, pulsamos con el botón derecho sobre la clase de prueba y seleccionamos *Run As > JUnit test*. Nos aparecerá la ventana de JUnit en Eclipse con los resultados:



Arriba tendremos una barra roja o verde (según si ha habido fallos o no), y el número de pruebas fallidas, y de errores producidos durante la ejecución de las pruebas. En la parte central vemos la jerarquía de todas las pruebas ejecutadas, con un icono que nos indicará si ha tenido éxito o no, y si seleccionamos alguna de las fallidas abajo vemos los detalles del error (resultado obtenido, resultado esperado, y línea de código en la que se ha producido). Haciendo doble click en los errores iremos a la línea de código que los provocó. Se puede relanzar un test pulsando *Ctrl + F11*, o pulsando sobre el botón "play".

Ejecutar pruebas fuera de Eclipse

Para ejecutar pruebas por sí solas, debemos utilizar un ejecutor de pruebas (*test runner*). JUnit proporciona algunos de ellos, como *junit.textui.TestRunner* (para mostrar los resultados en modo texto), o *junit.swingui.TestRunner* (para mostrar los resultados gráficamente). Para ejecutarlos podemos incluir el jar *junit.jar* en el CLASSPATH al ejecutar:

```
java -cp ./junit.jar junit.swingui.TestRunner
```

Nos aparecería una ventana donde indicamos el nombre del caso de prueba que queremos ejecutar (o lo elegimos de una lista), y luego pulsando *Run* nos mostrará los resultados. La barra verde aparece si las pruebas han ido bien, y si no aparecerá en rojo. En la pestaña *Failures* podemos ver qué pruebas han fallado, y en *Test Hierarchy* podemos ver todas las pruebas que se han realizado, y los resultados para cada una. En el cuadro inferior nos aparecen los errores que se han producido en las pruebas erróneas.

Para ejecutar el *TestRunner* u otro ejecutor de pruebas, podemos también definirnos un método *main* en nuestra clase de prueba que lance el ejecutor, en nuestro caso:

```
public static void main (String[] args){
    String[] nombresTest = {EmpleadoBRTTest.class.getName()};
    junit.swingui.TestRunner.main(nombresTest);
}
```

Vemos que al *main* del *TestRunner* se le pueden pasar como parámetros los nombres de las clases de prueba que queremos probar.

TDD Test Driven Development (Desarrollo guiado por pruebas)

Definición

Anteriormente se ha comentado que puede resultar recomendable implementar las pruebas antes que el método a probar. Existe una metodología de desarrollo denominada desarrollo guiado por pruebas (*Test Driven Development*, *TDD*) que se basa precisamente en implementar primero las pruebas y a partir de ellas implementar las funcionalidades de nuestra aplicación. Esta metodología consiste en los siguientes pasos:

- Seleccionar una funcionalidad de la aplicación a implementar.
- Diseñar e implementar una serie de casos de prueba a partir de la especificación de dicha funcionalidad. Evidentemente, en un principio la prueba ni siquiera compilará, ya que no existirá ni la clase ni el método a probar. Con Eclipse podemos hacer que genere automáticamente los componentes que necesitamos. Podemos ver que al utilizar una clase que no existe en Eclipse nos mostrará un icono de error en la línea correspondiente. Pinchando sobre dicho icono podremos hacer que se cree automáticamente la clase necesaria:

•

```
package es.ua.jtech.tienda.bo;

import static junit.framework.Assert.*;

import org.junit.Test;

public class EmpleadoBOTest {

    @Test
    public void testGetSalarioBruto() {
        EmpleadoBO ebo = new EmpleadoBO();
        float resultadoReal =
        float resultadoEspera

        assertEquals(resultadoReal, resultadoEspera);
    }
}
```

- Una vez compile, se deberá ejecutar la prueba para comprobar que falla. De esta forma nos aseguramos de que realmente las pruebas están comprobando algo.
- El siguiente paso consiste en hacer el mínimo código necesario para que se pasen los tests. Aquí se deberá implementar el código más sencillo posible, y no hacer nada que no sea necesario para pasar las pruebas, aunque nosotros veamos que pudiese ser conveniente. Si creemos que se debería añadir algo, la forma de proceder sería anotar la mejora, para más adelante añadirla a la especificación, y cuando las pruebas reflejen esa mejora se implementará, pero no antes. De esta forma nos aseguramos de que siempre todas las funcionalidades implementadas están siendo verificadas por alguna de las pruebas.
- Una vez las pruebas funcionan, refactorizaremos el código escrito para conseguir un código más limpio (por ejemplo para eliminar segmentos de código duplicados). Volviendo a ejecutar las pruebas escritas podremos comprobar que la refactorización no ha hecho que nada deje de funcionar.
- Por último, si todavía nos quedan funcionalidades por implementar, repetiremos el proceso seleccionando una nueva funcionalidad.

Estas pruebas también se denominan pruebas *red-green-refactor*, debido a que primero deberemos comprobar que las pruebas fallan (luz roja), y de esta forma tener confianza en que nuestras pruebas están comprobando algo. Después implementamos el código para conseguir que el test funcione (luz verde), y por último refactorizamos el código.

Una de las ventajas de esta tecnología es que se consigue un código de gran calidad, en el que vamos a tener una gran confianza, ya que va a estar probado desde el primer momento. Además, nos asegura que las pruebas verifican todos los requerimientos de la aplicación. Al eliminar los errores de forma temprana, con esta metodología se evita tener que depurar un código más complejo.

Fixtures

Es probable que en varias de las pruebas implementadas se utilicen los mismos datos de entrada o de salida esperada, o que se requieran los mismos recursos. Para evitar tener código repetido en los diferentes métodos de *test*, podemos utilizar los llamados *fixtures*, que son elementos fijos que se

crearán antes de ejecutar cada prueba. Para inicializar estos elementos fijos utilizaremos métodos marcados con las siguientes anotaciones:

Anotación	Comportamiento
@Before	El método se ejecutará antes de cada prueba (antes de ejecutar cada uno de los métodos marcados con @Test). Será útil para inicializar los datos de entrada y de salida esperada que se vayan a utilizar en las pruebas.
@After	Se ejecuta después de cada <i>test</i> . Nos servirá para liberar recursos que se hubiesen inicializado en el método marcado con @Before.
@BeforeClass	Se ejecuta una sola vez antes de ejecutar todos los <i>tests</i> de la clase. Se utilizarán para crear estructuras de datos y componentes que vayan a ser necesarios para todas las pruebas. Los métodos marcados con esta anotación deben ser estáticos.
@AfterClass	Se ejecuta una única vez después de todos los <i>tests</i> de la clase. Nos servirá para liberar los recursos inicializados en el método marcado con @BeforeClass, y al igual que este último, sólo se puede aplicar a métodos estáticos.

Imaginemos que tenemos una clase ColaMensajes a la que se pueden añadir una serie de mensajes de texto hasta llegar a un límite de capacidad. Cuando se rebase dicho límite, el mensaje más antiguo será eliminado. Para probar los métodos de esta clase en muchos casos nos interesará tener como entrada una cola llena. Para evitar repetir el código en el que se inicializa dicha cola, podemos hacer uso de *fixtures*:

```
public class ColaMensajesTest {

    ColaMensajes colaLlena3;

    @Before
    public void setUp() throws Exception {
        colaLlena3 = new ColaMensajes(3);
        colaLlena3.insertarMensaje("1");
        colaLlena3.insertarMensaje("2");
        colaLlena3.insertarMensaje("3");
    }

    @Test
    public void testInsertarMensaje() {
        List<String> listaEsperada = new ArrayList<String>();
        listaEsperada.add("2");
        listaEsperada.add("3");
        listaEsperada.add("4");
        colaLlena3.insertarMensaje("4");
        assertEquals(listaEsperada, colaLlena3.obtenerMensajes());
    }

    @Test
    public void testNumMensajes() {
        assertEquals(3, colaLlena3.numMensajes());
    }

    @Test
    public void testExtraerMensaje() {
        assertEquals("1", colaLlena3.extraerMensaje());
    }
}
```


Objetos mock

Hasta ahora hemos visto ejemplos muy sencillos, en los que el método a probar recibe todos los datos de entrada necesarios mediante parámetros. Sin embargo, en una aplicación real en la mayoría de los casos el comportamiento de los métodos no dependerá únicamente de los parámetros de entrada, sino que también dependerá de otros datos, como por ejemplo datos almacenados en una base de datos, o datos a los que se accede a través de la red. Estos datos también son una entrada del método de probar, pues el resultado del mismo depende de ellos, por lo que en nuestras pruebas de JUnit deberíamos ser capaces de fijarlos para poder predecir de forma determinista el resultado del método a probar.

Sin embargo, dado que muchas veces dependen de factores externos al código de nuestra aplicación, es imposible establecer su valor en el código de JUnit. Por ejemplo, imaginemos que el método `calculaSalarioNeto` visto como ejemplo anteriormente, dado que los tramos de las retenciones varían cada año, en lugar de utilizar unos valores fijos se conecta a una aplicación de la Agencia Tributaria a través de Internet para obtener los tramos actuales. En ese caso el resultado que devolverá dependerá de la información almacenada en un servidor remoto que no controlamos. Es más, imaginemos que la especificación del método nos dice que nos devolverá `BRException` si no puede conectar al servidor para obtener la información. Deberíamos implementar dicho caso de prueba, pero en principio nos es imposible especificar como entrada en JUnit que se produzca un fallo en la red. Tampoco nos vale cortar la red manualmente, ya que nos interesa tener una batería de pruebas automatizadas.

La solución para este problema es utilizar objetos *mock*. Éstos son objetos "impostores" que implementaremos nosotros y que se harán pasar por componentes utilizados en nuestra aplicación, permitiéndonos establecer su comportamiento según nuestros intereses. Por ejemplo, supongamos que el método `calculaSalarioNeto` está accediendo al servidor remoto mediante un objeto `ProxyAeat` que es quien se encarga de conectarse al servidor remoto y obtener la información necesaria de él. Podríamos crearnos un objeto `MockProxyAeat`, que se hiciese pasar por el objeto original, pero que nos permitiese establecer el resultado que queremos que nos devuelva, e incluso si queremos que produzca alguna excepción. A continuación mostramos el código que tendría el método a probar dentro de la clase `EmpleadoBR`:

```
public float calculaSalarioNeto(float salarioBruto) {
    float retencion = 0.0f;

    if(salarioBruto < 0) {
        throw new BRException("El salario bruto debe ser positivo");
    }

    ProxyAeat proxy = getProxyAeat();
    List<TramoRetencion> tramos;
    try {
        tramos = proxy.getTramosRetencion();
    } catch (IOException e) {
        throw new BRException(
            "Error al conectar al servidor de la AEAT", e);
    }

    for(TramoRetencion tr: tramos) {
        if(salarioBruto < tr.getLimiteSalario()) {
            retencion = tr.getRetencion();
            break;
        }
    }

    return salarioBruto * (1 - retencion);
}
```

```

}

ProxyAeat getProxyAeat() {
    ProxyAeat proxy = new ProxyAeat();
    return proxy;
}

```

Ahora necesitamos crear un objeto `MockProxyAeat` que pueda hacerse pasar por el objeto original. Para ello haremos que `MockProxyAeat` herede de `ProxyAeat`, sobrescribiendo los métodos para los que queramos cambiar el comportamiento, y añadiendo los constructores y métodos auxiliares que necesitemos. Debido al polimorfismo, este nuevo objeto podrá utilizarse en todos los lugares en los que se utilizaba el objeto original:

```

public class MockProxyAeat extends ProxyAeat {

    boolean lanzarExcepcion;

    public MockProxyAeat(boolean lanzarExcepcion) {
        this.lanzarExcepcion = lanzarExcepcion;
    }

    @Override
    public List<TramoRetencion> getTramosRetencion()
        throws IOException {
        if(lanzarExcepcion) {
            throw new IOException("Error al conectar al servidor");
        }

        List<TramoRetencion> tramos = new ArrayList<TramoRetencion>();
        tramos.add(new TramoRetencion(1000.0f, 0.0f));
        tramos.add(new TramoRetencion(1500.0f, 0.16f));
        tramos.add(new TramoRetencion(Float.POSITIVE_INFINITY, 0.18f));

        return tramos;
    }
}

```

Ahora debemos conseguir que dentro del método a probar se utilice el objeto *mock* en lugar del auténtico, pero deberíamos hacerlo sin modificar ni el método ni la clase a probar. Podremos hacer esto de forma sencilla si hemos utilizado métodos de *factoría* para tener acceso a estos componentes. Podemos crear una subclase de `EmpleadoBR` en la que se sobrescriba el método de factoría que se encarga de obtener el objeto `ProxyAeat`, para que en su lugar nos instancie el *mock*:

```

class TestableEmpleadoBR extends EmpleadoBR {

    ProxyAeat proxy;

    public void setProxyAeat(ProxyAeat proxy) {
        this.proxy = proxy;
    }

    @Override
    ProxyAeat getProxyAeat() {
        return proxy;
    }
}

```

Si nuestra clase a probar no tuviese un método de factoría, siempre podríamos refactorizarla para extraer la creación del componente que queramos sustituir a un método independiente y así permitir introducir el *mock* de forma limpia.

Nota

Tanto los objetos *mock* como cualquier clase auxiliar que hayamos creado para las pruebas, deberá estar contenida en el directorio de código de pruebas (test). En el directorio de código de la aplicación (src) sólo deberán quedar los componentes que sean necesarios para que la aplicación funcione cuando sea puesta en producción.

El código de JUnit para probar nuestro método podría quedar como se muestra a continuación:

```
TestableEmpleadoBR ebr;
TestableEmpleadoBR ebrFail;

@Before
public void setUpClass() {
    ebr = new TestableEmpleadoBR();
    ebr.setProxyAeat(new MockProxyAeat(false));

    ebrFail = new TestableEmpleadoBR();
    ebrFail.setProxyAeat(new MockProxyAeat(true));
}

@Test
public void testCalculaSalarioNeto1() {
    float resultadoReal = ebr.calculaSalarioNeto(2000.0f);
    float resultadoEsperado = 1640.0f;
    assertEquals(resultadoEsperado, resultadoReal, 0.01);
}

@Test(expected=BRException.class)
public void testCalculaSalarioNeto10() {
    ebrFail.calculaSalarioNeto(1000.0f);
}
```

Podremos utilizar *mocks* para cualquier otro tipo de componente del que dependa nuestro método. Por ejemplo, si en nuestro método se utiliza un generador de números aleatorios, y el comportamiento varía según el número obtenido, podríamos sustituir dicho generador por un *mock*, para así poder predecir en nuestro código el resultado que dará. De especial interés son las pruebas de métodos que dependen de los datos almacenados en una base de datos, ya que son los que nos encontraremos con más frecuencia, y en los que los *mocks* también nos pueden resultar de ayuda.