Maven y manejo de dependencias

Maven y manejo de dependencias **1**
- ¿Qué aprenderás? 2
- Introducción 2
- ¿Qué es Maven? 3
  - ¿Cómo hace Maven para conocer la dependencia de las librerías que estoy utilizando? 4
- Concepto de Artefacto 5
  - pom.xml 5
- Creando un proyecto Maven 6
  - Ejemplo de inyección de dependencias 9

¡Comencemos!

¿Qué aprenderás?

- Conocer la herramienta Maven.
- Conocer manejo de dependencias con Maven.
- Aplicar inyección de dependencias con Maven.

Introducción

A continuación, se explicará qué es Maven y para qué sirve, también se demostrará el concepto de inyección de dependencias con Maven mediante un sencillo ejemplo.

¡Vamos con todo!

¿Qué es Maven?

Maven es una herramienta para la gestión y el manejo de librerías, repositorios y proyectos dispuestas en un servidor para el uso de éstas en un sistema. Maven, define un ciclo de vida para la ejecución de un objeto, este se basa en siete etapas:

1. **Validar:** Valida que la estructura del proyecto esté correcta.

2. **Compilar:** Compila el código.

3. **Probar:** Genera pruebas unitarias en el código compilado. Estas pruebas no necesitan que el código esté empaquetado.

4. **Empaquetar:** Toma el código compilado y genera paquetes en formato para la distribución (por ejemplo .JAR).

5. **Verificar:** Ejecuta la verificación de los resultados de las pruebas, asegurando la calidad.

6. **Instalar:** Instala los paquetes en una carpeta local para poder utilizarlos como dependencias.

7. **Implementación:** Copia el paquete final en un repositorio remoto para compartirlo.

Maven permite importar repositorios o librerías al sistema que estemos desarrollando de manera rápida y sencilla, permitiendo que nuestro proyecto acceda y los utilice, además, se encarga de reconocer todo el árbol de dependencia de los proyectos, facilitando que el desarrollador no esté preocupado de las librerías que dependen de la librería que estemos usando.

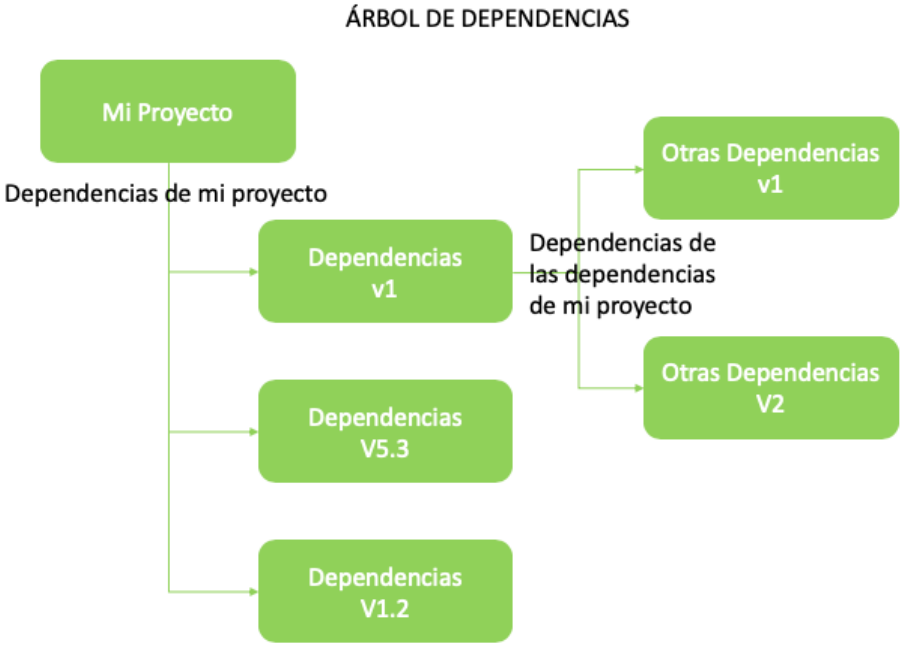


Imagen 1. Árbol de dependencias.
Fuente: Desafío Latam

¿Cómo hace Maven para conocer la dependencia de las librerías que estoy utilizando?

Maven implementa un concepto llamado Artefacto (*Artifact*), este es un bloque de código reutilizable que viene con metadatos (nombre, versión, dependencias, entre otros), debido a esto, un artefacto es un paquete de información listo para ser utilizado en cualquier proyecto. Por ende Maven, además de preocuparse por el nombre de la librería que se quiere utilizar en el proyecto y de la versión de este, se encarga de toda la información extra que posee la librería. Como por ejemplo: cuáles son sus dependencias, qué formato tiene, cuál es el grupo al que pertenece, entre otros.

Concepto de Artefacto

Un artefacto es un proyecto que gestiona Maven, el que permite conocer todo lo relacionado a la librería o directorio, ya sea su nombre, su versión, su grupo, sus dependencias, entre otros. Además, lo puede manipular y generar basándose en el ciclo de vida del repositorio.

Imagen 2. Artefacto.
Fuente: Desafío Latam

pom.xml

Maven utiliza un archivo llamado *pom.xml* (*Project Object Model*) que maneja toda la información relacionada al proyecto, agregando así el nombre, la dependencia, el identificador único de artefacto, entre otras etiquetas.

Creando un proyecto Maven

Crearemos un proyecto Maven en nuestro IDE Eclipse, en nuevo proyecto, seleccionamos la opción de *Maven Project* y le damos *next*:

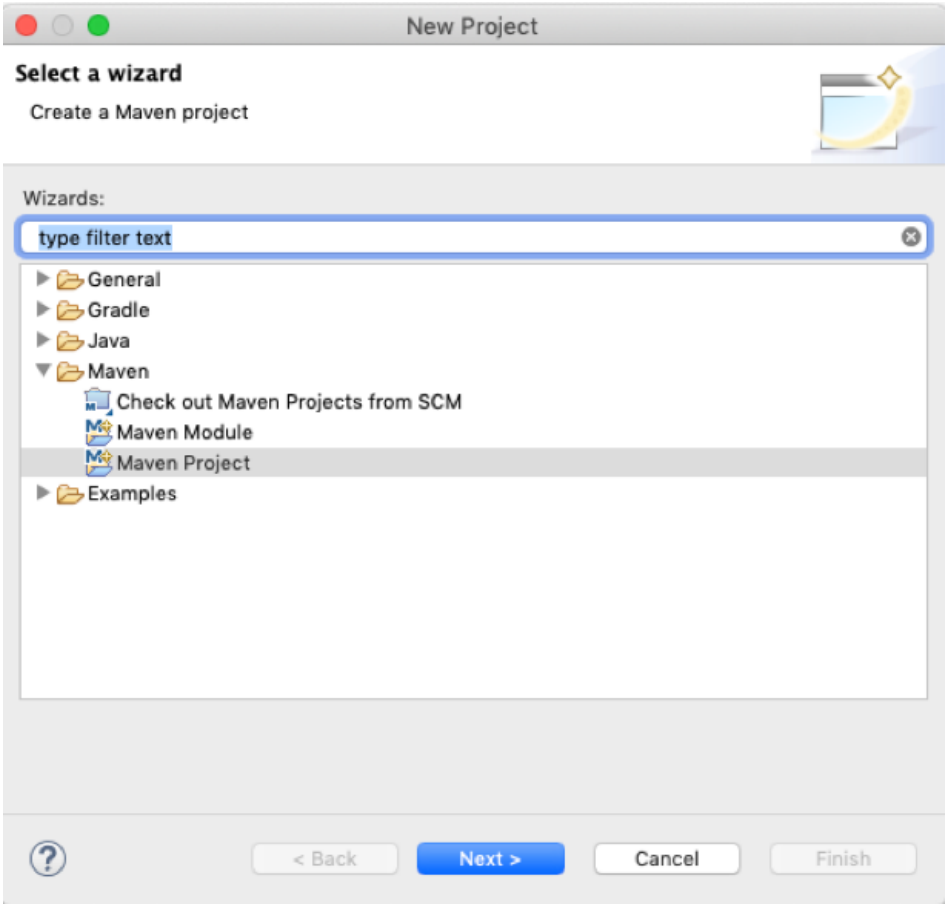


Imagen 3. Nuevo Proyecto.
Fuente: Desafío Latam

Después, seleccionamos "*Create a simple project*" para la configuración por defecto, luego esta información puede ser cambiada a nivel de código:

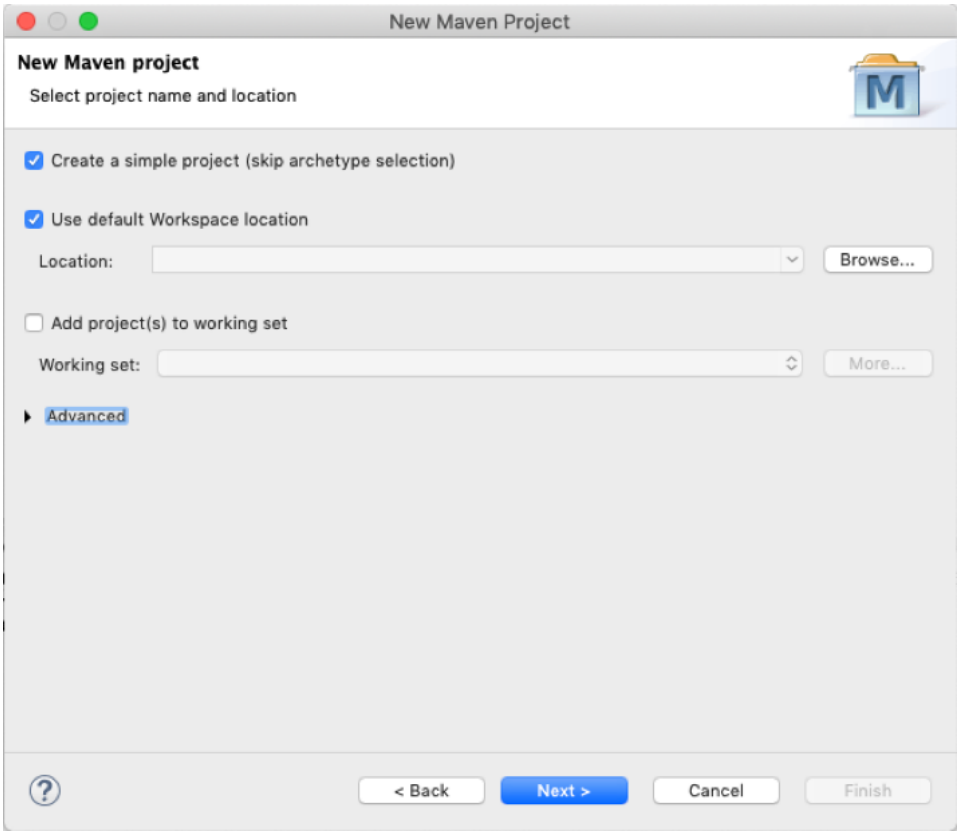


Imagen 4. Modificando las opciones del proyecto.
Fuente: Desafío Latam

Definimos la identificación del grupo y la identificación del artefacto, dejando los demás valores por defecto:

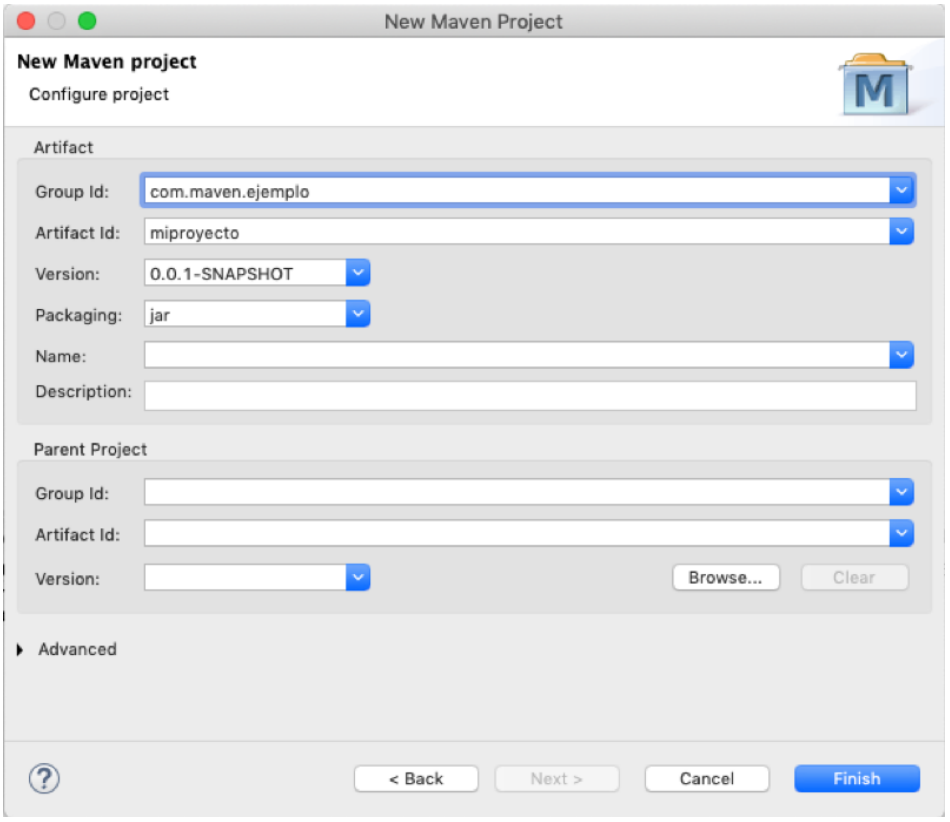


Imagen 5. Identificamos el grupo y artefacto.
Fuente: Desafío Latam

Finalmente, nos creará la estructura básica de un proyecto Maven. Esta estructura básica está definida por Maven para estandarizar su uso.

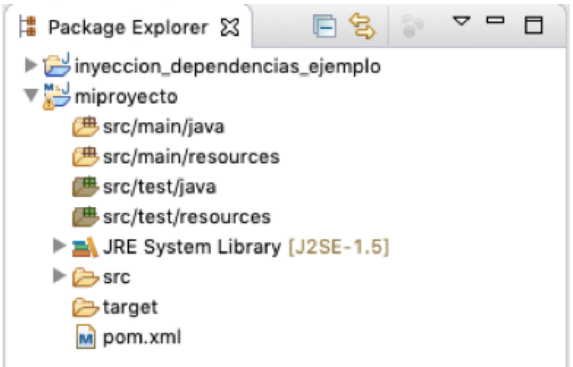


Imagen 6. Estructura del proyecto creado.
Fuente: Desafío Latam

Ahora, observamos el contenido del archivo *pom.xml*, que generó eclipse automáticamente.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
 <modelVersion>4.0.0</modelVersion>
 <groupId>com.maven.ejemplo</groupId>
 <artifactId>miproyecto</artifactId>
 <version>0.0.1-SNAPSHOT</version>
</project>
```

Este archivo es la base que posee la información de nuestro proyecto, aquí es donde Maven juega un papel muy importante, ya que es él quien utiliza esta información para generar artefactos (proyecto o librería) que podrías utilizar como dependencias de otro proyecto.

En este archivo se observa qué es un proyecto Maven, del grupo com.maven.ejemplo y artefacto miproyecto con versión *0.0.1-SNAPSHOT* (significa que el proyecto está preparándose para la versión 1). Esta información es la que se escribe al momento de generar el proyecto, aquí mismo, es donde se maneja la información de todas las dependencias que utilice el proyecto.

Ejemplo de inyección de dependencias

Supongamos que se necesita hacer una calculadora con operaciones básicas. Generamos la clase Calculadora donde tendrá métodos con las operaciones que se requieran hacer.

```
package com.micalculadora;
public class Calculadora{
   public static double suma (double a, double b) {
      return (a+b);
   }
   public static double multiplica (double a, double b) {
      return (a*b);
   }
   public static double resta (double a, double b) {
      return (a-b);
   }
   public static double divide (double a, double b) {
      return (a/b);
   }
   public static double resto (double a, double b) {
      return (a%b);
```

```
    }
}
```

Hasta aquí, nuestro código no hace nada, entonces según Maven, deberíamos crear pruebas unitarias, para esto utilizaremos el directorio de dependencia proporcionada por Maven para crear pruebas, llamada JUnit.

Vamos al sitio de Maven y buscamos JUnit y accedemos a la segunda opción.

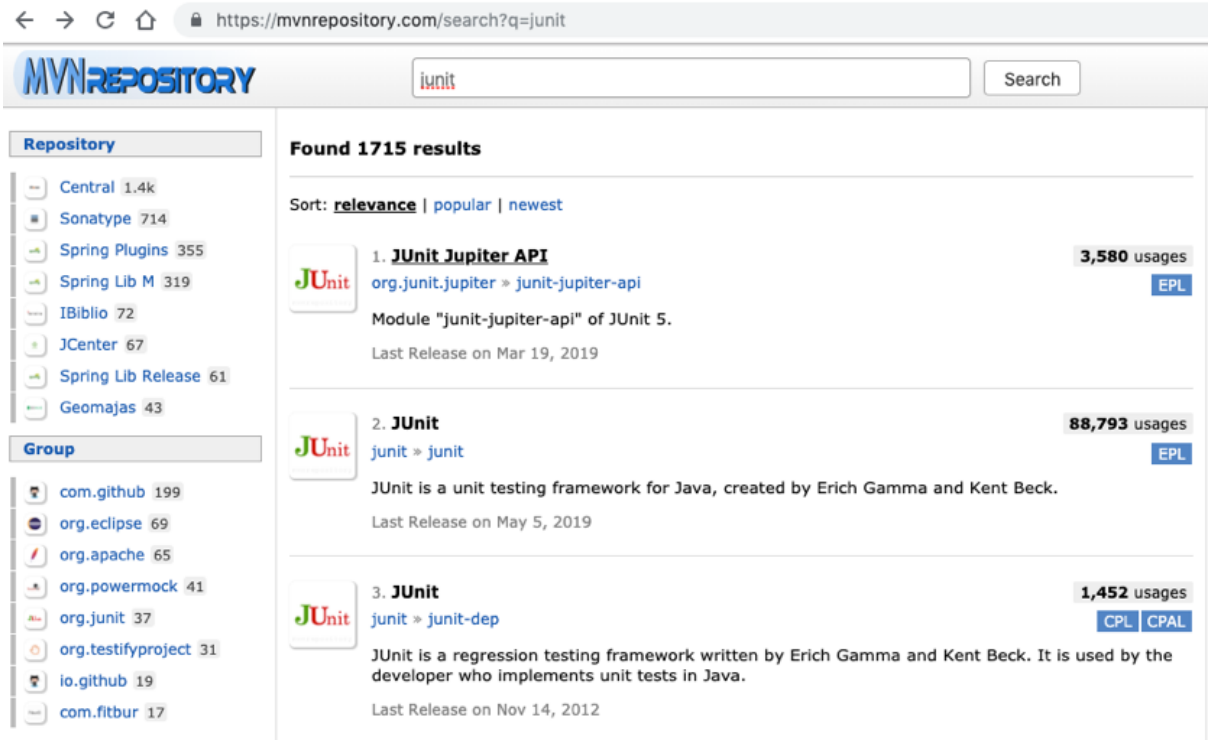


Imagen 7. Buscando JUnit
Fuente: Desafío Latam.

Una vez ahí, buscamos la última versión estable (no se recomienda trabajar con versiones beta), que sería la versión 4.12.

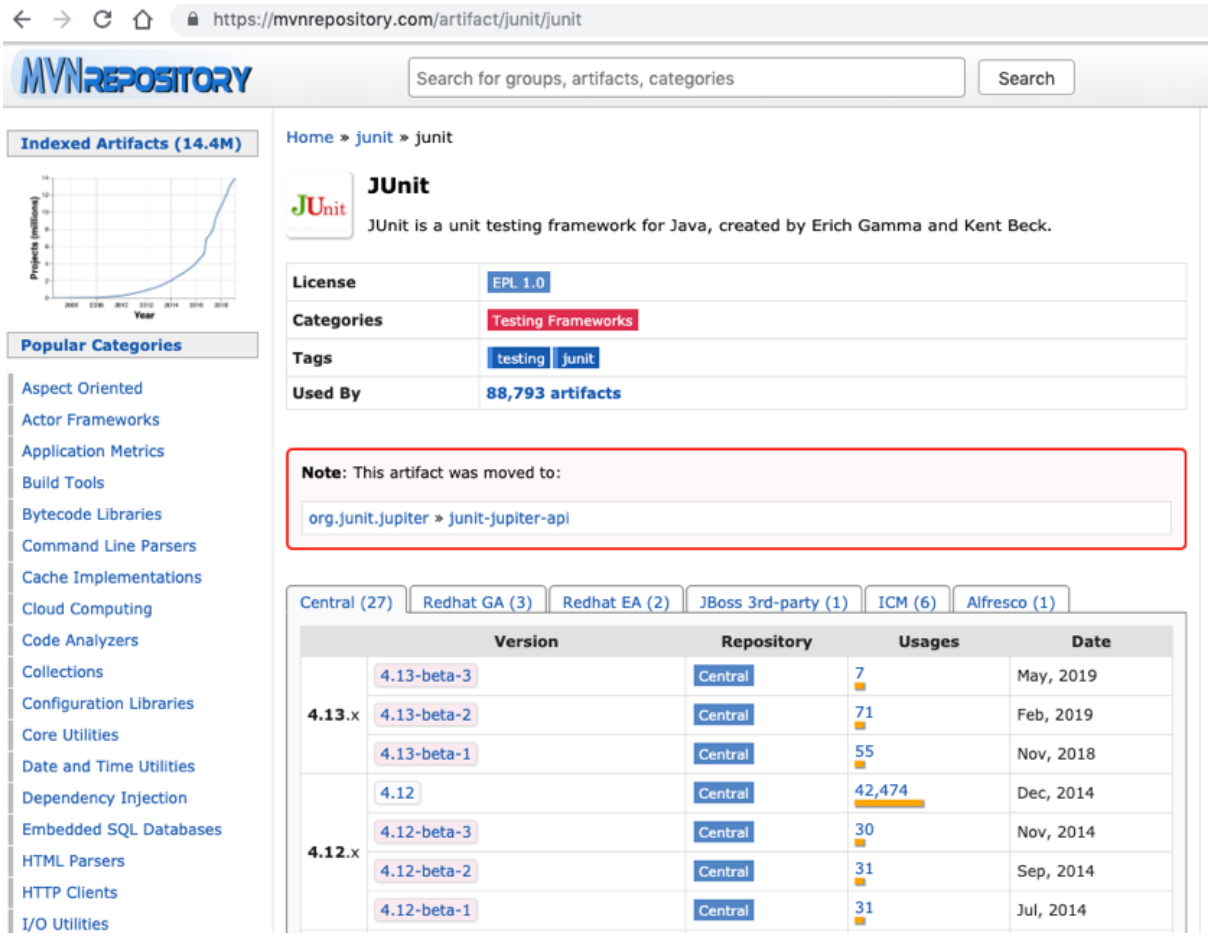


Imagen 8. Versión de JUnit.
Fuente: Desafío Latam

Aparece cómo incorporar el repositorio en Maven, copiamos ese código y lo pegamos en nuestro archivo pom.xml entre las etiquetas *<dependencies></dependencies>*.

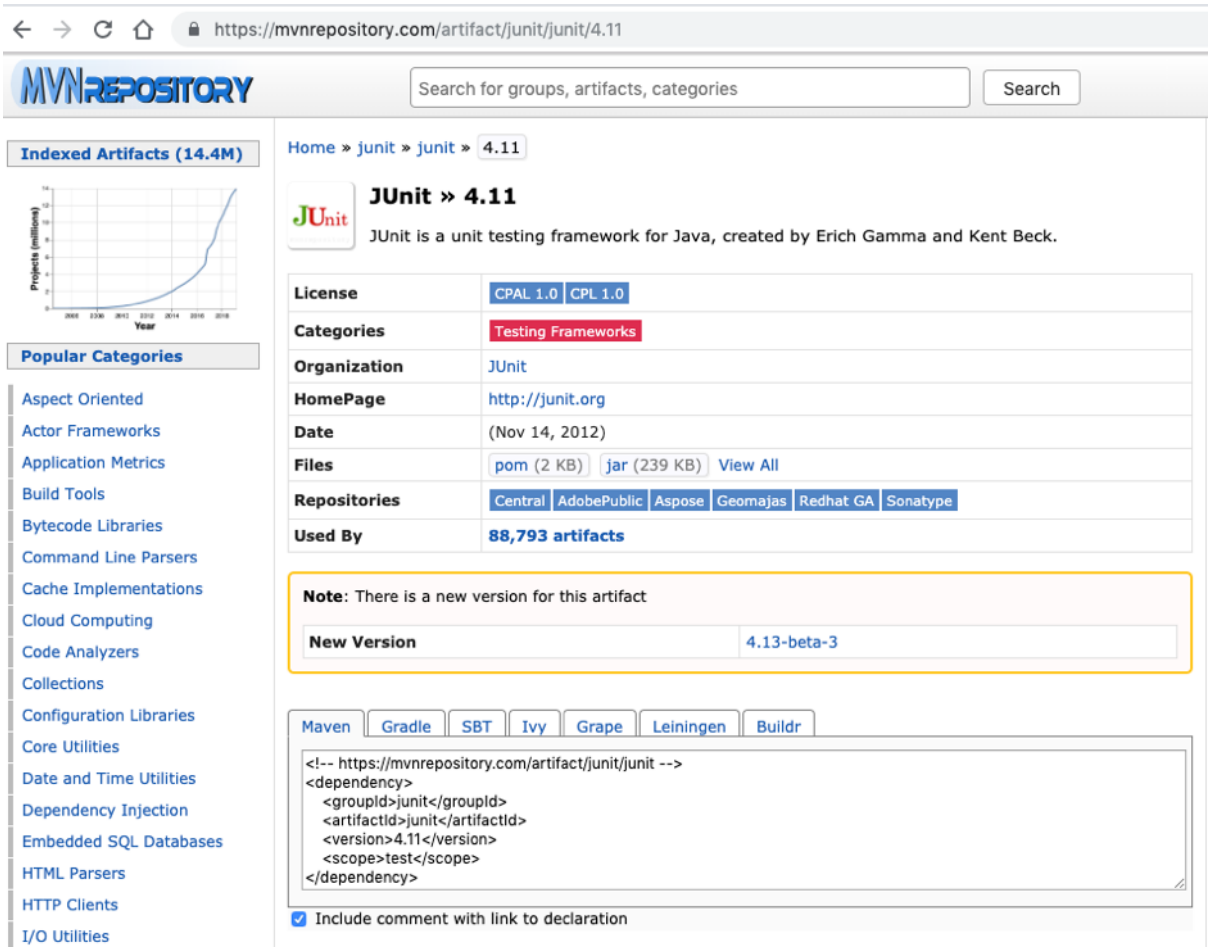


Imagen 9. Incorporando el repositorio con Maven.
Fuente: Desafío Latam

Nuestro archivo *pom.xml* se verá así.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
 <modelVersion>4.0.0</modelVersion>
 <groupId>com.calculadora</groupId>
 <artifactId>micalculadora</artifactId>
 <version>0.0.1-SNAPSHOT</version>
<dependencies>
   <!-- https://mvnrepository.com/artifact/junit/junit -->
   <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.11</version>
      <scope>test</scope>
   </dependency>
</dependencies>
</project>
```

Con este cambio en el archivo *pom.xml*, la estructura de nuestro proyecto agregó un nuevo repositorio *Maven Dependencies*, refiriéndose a las dependencias agregadas en el archivo pom.xml.

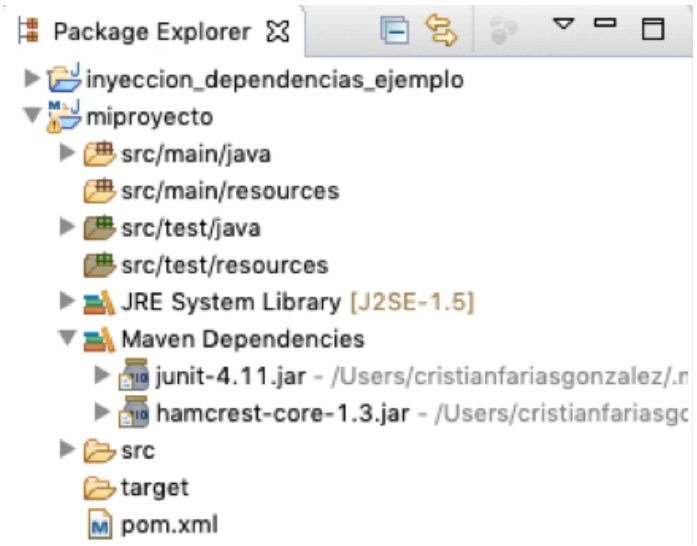
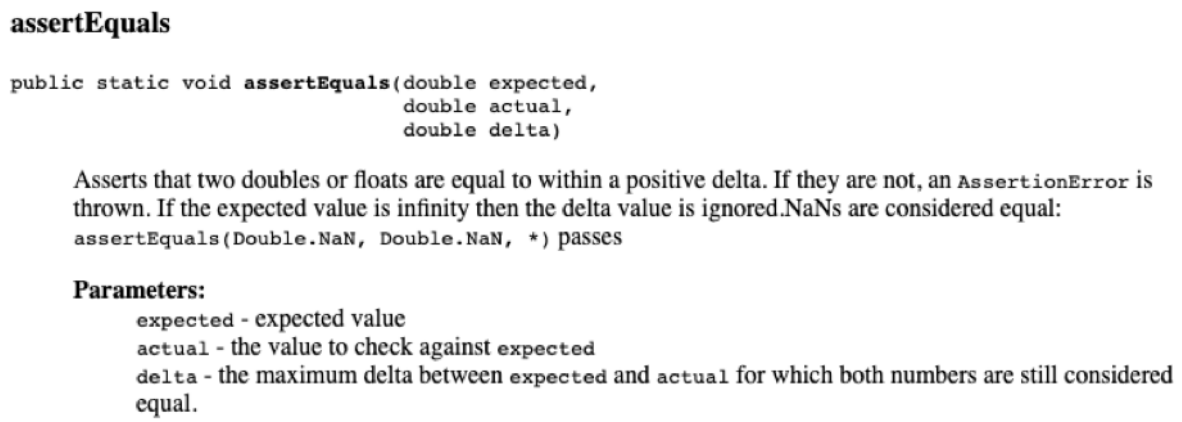


Imagen 10. Estructura del proyecto.
Fuente: Desafío Latam

Una vez agregada la librería que nos ayudará con las pruebas unitarias, haremos nuestra prueba unitaria, creamos una nueva clase que permita generar nuestras pruebas, utilizando el método *assertEquals*, de la clase de Assert permite realizar la comparación de valores. Documentación del método *assertEquals* utilizado:



assertEquals

```
public static void assertEquals(double expected,
                                double actual,
                                double delta)
```

Asserts that two doubles or floats are equal to within a positive delta. If they are not, an `AssertionError` is thrown. If the expected value is infinity then the delta value is ignored.NaNs are considered equal: `assertEquals(Double.NaN, Double.NaN, *)` passes

Parameters:

`expected` - expected value
`actual` - the value to check against `expected`
`delta` - the maximum delta between `expected` and `actual` for which both numbers are still considered equal.

Imagen 11. Assert utilizado.
Fuente: Desafío Latam

Esto quiere decir que recibe 3 parámetros:

- El primero es el valor que espera.
- El segundo es el valor actual, vale decir el valor resultado de la operación.
- El tercero, que es el delta, es el porcentaje de error de la igualación.

Generamos nuestro archivo de pruebas unitarias:

```
package com.micalculadora;
import org.junit.Test;
import static org.junit.Assert.*;
import com.micalculadora.Calculadora;
public class CalculadoraPrueba {
   @Test
   public void prueba() {
      assertEquals(4,Calculadora.suma(2,2),0);
      assertEquals(4,Calculadora.multiplica(2,2),0);
      assertEquals(0,Calculadora.resta(2,2),0);
      assertEquals(1,Calculadora.divide(2,2),0);
      assertEquals(0,Calculadora.resto(2,2),0);
   }
}
```

El resultado de esto nos refleja que nuestras pruebas son correctas.

Imagen 12. Resultado.
Fuente: Desafío Latam

A continuación, generemos el error, para comprobar que la librería realmente funciona, sólo cambiamos el número de comprobación de la suma y como tiene un valor de error de 0, la igualación debe ser estricta, este cambio nos arroja el error que verá:

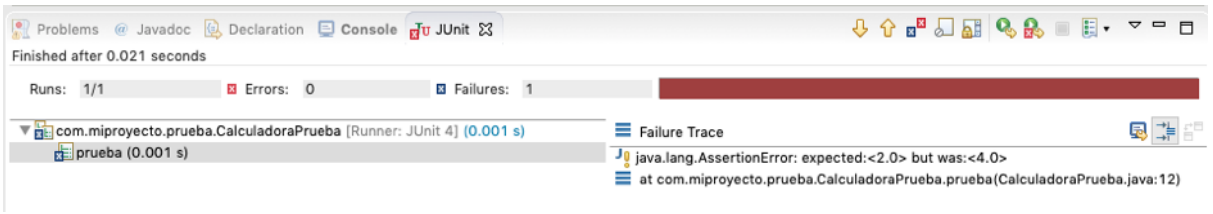


Imagen 13. Generando un error.
Fuente: Desafío Latam

Ahí en la información del error, dice que se esperaba un 2 y el resultado de la función fue 4. La clase que se utilizó para generar el error:

```
package com.miproyecto.prueba;
import org.junit.Test;
import static org.junit.Assert.*;
import com.miproyecto.Calculadora;
public class CalculadoraPrueba {
   @Test
   public void prueba() {
      assertEquals(2,Calculadora.suma(2,2),0);
   }
}
```

Con esto, ya demostramos el uso de dependencias, y no sólo las propuestas por Maven, si no que también de la clase que fue creada por nosotros.