# Apunte 15 - Testing y Mocking

## Introducción General

Una vez desarrollada una parte de un programa, o a veces inclusive antes del desarrollo, llega el momento de verificar que todo funciona como se supone que debería funcionar. Para ello se realizan una serie de pruebas, donde el programador ejecuta una parte del programa, conociendo de antemano los resultados que deberían darse y verifica que efectivamente se estén entregando esos resultados. Estas pruebas de partes del programa se conocen como pruebas unitarias, o tests unitarios.

Si bien esto puede hacerse en forma manual para programas chicos, llega el punto en el que uno quiere automatizar dichos tests, para poder ejecutarlos múltiples veces al ir haciendo cambios al programa. Eso garantiza, o ayuda a evitar que de forma accidental el programador introduzca un cambio que termine rompiendo o modificando la funcionalidad existente de forma no esperada.

El ecosistema Java posee varias herramientas pensadas para realizar este tipo de tests, y la mas común es la utilización de una librería llamada *JUnit*. Dicha librería nos provee una serie de anotaciones y clases que uno puede usar para generar las pruebas unitarias. Esta librería fue una de las primeras dedicadas a la automatización de pruebas, y fue creada por dos proponentes del uso de testing automatizado como son *Kent Beck* y *Erich Gamma*.

Una vez creados los casos de prueba estos se pueden ejecutar desde el propio entorno de desarrollo, sea IntelliJ u otro, ya que casi todos tienen soporte para la ejecución de pruebas unitarias. Además se pueden correr desde el propio *Maven* como parte del proceso de empaquetado del programa.

## Temario

- Introducción a *JUnit*
- Como importar JUnit en un proyecto Maven.
- Creación de un unit test
- Comprobaciones
- Ejecución de tests
- Mocking
- Uso de *mockito* para generar mocksi
- Pruebas usando postman
- Pruebas usando cURL

## Introducción a JUnit

JUnit es una librería que nos provee herramientas para la ejecución y creación de pruebas unitarias. Esta librería es una de las mas usadas para la generación de pruebas unitarias y actualmente se encuentra en la versión 5 (Al momento de escribir esto la versión 5.10)

Particularmente la versión 5 de JUnit se compone de tres componentes:

- *JUnit Platform*: Que es la plataforma que permite el descubrimiento y ejecución de las pruebas unitarias. Esta plataforma se encuentra integrada en casi todos los IDE de java y en las herramientas de construcción como *Maven* o *Gradle*
- *JUnit Jupiter*: Es un modelo de programación que nos permite escribir los tests y provee una serie de anotaciones y extensiones que ayudan a la escritura de los tests.
- *JUnit Vintage*: Es una capa de compatibilidad para poder seguir usando tests escritos para JUnit4 en proyectos que usan JUnit 5.

Para el caso de un proyecto que arranque con JUnit 5 no es necesario usar el JUnit Vintage, y la mayor parte del esfuerzo se va a centrar en la escritura de los tests usando el Junit Jupiter, que no es mas que un conjunto de anotaciones y clases a usar al momento de escribir una prueba unitaria.

## Como importar JUnit en un proyecto Maven

Para poder utilizar JUnit en un proyecto maven, basta con agregar una dependencia al artefacto con groupId *org.junit.jupiter* y artifactId *junit.jupiter*. En general al referirse a una dependencia de maven se suele usar la forma groupId:artifactId, o sea para este caso la dependencia necesaria sería *org.junit.jupiter:junit.jupiter*

Dicha dependencia se agrega en la sección *<dependencies>* del archivo pom.xml de la siguiente forma:

```
    <dependencies>
        <dependency>
            <groupId>org.junit.jupiter</groupId>
            <artifactId>junit-jupiter</artifactId>
            <version>5.10.0</version>
            <scope>test</scope>
        </dependency>
    </dependencies>
```

Agregando esta dependencia se agrega transitivamente la dependencia de junit-engine necesaria para la ejecución de los tests.

Nótese la inclusión del tag **scope** dentro de la definición de la dependencia. Esto le indica a Maven que esta dependencia será usada durante el proceso de testing, pero no será incluida dentro del artefacto generado en el packaging del proyecto.

## Creación de un unit test

Lo primero que hay que ver al momento de crear un unit test, es donde se deben ubicar los archivos del test. Por convención de Maven, dentro de la carpeta src hay dos carpetas, **main** y **test**. La carpeta main es donde van a estar todos los archivos de código fuente que forman parte del código de producción, o sea el que se va a terminar empaquetando con el proyecto. La carpeta test está para que en ella se ubiquen todos los test unitarios, estos fuentes sólo se compilan durante la ejecución de los tests, pero no forman parte del producto.

Dentro de JUnit un unit test es básicamente un método anotado con la anotación **@Test**, y los tests se agrupan en Suites, que son clases cuyo nombre generalmente termina en *Test* y contienen uno o más métodos anotados con la anotación **@Test**.

```
public class EjemploTest {

    public int suma(int a, int b) {
        return a+b;
    }

    @Test
    public void testSuma() {
        int suma = suma(1, 3);
        Assertions.assertEquals(4, suma);
    }

}
```

En el ejemplo previo, el método testSuma va a invocar al método suma y comprobar el resultado devuelto por la función suma contra un valor esperado.

Además de @Test hay varias anotaciones mas que se pueden usar para marcar métodos, algunas de ellas son:

- **@BeforeEach**: Marca un método que se va a ejecutar antes de la ejecución de cada uno de los tests de la suite
- **@BeforeAll**:Marca un método que se va a ejecutar antes de la ejecución de todos los tests de la suite (sólo se ejecuta una vez por suite)
- **@AfterEach**: Análogo a BeforeEach, pero se ejecuta luego de cada test
- **@AfterAll**: Igualmente análogo a BeforeAll, sólo se va a ejecutar una vez al terminar de ejecutar todos los tests de la suite
- **@Disabled**: Permite desactivar un test, para que no se ejecute
- **@Timeout**: Permite establecer un tiempo máximo de ejecución para un test, si este tiempo se excede el test falla.

## Comprobaciones

Dentro de un test, se espera que además de ejecutar algo y ver que no se produzcan excepciones, se compruebe que los resultados provistos por el código sean los correctos. Esto se hace mediante comprobaciones, conocidas como *assertions*. Si la comprobación es correcta, sigue la ejecución, pero si no es correcta se interrumpe la ejecución del test con un error.

Dentro de JUnit 5 las comprobaciones están dentro de la clase Assertions. Algunos de los métodos provistos por esta clase son:

- **assertTrue(boolean)***: Comprueba que el boolean sea true, o falla
- **assertTrue(boolean, mensaje)**: Comprueba que el boolean sea true o falla informando el mensaje indicado
- **assertFalse(boolean)**: Comprueba que el boolean sea false, o falla
- **assertFalse(boolean, mensaje)**: Comprueba que el boolean sea false o falla informando el mensaje indicado
- **assertEquals(esperado, valor)**: Comprueba que valor sea igual a esperado o falla. Este método tiene muchas variantes con diferentes tipos, notable de destacar es la variante de float que admite un tercer

parámetro indicando un delta que tiene que superarse para que se considere que los valores no son iguales.
- **assertEquals(esperado, valor, mensaje)**: Igual que el anterior, pero agregando el mensaje a mostrar en caso de falla.
- **assertNotEquals(esperado, valor)**: Contrario a assertEquals
- **assertNotEquals(esperado, valor, mensaje)**: Contrario a assertEquals
- fail()**: hace fallar el test
- fail(mensaje)**: hace fallar el test informando un mensaje
- **assertThows(clase, ejecutable)**: Comprueba que el código ejecutado en *ejecutable*, que es una interfaz funcional donde se puede usar un lambda, lance una Exception del tipo *clase*
- **assertDoesNotThrow(ejecutable)**: Comprueba que el código ejecutado en *ejecutable* NO lance una exception.

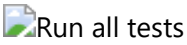
Todos los métodos definidos en Assertions son estáticos, con lo cual se pueden usar de la forma *Assertions.assertEquals(a, b)*, o también se suele hacer de forma habitual un import static para que todos los métodos assert estén disponibles para llamar directamente.

```
import static org.junit.jupiter.api.Assertions.assertTrue;
```

Haciendo este import se puede usar por ejemplo *assertEquals* como si estuviera definida dentro del test.

## Ejecución de tests

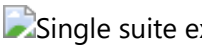
Para ejecutar los tests desde IntelliJ la forma mas simple es mediante el uso del botón derecho sobre el arbol de proyecto, y seleccionando la opción **Run 'All Tests'**

 Run all tests

Esto va a ejecutar todas las suites de tests presentes en el proyecto y luego se va a mostrar el resultado de dicha ejecución, marcando los test que se ejecutaron exitosamente y los que no.

 Test Results

Alternativamente se puede ejecutar una sola suite de test, o un test individual haciendo click en la flecha verde que presenta IntelliJ al abrir el código de dicha suite de test.

 Single suite execution

Otra forma de ejecutar los tests, es mediante el uso de maven. Dentro de los diferentes pasos del ciclo de vida que provee maven, existe uno llamado **test** que se puede invocar para ejecutar los test

Este paso del ciclo de vida también se termina ejecutando al realizar otras acciones con maven como pueden ser **deploy**, **package** o **install**.

```
18:50 $ mvn test
[INFO] Scanning for projects...
[INFO]
[INFO] -------------------< ar.edu.utn.frc.bso:introJunit >--------------------
```

```
[INFO] Building introJunit 1.0-SNAPSHOT
[INFO]   from pom.xml
[INFO] --------------------------------[ jar ]---------------------------------
[INFO]
[INFO] --- resources:3.3.0:resources (default-resources) @ introJunit ---
[INFO] Copying 0 resource
[INFO]
[INFO] --- compiler:3.10.1:compile (default-compile) @ introJunit ---
[INFO] Nothing to compile - all classes are up to date
[INFO]
[INFO] --- resources:3.3.0:testResources (default-testResources) @ introJunit ---
[INFO] skip non existing resourceDirectory
/home/fbett/UTN/2023/Backend/material/semana-
07/testing/introJunit/introJunit/src/test/resources
[INFO]
[INFO] --- compiler:3.10.1:testCompile (default-testCompile) @ introJunit ---
[INFO] Nothing to compile - all classes are up to date
[INFO]
[INFO] --- surefire:3.0.0:test (default-test) @ introJunit ---
[INFO] Using auto detected provider
org.apache.maven.surefire.junitplatform.JUnitPlatformProvider
[INFO]
[INFO] -------------------------------------------------------
[INFO]  T E S T S
[INFO] -------------------------------------------------------
[INFO] Running ar.edu.utn.frc.bso.EjemploTest
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.043 s -
in ar.edu.utn.frc.bso.EjemploTest
[INFO] Running ar.edu.utn.frc.bso.ListaEnArrayTest
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.006 s -
in ar.edu.utn.frc.bso.ListaEnArrayTest
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  1.300 s
[INFO] Finished at: 2023-10-01T18:50:48-03:00
[INFO] ------------------------------------------------------------------------
[WARNING]
[WARNING] Plugin validation issues were detected in 2 plugin(s)
[WARNING]
[WARNING]  * org.apache.maven.plugins:maven-compiler-plugin:3.10.1
[WARNING]  * org.apache.maven.plugins:maven-resources-plugin:3.3.0
[WARNING]
[WARNING] For more or less details, use 'maven.plugin.validation' property with
one of the values (case insensitive): [BRIEF, DEFAULT, VERBOSE]
[WARNING]
```

# Mocking

Durante el desarrollo de pruebas unitarias, es normal que en algún punto tengamos que hacer que algún objeto devuelva un valor predefinido, esperado por el test.

Por ejemplo, al hacer tests a un servicio que usa un repositorio, va a ser necesario, para poder probar algunas cosas, hacer que ese repositorio devuelva cierto valor en concreto. Esto se logra implementando una versión del repositorio que devuelve valores fijos, o configurables, y usando dicha versión al instanciar el servicio en el test.

Por ejemplo dado este servicio de ejemplo:

```
package ar.edu.utn.frc.bso;

import java.util.List;
import java.util.Optional;

public class ServicioAlumnos {

    private RepositorioAlumnos repositorio;

    public ServicioAlumnos(RepositorioAlumnos repositorio) {
        this.repositorio = repositorio;
    }

    public Alumno obtenerAlumno(int legajo) {
        List<Alumno> lista = repositorio.listar();
        for(Alumno a: lista) {
            if (a.getLegajo() == legajo) {
                return a;
            }
        }
        return null;
    }
}
```

Uno podría intentar hacer dos tests para el método *obtenerAlumno*. Uno podría probar el caso en el que se encuentra el alumno, y uno el caso de que no se encuentre. Entonces, para ello se podría hacer una clase que herede de *RepositorioAlumnos* y agregar métodos para definir la lista de alumnos a devolver, para que cada test defina de antemano lo que debería devolver la llamada a listar() del repositorio.

Esto quedaría así:

```
package ar.edu.utn.frc.bso;

import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
```

```
import java.util.List;

import static org.junit.jupiter.api.Assertions.*;


class RepositorioTest extends RepositorioAlumnos {
    private List<Alumno> listaADevolver;

    public void setListaADevolver(List<Alumno> l) {
        this.listaADevolver = l;
    }

    @Override
    public List<Alumno> listar() {
        return listaADevolver;
    }
}
public class ServicioAlumnosTest {

    private ServicioAlumnos servicio;
    private RepositorioTest repositorioTest;

    @BeforeEach
    public void setup() {
        repositorioTest = new RepositorioTest();
        servicio = new ServicioAlumnos(repositorioTest);
    }

    @Test
    public void testAlumnoExistente() {
        Alumno alumnoEsperado = new Alumno("Pepe", 123);
        repositorioTest.setListaADevolver(List.of(alumnoEsperado));
        Alumno x = servicio.obtenerAlumno(123);
        Assertions.assertEquals(alumnoEsperado, x);
    }

    @Test
    public void testAlumnoNoExistente() {
        Alumno alumnoEsperado = new Alumno("Laura", 234);
        repositorioTest.setListaADevolver(List.of(alumnoEsperado));
        Alumno x = servicio.obtenerAlumno(123);
        Assertions.assertNull(x);
    }

}
```

La clase de test del repositorio hereda de la clase real (o implementa la interfaz si hubiera una), y permite establecer la lista de alumnos a devolver. Con esta funcionalidad cada test define la lista previamente a llamar al servicio, con lo cual se puede predecir que debería devolverse en cada caso.

## Uso de *mockito* para generar mocks

Si bien la generación se puede hacer manualmente como en el ejemplo previo, existen librerías que nos permiten hacer lo mismo, pero en forma automática sin tener que escribir tanto código a mano.

Una de dichas librerías es Mockito. Esta librería permite generar, a partir de la clase un objeto llamado *mock* donde se pueden establecer los resultados que debe devolver cada llamada a un método de dicho mock.

Para usar esta librería, es necesario agregarla como dependencia, de la siguiente forma:

```
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-core</artifactId>
    <version>5.5.0</version>
    <scope>test</scope>
</dependency>
```

Luego, en una clase de test se puede generar un mock usando el método *Mockito.mock*, y el objeto devuelto permite la verificación de llamadas, o predefinir los resultados que deben devolverse al llamar un método

```
List mockedList = mock(List.class);

mockedList.add("one");
mockedList.clear();

verify(mockedList).add("one");
verify(mockedList).clear();

when(mockedList.get(0)).thenReturn("first");
System.out.println(mockedList.get(0));
```

En el código previo se pueden apreciar los dos usos de los mocks:

- La verificación, donde se comprueba que se haya llamado al método add y el método clear
- La definición de resultados (conocida como *stubbing*), donde en el ejemplo se define que al llamar a get(0), se va a devolver la cadena "first"

Usando esto, podemos modificar el test previo de la siguiente forma:

```
package ar.edu.utn.frc.bso;

import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.mockito.Mockito;

import java.util.List;
```

```java
public class ServicioAlumnosTestMockito {

    private ServicioAlumnos servicio;
    private RepositorioAlumnos repositorio;

    @BeforeEach
    public void setup() {
        repositorio = Mockito.mock(RepositorioAlumnos.class);
        servicio = new ServicioAlumnos(repositorio);
    }

    @Test
    public void testAlumnoExistente() {
        Alumno alumnoEsperado = new Alumno("Pepe", 123);
        Mockito.when(repositorio.listar()).thenReturn(List.of(alumnoEsperado));
        Alumno x = servicio.obtenerAlumno(123);
        Assertions.assertEquals(alumnoEsperado, x);
    }

    @Test
    public void testAlumnoNoExistente() {
        Alumno alumnoEsperado = new Alumno("Laura", 234);
        Mockito.when(repositorio.listar()).thenReturn(List.of(alumnoEsperado));
        Alumno x = servicio.obtenerAlumno(123);
        Assertions.assertNull(x);
    }

}
```

## Pruebas usando Postman

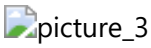
Si bien las pruebas unitarias son una herramienta muy potente en nuestros procesos de desarrollos, ya que como se comentó anteriormente asegura la calidad de una mínima unidad de codigo (método de clase), aislándola y comprobando su funcionamiento (asserts), muchas veces se hace necesario comprobar no solo funcione esa unidad sino que funcione correctamente nuestra aplicación o api desde que se captura una request de un endpoint hasta que dicho endpoint retorna una response con la información solicitada.

Para ese propósito existen muchas herramientas que nos permiten asegurar el correcto funcionamiento de nuestra apicación, siendo las mas difundidas herramientas embebidas como Swagger pero tambien herramientas externas como las que vamos a analizar en esta sección que es Postman. Sin mucho mayor detalle, el objetivo de Postman es permitir realizar pruebas de nuestras APIs llamando sus diferentes endpoint a través de las URI que fueron definidos para ellos, dicho de otra manera mas sencilla Postman lo que busca es emular como funcionaría una request a nuestro endpoint desde una aplicacion Front End usando el protocolo Http.

La aplicación de Postman va más alla, no solo nos permite testear nuestras APIs, tambien documentarlas y si deseamos podremos generar nuestros ambientes de prueba, generando la colecciones de endpoints y las variables generales que se desean aplicar, por supuesto observando las distinitas opciones de seguridad que se le pueden aplicara nuestras API. Para descargar la aplicación usted lo puede hacer desde aqui, una vez descargada al apliación tendria una estructura como se presenta en la siguiente imagen

Si desglozamos esta imagen la aplicación presenta una serie de paneles y pestañas, los mas importantes son:

- El Panel de izquierda *My Workspace* alli definiremos nuestras Colecciones y Ambientes principalmente. Las Colcciones podemos verlos como sub carpetas que tendrán los endpoints que deseemos testear, dando la herramienta la flexibilidad de agrupar los mismos en la/s colecciones que deseemos, como se muestra en la imagen

- A la derecha el panel que tenemos es el panel principal de trabajo que nos va a permitir realizar nuestras pruebas. Para poder activarlo solo debemos agregar nu nueva request a nuestra colección, abierto ese panel podremos elegir el verbo del protocolo Rest que tienen muestro endpoint, y luego asociamos nuestra URI en campo siguiente

Una vez establecidos esos valores, siguiente el ejemplo del proyecto clase-spring, tendríamos una prueba a un endpoint marcado como GET cuya URI seria *http://localhost:8080/api/task/1*. A partir de este punto inicial podemos comenzar a personalizar o utilizar cada uno de los tabs que estan en la parte inferior del panel principal, una descripción breve de cada uno de ellos sería:

- **Params** Permite establecer los parámetros de query string que viajaría con nuestro request, en la grilla se obtiene una configuración key (nombre del parámetro de querystring) value (que valor se le asigna) (Por ejemplo nombre=German), se pueden agregar tantos como se definan en nuestra API y la herramienta los va a parsear en forma correcta key1=value1&key2=value2&...&keyN=valueN.

- **Auth** Esta sección nos va a permitir colocar los tokens de autenticación que requiera nuestra API, si la misma es básica y no tiene autenticación va vacío, cabe aclarar que esto no se recomienda, todas las API deberían tener seguridad

- **Headers** Esta sección es importante, toda request o response tienen un header, que seria la metadata de la llamada, en cuanto a como debe ser la request que se hace, por defecto hay 6 hidden que son las mínimas requeridas para que nuestra llamada no devuelva un HttpError Bad Request. De los headers que podemos usar el mas importante es el *Content-Type*, que define como vamos a enviar la información o recibirla, siendo el mas usado application/json o application/octet-stream (usado para transferir archivos), la lista es larga y queda en uds investigar mas en profundidad.

- **Body** Sección que usaremos para colocar lo que se llama payload,o lo que enviamos con el BodyRequest, el objeto JSON que queremos usar en nuestros verbos POST, PUT, PATCH, para los demás va vacío, de lo contrario genera un error.

- Las demás pestañas son mas de especialización de tareas, de ellas es particularmente importante la de Settings que permite establecer las configuración de como se llamará a ese endpoint, ya sea si activa certificados SSL por ejemplo o el encode de la url y otras tantas mas que se pueden utilizar dependiendo del caso.

Una vez configurado nuestro endpoint solo resta apretar Send y si nuestra API esta desplegada en un servidor cloud o bien corriendo localmente, como es el caso de este ejemplo, Postman tratara de enviar el request y si

es exitoso o no, nos lo informará con el código de Http y la información, como muestra la siguiente imagen.


Con lo expuesto anteriormente es mas que suficiente para probar nuestras APIs, pero esta herramienta nos permite establecer *Environments*, y ¿qué serían dichos ambientes? Pues ellos nos permiten establecer variables globales que después podemos usar en nuestras colecciones, valores que sabemos que no van a variar permanentemente mientras probemos, como puede ser un token de autorización que se renueva cada cierto tiempo, o bien el nombre del usuario que ese token valida, o diferentes valores que consideramos generales y necesarios para nuestras pruebas como podría ser también rutas de subidas de archivos, etc. Claramente las variables tienen tipos y alcances que son descritos en la documentación, una vez declaradas las podremos usar con la siguiente notación **{{nombre_variable}}**. Al crearlas se le puede asignar un valor inicial, aunque luego se puede asignar valores a traves de la escritura de tests (estos temas son avanzados y queda a criterio del alumno ver que tan profundo recorre ese camino)

## Pruebas usando cURL

Otra forma de probar nuestras APIs es mediante lo que se llama “Client for URLs” o “Curl URL Request Library” o como se lo conoce cURL, el cual es un programa dirigido por lineas de comandos con su correspondiente librería para la transferencia de información. Los comandos que se ejecutarán tendrán las siguiente estructura:

```
> C:\Users\user>curl.exe [options …] <url>
```

Y si para poder usar esta aplicación se debe descargar para el sistema operativo en donde se desee utilizar, ya que es una herramienta desarrollada en un lenguaje C. Para saber si la aplicación esta instalada solo basta con ejecutar este comando

```
> C:\Users\user>curl.exe --version
```

Si no fue exitoso el comando dará un error de este estilo *curl : The remote name could not be resolved:* '*--version*' de lo contrario nos presentara la versión instalada. Para probar tan solo en una consola se pondría el siguiente comando

```
> curl --location http://localhost:8080/api/task/1
```

y se obtendrá el resultado de esa llamada a una operación como muestra la imagen

Si lo que deseamos es testear un endpoint el cual llega un payload en el Request Body, como puede ser y de hecho sera en la mayoría de las veces un JSON seria de la siguiente manera

```
> curl --header "Content-Type: application/json" \
    --request POST \
    --data '{"description": "Entregar TP","scheduleDate": "2023-09-
```

```
29T12:34:37.275Z"}' \
        http://localhost:8080/api/task
```

El resultado que retornaría sera el Http Status 201 Created o 204 No Content, dependiendo de que código se este retornando en la API

## Proyectos de Ejemplo

- IntroJunit Proyecto con ejemplos de unit tests

## Software requerido

- JUnit
- Maven
- Mockito
- Postman

## Puntos de continuación

Una cosa que es interesante ver, teniendo en mente la idea de testing unitario, es una metodología de desarrollo que se basa en realizar los test antes que el código de producción. A esta metodología se la conoce como TDD (Test Driven Development)

## Bibliografía


- Junit User Guide
- Mockito documentation
- Postman documentation
- cURL


## Autor/es y agradecimientos

- Ing. Federico Bett - Ing. Germán Romani - Cátedra de Backend de Aplicaciones