Universidad Nacional de Costa Rica

Facultad de Ciencias Exactas y Naturales

Escuela de Informática

Herramientas de automatización y Testing de aplicaciones modernas

Profesor: Carlos Loría Sáenz

Estudiantes:
Braslyn Rodríguez Ramírez
Enrique Méndez Cabezas
Philippe Gairaud Quesada

Ciclo II 2020

Contenido

Tabla de Ilustraciones

Introducción

El siguiente documento corresponde a la Tarea de Investigación: Sobre Herramientas de automatización moderna de aplicaciones para el curso de Paradigmas de Programación, se abarcan diferentes temáticas relacionadas con los conceptos principales a investigar como el concepto de build dentro de la ingeniería de software, herramientas de build, herramientas de testing, manejo de versiones e integración con herramientas de desarrollo y entrega continuos, tratando de relacionar estos temas con los conceptos de metodologías agiles y dev-ops.

El objetivo principal de este documento es el de funcionar como una introducción y guía al concepto de build y las herramientas utilizadas para completar este proceso como por ejemplo Ant y Maven las cuales serán detalladas en los próximos apartados con la finalidad de que el lector al finalizar la lectura del documento obtenga un mayor conocimiento en las temáticas desarrolladas y pueda utilizar correctamente tanto las herramientas de build abarcadas como las herramientas de testing.

Desarrollo

Proceso de build

En ingeniería de software el concepto de build se conoce como el proceso en el que se traduce el código escrito por el programador en un elemento de software que es ejecutado por la computadora. (Verma, scmQuest, n.d.)

Básicamente, el proceso de build se puede describir como una serie de pasos o actividades realizadas en un orden cronológico concreto que varían según cada lenguaje y cada sistema operativo en el que se crea el programa tomando todos los archivos de código fuente, librerías, archivos de bases de datos, entre otros para posteriormente ser compilados y finalmente generar una versión ejecutable del programa.

El proceso de build se puede dividir en los siguientes pasos:

- Obtener la última versión del código directamente del repositorio de código establecido para el proyecto.
- Comprobar las dependencias utilizadas en el proyecto.
- Compilar el código.
- Realizar las pruebas unitarias(unit test) de cada módulo correspondiente.
- Enlazar las librerías, código, y otros archivos necesarios (Etapa de linking).
- Deploy del proyecto (ya sea para pruebas o para producción).

Todos estos pasos descritos anteriormente pueden ser realizados mediante el uso de herramientas de build las cuales automatizan este proceso, algunas de estas herramientas se explicarán en detalle en la siguiente sección de este documento, específicamente en la sección titulada como Herramientas de Build.

Integración continúa

La integración continua es un concepto el cual se encentra fuertemente relacionado con el tema de build. La integración continua corresponde a una práctica de desarrollo en la cual todos los cambios implementados por el equipo de programadores se integran de manera periódico en un repositorio de código.

La importancia de la integración continua recae principalmente en la facilidad que ofrece a la hora de la combinación de cambios ya que es en esta etapa donde se puede dar una acumulación de errores lo que directamente afecta la velocidad de la integración de los cambios y el tiempo de respuesta a las necesidades del cliente.

Herramientas de Build.

Maven

Maven es una herramienta de *build* desarrollada por *Apache Software Foundation* orientada principalmente al proceso de "*building"* de Java que permite la compilación, testeo y distribución, también puede ser considerada como una herramienta de administración de proyectos ya que esta permite generar reportes y facilitar la comunicación entre un grupo de trabajo.

Conceptos básicos

Maven trabaja toda su configuracion en el archivo *"pom.xml" y* por defecto asume que la carpeta del código fuente *"${basedir}/src/main/java"* los recursos en *"${basedir}/src/main/resources"*, tambien se espera que el resultado sea un JAR, en donde guardara el byte code en *"{basedir}/target/classes"* y el .JAR para la distribución en *"{basedir}/target"*.

Repositorios Maven

En Maven un repositorio es el encargado de almacenar los diferentes artifacts y dependencias. Este repositorio puede ser del tipo local o remoto, en cuanto al repositorio local, corresponde a un directorio local almacenado en el equipo en el que se ejecuta Maven mientras que, para acceder a los repositorios remotos, estos se encuentran configurados por un tercero y para los que se utilizan diferentes protocolos como file:// y https:// para acceder a los artifatcs y dependencias almacenados en ellos.

Maven en el proceso de *build* busca en el archivo *pom* todas las dependencias bajo la etiqueta de <dependencies> las cuales son todas las referencias necesarias para la construcción del proyecto, por defecto el repositorio buscara en el sitio *http://repo.maven.apache.org/maven2*, también se pueden declarar repositorios alternos bajo la etiqueta <*repositories*>

Archivo POM

El archivo POM (Project Object Model) es el archivo principal de un proyecto Maven. Corresponde a un archivo XML el cual contiene todas las configuraciones, dependencias, plugins, entre otros elementos necesarias para compilar y ejecutar el proceso de build del proyecto. Todos los archivos POM extienden el archivo Super POM el cual es el archivo POM configurado por defecto de Maven lo que implica que todos los archivos POM del proyecto heredan las configuraciones inciales de este Super POM.

Según la documentacion oficial de Maven, se deben especificar mininamente los siguientes elementos:

- Project
- modelVersion: la version de pom del documento.
- groupId: Id de grupo generalmente único para cada proyecto u organización.

- artifactId: Nombre del proyecto
- Version: version actual del proyecto

La siguiente imagen corresponde a la configuración inicial básica de un archivo POM:

```
1. <project>
2.   <modelVersion>4.0.0</modelVersion>
3.
4.   <groupId>com.mycompany.app</groupId>
5.   <artifactId>my-app</artifactId>
6.   <version>1</version>
7. </project>
```

Ilustración 1 Configuración inicial básica de un archivo POM, Tomado de: https://maven.apache.org/

El archivo POM contiene los siguientes elementos

- Dependencias
- Desarrolladores y Contribuidores
- Lista de Plugins
- Ejecución de los Plugins
- Configuración de los Plugins
- Recursos

Dependencias

Dentro de Maven, una dependencia corresponde a otro tipo de archivo ya sea JAR, ZIP u otro que el proyecto necesita para compilar, para el proceso de build, para testing o para ejecutar el proyecto. Como se mencionó en el punto anterior, estas dependencias se especifican dentro del archivo POM.

Es en el momento de ejecución o de build cuando Maven se encarga de resolver estas dependencias cargándolas desde el repositorio local de código (Gupta, s.f), si las dependencias no están presentes en este repositorio Maven se encarga de descargarlas de un repositorio remoto al repositorio local.

```
<dependencies>
    <dependency>
      <groupId>javax.servlet</groupId>
      <artifactId>javax.servlet-api</artifactId>
      <version>4.0.1</version>
    </dependency>
</dependencies>
```

Ilustración 2 inclusión de una dependencia dentro del archivo POM

Inclusión de una dependencia dentro del archivo POM.

Archetypes

La palabra *Archetypes* significa *"patrones originales",* este es un sistema que provee templates de proyectos maven que sirven de base para iniciar un proyecto de forma rápida y consistente, estos se pueden crear o usar los que trae Maven por defecto.

Maven Plugins

Los plugins son el corazón de Maven, estos desempeñan distintas tareas, existen los siguientes tipos:

- Plugins de *Build*
- Plugins de Reports

Cada plugin tiene un goal, la cual es responsable de una tarea en específico que siguen al momento de correr el build y tienen diferentes tipos de goal dependiendo del ciclo de vida de building se encuentre.

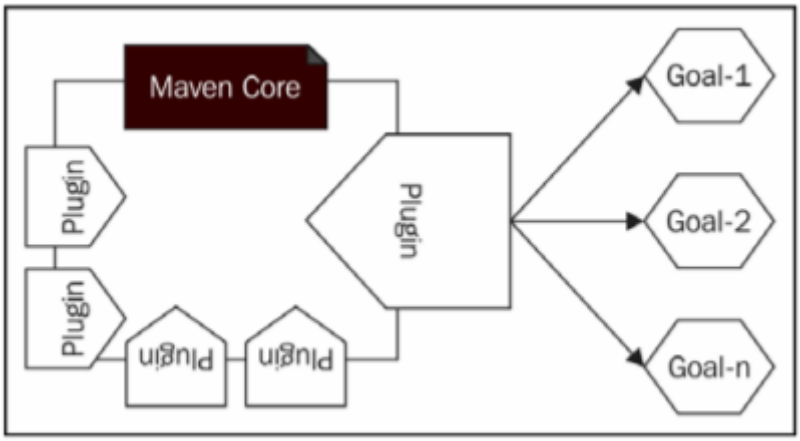

Ilustración 3 Estructura del Core de Maven con Plugins Tomado de [Bharathan, 2015]

Maven se puede correr con solo el core pero el verdadero potencial está en los plugins, Maven fue pensado en forma que no todo lo haga el core si no que se le puedan añadir funcionalidades que se ejecuten en conjunto o en determinado ciclo de vida.

Algunos plugins utilizados en un proyecto a manera de ejemplo son:

- Apache Maven Compiler Plugin
- Maven Surefire Plugin
- Apache Maven Assembly Plugin

El apache Maven Compiler se usa para compilar los sources del proyecto

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>3.8.1</version>
    <configuration>
        <release>14</release>
        <compilerArgs>--enable-preview</compilerArgs>
    </configuration>
</plugin>
```

Ilustración 4 Plugin Compiler de maven

El Apache Maven Assembly Plugin es usado para compilar los dos servidores en el proyecto de ejemplo mencionado al mismo tiempo y que tengan su manifest apuntando a la clase main.

Como se puede apreciar en las dos imágenes el plugin es incluido en el POM y configurado para que se ejecute de una determinada forma.

Ciclos de vida de "Build"

Un ciclo de vida de Build se compone de una fase bien definida, cada fase agrupa los *goals* de los plugins y el ciclo de vida determina el orden de ejecución de los mismos. (Bharathan, 2015)

Maven viene ya por defecto con tres ciclos de vida estandar

- **Clean**: ciclo en el que se borran los archivos y documentos producidos por maven
- **Default**: Este se encarga del build y del despliegue de la aplicación.
- **Site**: Encargada de la elaboración de la documentación del proyecto.

Empezando con Maven

Para empezar con Maven podemos empezar con el archetype de ejemplo en cual es

```
mvn -B archetype:generate -DgroupId=com.mycompany.app -DartifactId=my-app -DarchetypeArtifactId=maven-archetype-quickstart -DarchetypeVersion=1.4
```

Ilustración 5 archetype inicial Maven

Este arquetipo es un buen ejemplo para aprender cómo funciona el building con Maven, ya que nos proporciona la estructura básica para la carpeta de "src" y un POM básico para compilar el proyecto, para hacerlo basta con e ejecutar mvn compile.

Ventajas de utilizar Maven

- Todas las dependencias requeridas para el correcto funcionamiento del proyecto son añadidas automáticamente en el momento de leer el archivo POM. (learntek , 2019)
- Gracias al punto anterior, añadir una dependencia nueva al proyecto se puede realizar de manera sencilla especificándola en el archivo POM.
- Disponibilidad de una gran variedad de plugins.
- Maven facilita inicializar el proyecto en diferentes ambientes ya sea de desarrollo o de ejecución. (learntek , 2019)
- Configuración inicial mínima.

Desventajas de utilizar Maven

- Si el código Maven para una determinada dependencia no existe, la dependencia no se puede utilizar usando Maven.
- Tiene una curva de aprendizaje mayor comparada con otras herramientas de build. (learntek , 2019)

Gradle

Gradle es una herramienta open source para el proceso y automatizado de *build*, los scripts de gradle están escritos en en groovy o Kotlin. Gradle se ejecuta en una JVM, y es reconocido por la mayoría de los IDE's por lo que ofrece gran versatilidad para un equipo de trabajo, también es una herramienta de Building para cualquier software, el único limitante es el manejo de dependencias.

Gradle ofrece un gran rendimiento evitando el trabajo innecesario en tareas, así como el build cache lo que permite que se pueda reusar para otras tareas. Gradle se ejecuta en la JVM por lo que se puede ejecutar en diferentes plataformas que soporten java. Es extensible debido a que se pueden integrar tareas propias o sistemas de build, como lo es el soporte de build Android.

Conceptos básicos

Los archivos de Build se encuentran en groovy, el cual es un lenguaje dinámico de JVM similar a java, lo que permite crear procesos de building complejos en los que el desarrollador puede programar código en el mismo archivo de build para una tarea en especifica en el proceso, así también Gradle le presenta al usuario la posibilidad de un DSL «*Domain Specific Language*» diseñado para tareas de build . (Berglund & McCullough, 2011).

Extensibilidad

Tareas Personalizadas

Permite elaborar un trabajo que una tarea existente no permite.

Acciones personalizadas en tareas

Permite añadir lógica en el proceso de build personalizando el tiempo de ejecución de tareas, por medio de «*Task.doFirst()*», «*Task.doLast ()*».

Propiedades Extra

Permite agregar propiedades personalizadas a un proyecto o Tarea, estas se pueden utilizar en cualquier tarea.

Gradle vs Maven

Gradle es un sistema de build similar a Maven en donde la convención esta antes que la configuración, por lo que son muy parecidos, aun así, Gradle fue construido teniendo en cuanta la libertad del usuario por lo que ofrece mayores alternativas de configuración (Gradle vs Maven Comparison, n.d.)

En eficiencia Gradle posee ventaja sobre Maven, ya que posee tres funcionalidades que son:

- Incrementabilidad: Gradle Evita el trabajo innecesario ejecutando solamente las tareas cuando es necesario.
- Build Cache: Reutiliza las salidas de otra build de gradle con los mismos datos de entrada, así como entre maquina por medio de un chache compartido.
- Gradle Daemon: Un proceso de larga duración que retiene la información en memoria.

Archivo de Build

Tenemos en la imagen un archivo POM de Maven y el archivo Build de Gradle, en los que vemos la facilidad de Gradle para iniciar un proyecto simple

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/xsd/maven-4.0.0.xsd">
   <modelVersion>4.0.0</modelVersion>
   <groupId>com.mycompany.app</groupId>
   <artifactId>my-app</artifactId>
   <packaging>jar</packaging>
   <version>1.0-SNAPSHOT</version>

   <dependencies>
      <dependency>
         <groupId>junit</groupId>
         <artifactId>junit</artifactId>
         <version>4.11</version>
         <scope>test</scope>
      </dependency>
   </dependencies>
</project>
```

Gradle

```
apply plugin: 'java'
group = 'com.mycompany.app'
archivesBaseName = 'my-app'
version = '1.0-SNAPSHOT'

repositories {
   mavenCentral()
}

dependencies {
   testCompile 'junit:junit:4.11'
}
```

Ilustración 6 Comparación de entre el archivo de POM de Maven y el archivo build de Gradle tomado de https://gradle.org/books/manning-publications-gradle-in-action.pdf

Primeros Pasos con Gradle

Iniciamos en la carpeta donde queremos el proyecto, usamos el "*Gradle init"* para iniciar el proyecto. Se iniciará el Daemon si es que no está activo, y nos pedirá que tipo de proyecto generar.

```
C:\Users\ecabezas\Documents\Test-Gradle>gradle init
Starting a Gradle Daemon, 1 incompatible Daemon could not be reused, use --status for details

Select type of project to generate:
  1: basic
  2: application
  3: library
  4: Gradle plugin
Enter selection (default: basic) [1..4] 2
```

Ilustración 7 Comando « gradle init »

Para nuestro inicio rápido con Java seleccionamos la opción "2: aplication", luego nos saldrá el lenguaje a usar en el proyecto, seleccionamos el "3:Java".

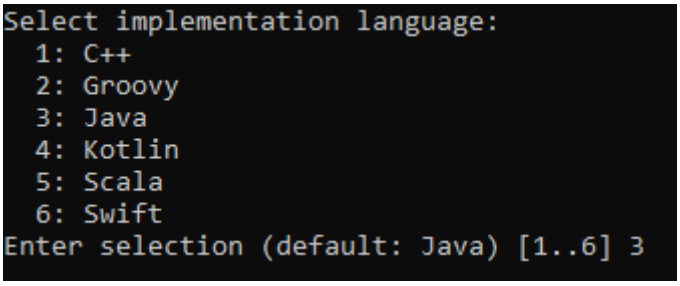

Ilustración 8 Comando « gradle init »

Después nos pedirá que tipo de DSL para el Build Script, seleccionamos “1: Groovy”

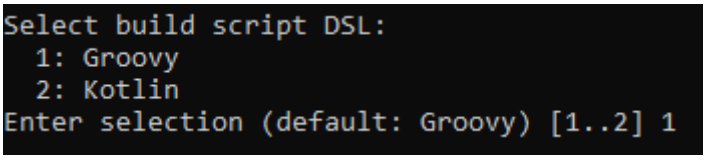

Ilustración 9 Comando « gradle init »

También se nos pedirá que framework test.

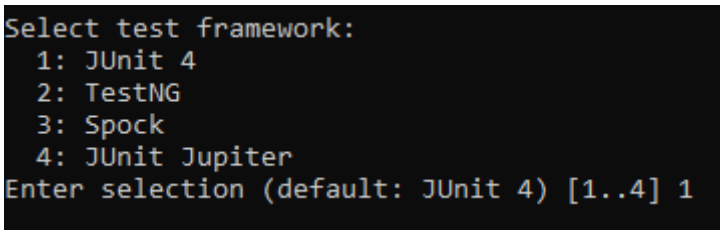

Ilustración 10 Comando « gradle init »

Una vez personalizados se nos pedirá el nombre del proyecto y el paquete fuente.

Ilustración 11 Comando « gradle init »

Ya con esta configuración básica podemos iniciar un proyecto con gradle.

Plugins

El core de Gradle provee pocas herramientas para una automatización total, por lo que se necesita de los plugins para añadir las funcionalidades que nos ayuden a compilar código de Java.

Los Plugins añaden nuevas tareas, Objetos y convenciones, también extienden los objetos core y los de otros plugins.

El uso de plugins extienden las capacidades del proyecto lo que extiende el modelo de Gradle, configura el proyecto según las convenciones y aplica las configuraciones específicas, también trae beneficios como la reutilización a través de múltiples proyectos, otorga alto grado de modularización por lo que mantiene la compresión y la organización.

Tipos de Plugins

Existen dos tipos de plugins, los plugins binarios y los plugins de script.

Plugins Binarios: Estos plugin están programados implementando la interfaz de Plugin o de forma declarativa usando uno de los DSL de Gradle. Estos plugins pueden estar dentro del build script, dentro de la jeraquia del proyecto o en un plugin jar.

Plugins de Script: Estos son build script adicionales que configuran el build y se implementa una manipulación declarativa al build.

Ant

Apache Ant es una librería de java y una herramienta de línea de comandos cual misión es llevar a cabo procesos especificados en archivos de build. (Apache Ant - Welcome, n.d.).

Ant escribe sus archivos de build en XML, por lo que la hace familiar para la gran mayoría de personas y es un archivo que no sufre cambios en los diferentes sistemas operativos por lo que lo hace portable.

Ant Targets y Task

Cada sección nombrada en el proceso de build es un target, los que tienen un determinado número de tareas.

Estructura de un archivo Ant build

```
<project name="Example Application Build" default="default" basedir=".">

  <!-- Compile the stand-alone application -->
  <target name="default">
    <javac srcdir="./src" destdir="build"/>
    <echo message="Application compiled"/>
  </target>

</project>
```

Ilustración 12 Archivo Build de Ant

La etiqueta Project es el elemento raíz de cada archivo de build Ant, y define el objetivo *target* para la construcción del proyecto.

Las tareas de Ant se dividen en tres categorías: core, optional, y personalidades; Las Tareas *Core* son las que el equipo de desarrollo de Ant, las tareas *opcionales* están ligadas con Ant y dependen de librerías, las Tareas perso dan la ventaja de de ser flexibles, son implementadas en tareas que requieren un paso que requiera intervención del Command Line.

Propiedades, son una manera de personalizar un proceso de build proveer atributos que se usan repetidamente.

Herramientas de Testing

Las herramientas de testing corresponde a software que facilita al programador el ejecutar pruebas de una manera automática sobre su código. Actualmente se pueden encontrar una gran variedad de herramientas de testing, tanto de código abierto como de pago.

Estas herramientas se pueden clasificar según la finalidad de las pruebas para las que son utilizadas como por ejemplo búsqueda de errores, integración, GUI, seguridad o testing unitario.

En este apartado se describen específicamente dos herramientas de testing unitario una para el lenguaje Java y otra para JavaScript.

JUnit

Junit es un *framework* para pruebas de unidad, consiste en una variedad de clases las cuales pueden ser usadas para construir diversos casos de prueba en las aplicaciones Java. (Albing & Schwarz, 2005).

Características Importantes:

- Software de código abierto.
- Ejecución de pruebas de manera automática.
- Resulta relativamente sencillo de utilizar para el usuario.
- Utilización de aserciones.
- Anotaciones para identificar los metodos.
- Los casos de prueba pueden ser organizados en conjuntos llamados suites.
- Integración con IDEs como NetBeans, Eclipse e IntelliJ.

Empezando con JUnit

Configuración del entorno de desarrollo

Para poder llevar a cabo pruebas en JUnit sin la necesidad de utilizar un IDE, es necesario obtener desde el sitio oficial tanto la última versión del jar de JUnit como la última versión del jar de hamcrest. Posteriormente agregar los correspondientes jar a las variables del sistema para poder compilar y ejecutar el código Java correspondiente a las pruebas mediante la terminal.

Otra opción para utilizar JUnit es mediante la herramienta de build que se esté utilizando en el proyecto como por ejemplo en el caso de estar trabajando con Maven se debe añadir la dependencia correspondiente en el archivo pom.xml.

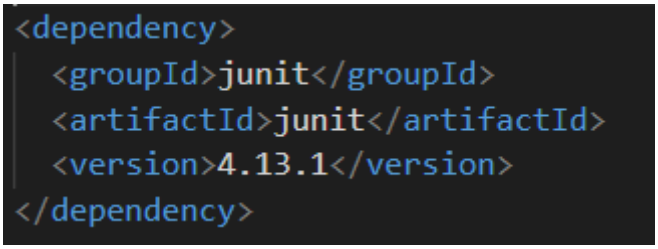

Ilustración 13 Dependencia de JUnit para Maven, Tomado de: https://junit.org/

Por último, en cuanto a la integración con IDEs, JUnit puede ser utilizado fácilmente en como NetBeans, Eclipse e IntelliJ. Cada uno de los anteriores IDEs tiene su propia forma de manejar los tests.

Primeramente, en el caso de NetBeans se debe seleccionar el archivo sobre el que se quieren ejecutar las pruebas de la siguiente forma:

Click derecho > Tools > Create/Update Tests

NetBeans desplegara una ventana como la siguiente en la cual se debe especificar JUnit como el framework de pruebas

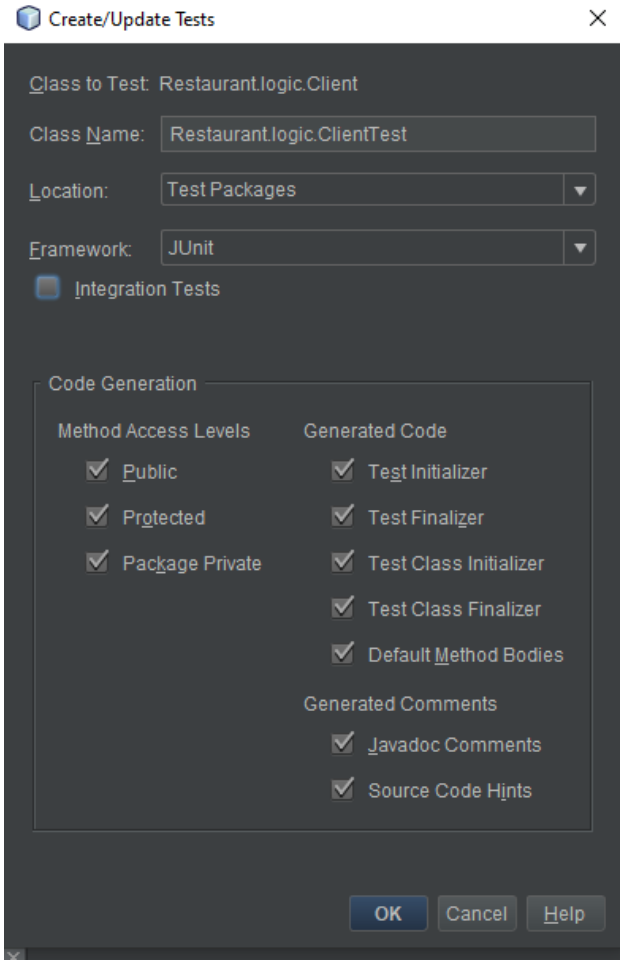

Ilustración 14 Ventana Create/Update Tests NetBeans

Finalmente, NetBeans genera automáticamente el archivo de test.java correspondiente con las pruebas de cada función.

Para el caso de IntelliJ la librería de JUnit se puede agregar automáticamente al *classpath* una vez que el IDE detecta que se está utilizando código de la misma. Además, también se puede agregar de forma manual desde la configuración del proyecto en el apartado de librerías, seleccionando específicamente las librerías de Maven y cualquiera de las siguientes opciones:

org.junit.jupiter:junit-jupiter:5.4.2 o org.testng:testng:6.14.3

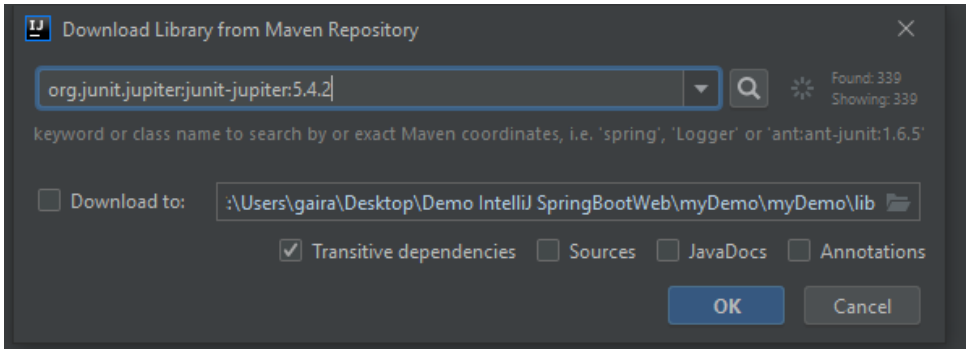

Ilustración 15 Ventana Descarga de librerías IntelliJ

Por último, se debe seleccionar una carpeta como la carpeta por defecto para pruebas haciendo click derecho sobre la carpeta y seleccionando la opción *Mark Directory as* y luego *Test Sources Root*, cabe recalcar que este paso al igual que el anterior solo son necesarios si no está utilizando alguna herramienta de build y el proyecto utilice el builder nativo del IDE.

Ejemplo inicial

El siguiente ejemplo corresponde a un ejemplo básico el cual se puede considerar un Hello World en JUnit, esto con el fin de observar un test case y de un test runner y las funciones básicas que son parte de la librería JUnit.

En primer lugar, se puede observar el unit test que queremos ejecutar el cual contiene el caso a ser probado para este ejemplo. En este caso el método assertEquals(actual, expected) recibe el valor a ser probado y el resultado esperado:

```
import org.junit.Test;
import static org.junit.Assert.assertEquals;
public class TestJunit {
   @Test
   public void test() {
      String str= "Hello World!";
      assertEquals("Hello World!",str);
   }
}
```

Ilustración 16 Ejemplo Test Case. Tomado de: http://www.java2s.com/

El test runner correspondiente el cual se encarga de ejecutar el caso de prueba anterior es el siguiente:

```java
import org.junit.runner.JUnitCore;
import org.junit.runner.Result;
import org.junit.runner.notification.Failure;


public class TestRunner {
    Run | Debug
    public static void main(String[] args) {
        Result result = JUnitCore.runClasses(TestJunit.class);
        for (Failure failure : result.getFailures()) {
            System.out.println(failure.toString());
        }
        System.out.println(result.wasSuccessful());
    }
}
```

Ilustración 17. Tomado de: Ejemplo Test Runner. http://www.java2s.com/

Para ejecutar este ejemplo desde la terminal basta con primeramente compilar los archivos correspondientes y luego ejecutar el archivo *TestRunner.java.*

```
C:\Users\gaira\OneDrive\Escritorio\JUnitTest>javac TestJunit.java TestRunner.java

C:\Users\gaira\OneDrive\Escritorio\JUnitTest>java TestRunner
true
```

Ilustración 18 Compilación y ejecución del caso de prueba

En este caso la prueba es completada con éxito por lo que se puede observar cómo al ejecutar el test runner se obtiene true, por otro lado, si se cambia alguno de los parámetros en el assert se obtiene este resultado:

```
C:\Users\gaira\OneDrive\Escritorio\JUnitTest>javac TestJunit.java TestRunner.java

C:\Users\gaira\OneDrive\Escritorio\JUnitTest>java TestRunner
test(TestJunit): expected:<H[i]!> but was:<H[ello World]!>
false
```

Ilustración 19 Demostración de un caso de prueba fallido

Sintaxis y elementos básicos

A partir del ejemplo anterior se puede empezar a describir tanto la sintaxis como los elementos básicos que componen una prueba con JUnit y que son necesarios entender.

- **Anotaciones:** En JUnit se utilizan anotaciones en los métodos para darles cierta configuración a los mismos. Algunas de las anotaciones más importantes de JUnit son las siguientes:
    - @Test: Sirve para identificar un método como un método de test.

    - @Before: Se ejecuta antes de cada método test, utilizado para preparar todo antes del test como por ejemplo inicializar clases o variables.
    - @After: Se ejecuta después de cada método test, se utiliza para limpiar el ambiente de pruebas.
    - @BeforeClass: Se ejecuta una única vez antes de todos los tests.
    - @AfterClass: Se ejecuta una única vez después de todos los tests.
    - @Ignore: Sirve para deshabilitar un test.
    - @Test(timeout): Mediante esta anotación se le indica JUnit que si un método tarda más del tiempo esperado debe fallar.
    - @Test(expected): Esta anotación sirve para indicar que el método debe retornar la excepción esperada.

- **Asserts:** Los asserts o afirmaciones son métodos con los cuales se puede verifican las pruebas que se ejecutan se les puede especificar como parámetro el resultado esperado y el real. La documentación oficial de JUnit sobre la clase assert especifican una gran variedad de métodos para llevar a cabo los unit test de los cuales destacan:
    - assertEquals(expected, actual).
    - assertTrue(condition).
    - assertFalse(condition).
    - assertNull(Object).
    - assertNotNull(Object).
- **Test cases:** Corresponde a la clase que contiene el método a probar con este especificado mediante la etiqueta @Test.
- **Test runners:** Se encarga de ejecutar la prueba mediante el test case y el uso de los assert.
- **Test suites:** Mediante el uso de test suites se pueden combinar diferentes test cases mediante la anotación o etiqueta @SuiteClasses donde se especifican todas las clases que contienen los test cases.

Mocha

Mocha es un framework de pruebas de JavaScript que se ejecuta en Node.js. El cual permite la posibilidad de crear tanto tests síncronos como asíncronos con una gran facilidad. (Márquez, 2017).

Algunas características importantes que se pueden destacar de Mocha frente a otras herramientas de testing son las siguientes:

- Diferentes métodos de instalación mediante npm (global o local).
- Soporte para testing asíncrono.
- Soporte para una gran variedad de navegadores.
- Librerías de Assertions o Afirmaciones.

Empezando con Mocha

Instalación

Primeramente, para proceder con la instalación de Mocha, es necesario disponer de node en el equipo, ya que su instalación es mediante npm (Node Package Manager). Posteriormente se puede proceder a instalar Mocha desde la línea de comandos.

Según la documentación oficial de Mocha el tipo de instalación recomendada es global, por lo que el comando a ejecutar es el siguiente:

npm install --global mocha

Finalmente, al ejecutar este comando el entorno de desarrollo estará preparado para ejecutar pruebas unitarias mediante Mocha.

Ejemplo inicial

A continuación, se describe como llevar a cabo una primera prueba unitaria básica utilizando Mocha.

El demo base a utilizado para este ejemplo es el siguiente:

```
var assert = require("assert");

describe("Numbers",function(){
  it('should add two numbers',function(){
    assert.strictEqual(6, 3 + 2);
  })
});
```

Ilustración 20 Demo base. Tomado de: https://solidgeargroup.com/

Para ejecutar la prueba o el *unit test* se debe crear una carpeta *test* dentro de la carpeta del proyecto con sus respectivos archivos *JavaScript.* Luego al ejecutar mocha mediante la terminal, mocha se encarga de buscar la carpeta *test* y correr los archivos automáticamente.

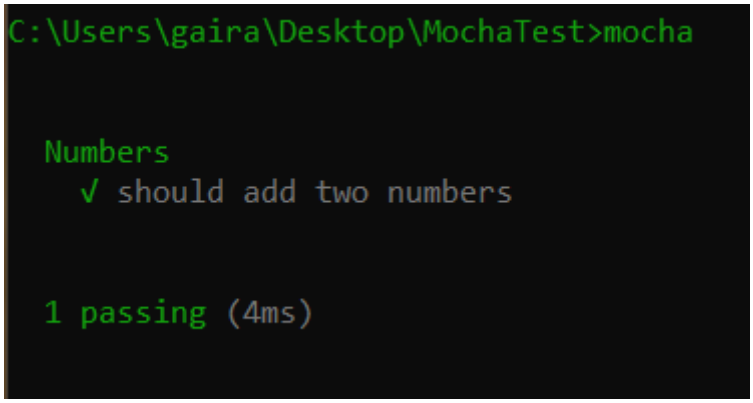

Ilustración 21 Ejecución del caso de prueba.

En este caso se puede observar como la prueba fue completada con éxito, en cambio, si la prueba llegara a fallar se mostraría un resultado como el siguiente:

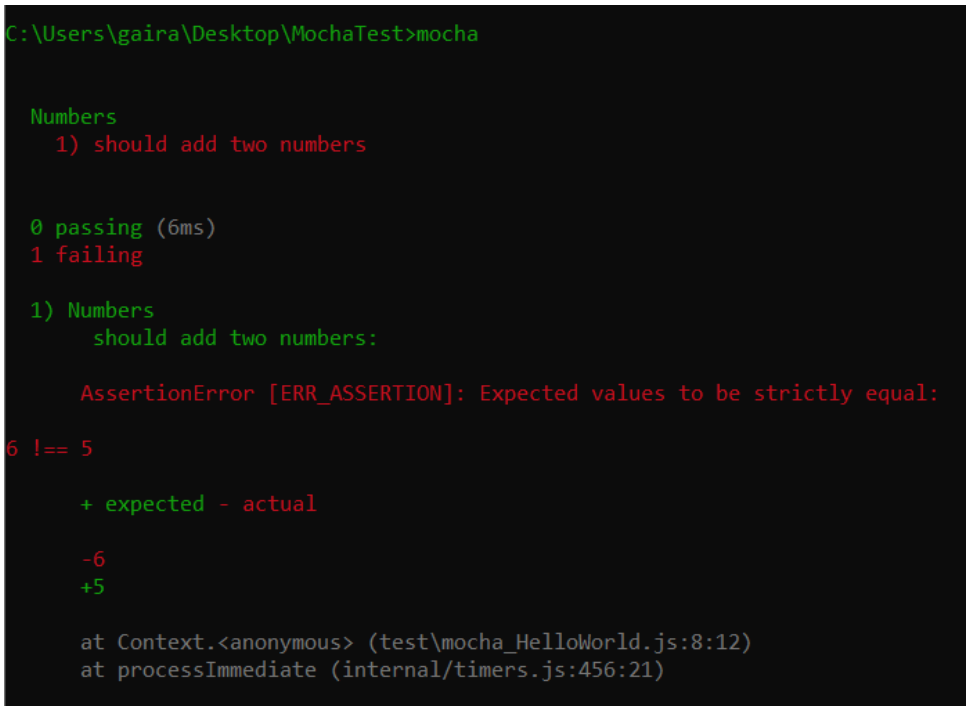

Ilustración 22 Demostración de un caso de prueba fallido

Sintaxis básica

En el ejemplo anterior se puede observar la sintaxis para llevar a cabo una un unit test con Mocha, a continuación, se describe la sintaxis básica utilizada además de otras funciones.

- **assert:** Mediante el módulo assert de Node se llevan a cabo las comprobaciones de la función, en este caso se utiliza el método notStrictEqual(actual, expected[, message]) el cual recibe un valor actual y un valor esperado y comprueba si ambos valores son iguales. Este módulo assert posee otros métodos los cuales pueden ser consultados en

la documentación oficial de Node como por ejemplo assert.fail([message]) o assert.match(string, regexp[, message]).

- **describe:** Esta función es usada para agrupar un conjunto de pruebas similares, aunque inicialmente describe no es necesaria para llevar a cabo las pruebas, según (Especificar), se recomienda para dar un fácil mantenimiento al código.
- **it:** Por último, la función it() es la que contiene el código que se va a probar y utiliza el módulo assert descrito anteriormente.
- **before:** Sirve para ejecutar un bloque de código dentro del método describe antes que cualquier it.
- **after:** Se ejecuta después de realizar los test dentro del describe.

Pruebas asíncronas

Mocha es capaz de llevar a cabo pruebas asincronas tanto utilizando callbacks como done(), utilizando promesas o mediante async/await, esto resulta sumamente útil para manejar la asincronía de JavaScript en pruebas de procesos como escribir en un archivo o insertar en una base de datos.

En la documentación oficial de mocha se puede encontrar un ejemplo básico para cada una de las posibles formas con las que se pueden ejecutar tests asíncronos.

- Callback done: done puede ser utilizada tanto como argumento dentro de la función it() o utilizarse como función done() directamente de esta forma se le puede indicar a Mocha cuando un método se ha ejecutado y esperar a que se ejecuten para terminar las pruebas.

```
describe('User', function() {
  describe('#save()', function() {
    it('should save without error', function(done) {
      var user = new User('Luna');
      user.save(function(err) {
        if (err) done(err);
        else done();
      });
    });
  });
});
```

Ilustración 23 Ejemplo done(). Tomado de: https://mochajs.org/

```
describe('User', function() {
  describe('#save()', function() {
    it('should save without error', function(done) {
      var user = new User('Luna');
      user.save(done);
    });
  });
});
```

Ilustración 24 Ejemplo done. Tomado de: https://mochajs.org/

- Promesas: Este caso se utiliza cuando se debe retornar una promesa. En la documentación oficial se menciona que a partir de la versión 3.0.0 de Mocha no se deben retornar promesas y luego utilizar done ya que esto genera una excepción.

```
const assert = require('assert');


it('should complete this test', function() {
  return new Promise(function(resolve) {
    assert.ok(true);
    resolve();
  });
});
```

Ilustración 25 Ejemplo promise. Tomado de: https://mochajs.org/

- Async/Await: En este ejemplo el test se lleva a cabo utilizando async/await.

```
beforeEach(async function() {
  await db.clear();
  await db.save([tobi, loki, jane]);
});

describe('#find()', function() {
  it('responds with matching records', async function() {
    const users = await db.find({type: 'User'});
    users.should.have.length(3);
  });
});
```

Ilustración 26 Ejemplo async/await. Tomado de: https://mochajs.org/

Conclusión

Durante la investigación realizada para desarrollar este documento, obtuvimos como grupo de trabajo un primer acercamiento al concepto de build y testing, así como una introducción a sus respectivas herramientas más específicamente a Maven, Gradle y Ant para el tema de build y JUnit, Mocha para el tema de testing.

En cuanto al tema de build, aprendimos dos herramientas útiles que permiten el build de proyectos Java y otros lenguaje. Son de gran utilidad, pues permiten compilar nuestros proyectos de forma rápida y sin tener que preocuparnos por el comando de build de java, también gracias a que el build es de forma automática nos descargan desde un repositorio las librerías necesarias para la compilación del software. Maven al principio es un poco complicado y tiene una curva mayor que gradle, que en cambio es más fácil y con menos complicaciones, también el fácil uso de los comandos por medio del CLI que nos quitan la necesidad de un IDE y así poder programar desde un editor de texto.

Para el tema de testing se pudieron estudiar dos herramientas las cuales resultan de gran utilidad para realizar unit test en nuestros proyectos. Por un lado, JUnit se presenta como una herramienta para realizar pruebas en Java, la cual posee un nivel de integración muy alto tanto con las herramientas de build estudiadas y con diferentes IDEs. Además, aunque parezca tener una curva de aprendizaje muy alta, JUnit dispone de una gran cantidad de documentación oficial y gran variedad de ejemplos, por lo que con el tiempo suficiente resulta sencillo dominar la herramienta. En el caso de Mocha, herramienta utilizada para realizar pruebas en JavaScript, esta posee una configuración inicial más sencilla al igual que más facilidad para ejecutar las pruebas. Aun así, cabe recalcar la presencia del concepto de assert o afirmaciones dentro de las dos herramientas estudiadas el cual consideramos como grupo de trabajo un concepto de gran importancia en este tema.

Como comentario final, vimos la gran importancia conocer y saber utilizar este tipo de herramientas las cuales pueden resultar de gran ayuda a la hora de trabajar en un proyecto, facilitando la manera de hacer test y la posibilidad de construir el proyecto sin importar la plataforma y sin la necesidad de usar un IDE.

Referencias

learntek . (4 de junio de 2019). *learntek* . Obtenido de https://www.learntek.org/blog/what-is-maven/

Abuse, S. (15 de Abril de 2020). *Digital Ocean*. Obtenido de https://www.digitalocean.com/community/tutorials/how-to-test-a-node-js-module-with-mocha-and-assert-es

Albing, C., & Schwarz, M. (2005). *Java™ Application Developmenton Linux.* Pearson Education, Inc.

all about testing. (s.f.). Obtenido de http://www.allabouttesting.biz/what-is-mocha-js/

Apache Ant - Welcome. (s.f.). Obtenido de Apache Ant: https://ant.apache.org/

Apache. (s.f.). *Maven* . Obtenido de Introduction to Repositories: https://maven.apache.org/guides/introduction/introduction-to-repositories.html

atlassian. (s.f.). *atlassian*. Obtenido de https://www.atlassian.com/continuous-delivery/continuous-integration

Berglund, T., & McCullough, M. (2011). *Building and Testing with Gradle: Understanding Next-Generation Builds.* O'Reilly Media, Inc.

Bharathan, R. (2015). *Apache Maven Cookbook.* Birmingham : Packt Publishing Ltd.

Garzas, J. (s.f.). *Jarvier Garzas.com*. Obtenido de https://www.javiergarzas.com/2016/03/tdd-nodejs-mocha-parte-1-entendiendo-conceptos.html

howtodoinjava. (s.f.). *howtodoinjava*. Obtenido de https://howtodoinjava.com/maven/maven-dependency-management/

Java JUnit Tutorial - JUnit Introduction. (s.f.). Obtenido de Java2s: http://www.java2s.com/Tutorials/Java/JUnit/index.htm

Java T point. (s.f.). Obtenido de https://www.javatpoint.com/software-testing-tools

Java T Point. (s.f.). Obtenido de https://www.javatpoint.com/junit-tutorial

Jet Brains. (27 de Octubre de 2020). Obtenido de https://www.jetbrains.com/help/idea/testing.html

K, A. (12 de 12 de 2018). *linuxnix*. Obtenido de https://www.linuxnix.com/what-is-maven-and-what-are-its-benefits/

Márquez, R. (15 de 2 de 2017). *Paradigma*. Obtenido de https://www.paradigmadigital.com/dev/testeando-javascript-mocha-chai/

Maven: The Complete ReferenceiMaven: The Complete Reference. (s.f.). Sonatype, Inc.

Mocha. (s.f.). Obtenido de Mocha: https://mochajs.org/

MochaJS. (s.f.). Obtenido de https://mochajs.org/

Turrado, J. (3 de julio de 2019). *Campus mvp*. Obtenido de https://www.campusmvp.es/recursos/post/integracion-continua-que-es-y-por-que-deberias-aprender-a-utilizarla-cuanto-antes.aspx

Verma, R. (s.f.). *scmquest*. Obtenido de https://scmquest.com/software-build-knowledge/

Vicente, D. (s.f.). *Solid Gear*. Obtenido de https://solidgeargroup.com/mocha-unit-tests-en-javascript/

Vogel, L. (21 de Julio de 2016). *Vogella*. Obtenido de https://www.vogella.com/tutorials/JUnit/article.html