Arquitectura de Software
Universidad de Antioquia

Laboratorio. Implementación de un entorno colaborativo de integración continua con Spring Boot, GitHub Actions, y SonarCloud

Elaborado por: Diego José Luis Botia – Profesor Departamento Ingeniería de Sistemas.

Introducción:

En el siguiente laboratorio, se aprenderá a construir un esquema de integración continua y despliegues continuos (CI /CD) para una aplicación realizada en SpringBoot, por medio del enlace de un repositorio en **GitHub** y enlazado a la plataforma **GitHub Actions**. También se aprenderá a realizar pruebas de cobertura del código a través de las plataformas **SonarCloud** y **JaCoCo**. Por último, se hará un test de seguridad con la plataforma **Snyk.**

CI/CD es un enfoque que aumenta la frecuencia de distribución de aplicaciones mediante la introducción de la automatización en la fase de desarrollo de aplicaciones.

Los principales conceptos relacionados con el enfoque CI/CD son la integración continua, la distribución continua y el despliegue continuo. El CI/CD representa una solución a los problemas que plantea la integración de nuevos segmentos de código para los equipos de desarrollo y operaciones (el llamado "infierno de la integración").

En concreto, el CI/CD garantiza la automatización y monitorización continuas a lo largo del ciclo de vida de la aplicación, desde las fases de integración y pruebas hasta la distribución y despliegue. Juntas, estas prácticas suelen denominarse pipeline CI/CD, y se basan en la colaboración ágil entre los equipos de desarrollo y operaciones.

Teoría Básica de los pipelines de CI/CD y su proceso de automatización

1. ¿Qué es CI/CD?

CI/CD es un conjunto de prácticas que combina la **Integración Continua (CI)**, la **Entrega Continua (CD)** y, en algunos casos, el **Despliegue Continuo**. Estas prácticas buscan reducir los problemas asociados con la integración de código, acelerar la entrega de software y garantizar su calidad mediante la automatización.

1.1. Integración Continua (CI)

La Integración Continua se centra en automatizar y mejorar la integración del código desarrollado por múltiples colaboradores. Los desarrolladores trabajan en diferentes partes de una aplicación, y sus cambios deben integrarse frecuentemente en un repositorio central (Por ejemplo, GitHub, GitLab).

- **Prácticas clave**:
  - Los desarrolladores suben (commit) su código al repositorio varias veces al día.
  - Cada cambio desencadena una ejecución automática de pruebas (unitarias, de integración, etc.).
  - Se construye el software (ejemplo., compilación de un JAR en Java) para detectar errores rápidamente.
- **Beneficios**:
  - Detección temprana de errores.
  - Reducción del "infierno de la integración" (conflictos al combinar código).
  - Mayor colaboración entre equipos.

1.2. Entrega Continua (CD)

La Entrega Continua extiende la CI al asegurar que el software esté siempre listo para ser desplegado en un entorno de producción o preproducción. Los cambios que pasan las pruebas se empaquetan y están disponibles para su despliegue, pero el despliegue final suele requerir aprobación manual.

- **Prácticas clave**:
  - Automatización de la construcción de artefactos (ejemplo, imágenes de Docker, JARs).
  - Ejecución de pruebas avanzadas (ejemplo, pruebas de carga, seguridad).
  - Preparación de entornos de staging para validar el software.

- **Beneficios**:
  - Mayor confianza en la calidad del software.
  - Despliegues más predecibles y controlados.

1.3. Despliegue Continuo

El Despliegue Continuo lleva la entrega continua un paso más allá al automatizar completamente el despliegue a producción. Cada cambio que pasa las pruebas se despliega automáticamente sin intervención manual.

- **Prácticas clave**:
  - Despliegue automático a producción tras pruebas exitosas.
  - Monitoreo continuo post-despliegue.
- **Beneficios**:
  - Lanzamientos más frecuentes y rápidos.
  - Retroalimentación inmediata sobre el comportamiento en producción.

2. Componentes de un Pipeline CI/CD

Un **pipeline CI/CD** es una secuencia automatizada de pasos que procesa el código desde el desarrollo hasta el despliegue. Los componentes principales incluyen:

1. **Repositorio de código**:
   - Herramientas como GitHub, GitLab o Bitbucket almacenan el código fuente.
   - Los desarrolladores suben cambios a través de commits y ramas.
2. **Construcción**:
   - Compila el código (por ejemplo, **mvn package** para Java) y genera artefactos (JAR, WAR, imagen de Docker).
   - Ejemplo: Construir un JAR de una aplicación Spring Boot.
3. **Pruebas**:
   - Ejecuta **pruebas** automatizadas:
     - **Unitarias**: Validan funciones individuales (ejemplo, JUnit).
     - **Integración**: Verifican la interacción entre componentes.
     - **Cobertura**: Mide el porcentaje de código probado (ejemplo., JaCoCo).
   - Herramientas: JUnit, TestNG, Selenium.
4. **Análisis de calidad**:
   - Evalúa la calidad del código mediante análisis estático.
   - Herramientas: SonarCloud, Snyk (para detectar vulnerabilidades).
5. **Despliegue**:
   - Empaqueta el artefacto (ejemplo, en una imagen de Docker) y lo despliega a entornos como staging o producción.
   - Ejemplo: Subir una imagen a Docker Hub y desplegarla en AWS ECS.
6. **Monitoreo**:
   - Supervisa la aplicación en producción para detectar errores o problemas de rendimiento.
   - Herramientas: Prometheus, Grafana, AWS CloudWatch.

3. Automatización del Pipeline CI/CD

La automatización es el núcleo de CI/CD, ya que elimina tareas manuales repetitivas y reduce errores humanos. Un pipeline automatizado se configura típicamente con herramientas como **GitHub Actions**, **Jenkins**, **GitLab CI/CD** o **CircleCI**.

3.1. Beneficios de la automatización

- **Eficiencia**: Reduce el tiempo necesario para probar y desplegar.
- **Consistencia**: Garantiza que los pasos se ejecuten de la misma manera cada vez.
- **Escalabilidad**: Soporta equipos grandes y proyectos complejos.
- **Transparencia**: Proporciona registros claros de cada ejecución (logs).

3.2. Estructura de un pipeline automatizado

Un pipeline típico en una herramienta como GitHub Actions incluye:

1. **Desencadenadores (Triggers)**:
   - Eventos que inician el pipeline, como un **push** a la rama main o un **pull request**.

- Ejemplo:

```
on:
  push:
    branches:
      - main
```

2. **Trabajos (Jobs)**:
   - Conjuntos de pasos que se ejecutan en un entorno específico (ejemplo, ubuntu-latest).
   - Ejemplo: Un trabajo para pruebas y otro para despliegue.
3. **Pasos (Steps)**:
   - Acciones específicas, como clonar el repositorio, configurar un entorno, ejecutar pruebas o desplegar.
   - Ejemplo:

```
steps:
  - uses: actions/checkout@v1
  - name: Run Tests
    run: mvn -B test
```

4. **Dependencias**:
   - Define el orden de ejecución entre trabajos (ejemplo., el despliegue depende de pruebas exitosas).
   - Ejemplo:

```
deploy:
  needs: tests
```

3.3. Herramientas para automatización

- **GitHub Actions**: Automatiza flujos de trabajo directamente desde el repositorio de GitHub.
- **Jenkins**: Servidor de automatización flexible para pipelines personalizados.
- **GitLab CI/CD**: Integrado con GitLab para flujos de trabajo nativos.
- **CircleCI**: Plataforma basada en la nube para pipelines rápidos.
- **Herramientas complementarias**:
  - **SonarCloud**: Análisis de calidad de código.
  - **Snyk**: Detección de vulnerabilidades.
  - **JaCoCo**: Medición de cobertura de pruebas.
  - **Docker**: Empaquetado de aplicaciones en contenedores.

3.4. Ejemplo de pipeline automatizado

Un pipeline simple para una aplicación Spring Boot usando GitHub Actions podría incluir:

```
name: CI/CD Pipeline
on:
  push:
    branches:
      - main
jobs:
  tests:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v1
      - name: Set up JDK
        uses: actions/setup-java@v1
        with:
          java-version: '11'
      - name: Run Tests
        run: mvn -B test
  build:
    needs: tests
```

```
runs-on: ubuntu-latest
steps:
  - uses: actions/checkout@v1
  - name: Build with Maven
    run: mvn -B package -DskipTests
  - name: Build & Push Docker
    run: |
      docker build -t dbotiaudea/gitlab:latest .
      docker push dbotiaudea/gitlab:latest
```

Este pipeline:

- Ejecuta pruebas unitarias.
- Construye un JAR.
- Crea y sube una imagen de Docker a Docker Hub.

4. Mejores prácticas para CI/CD

1. **Commits pequeños y frecuentes**: Facilita la integración y detección de errores.
2. **Pruebas exhaustivas**: Incluye pruebas unitarias, de integración y de seguridad.
3. **Automatización completa**: Minimiza la intervención manual para reducir errores.
4. **Entornos consistentes**: Usa contenedores (Docker o Podman) para garantizar que los entornos de desarrollo, pruebas y producción sean idénticos.
5. **Monitoreo y retroalimentación**: Implementa monitoreo para detectar problemas tras el despliegue.
6. **Seguridad**: Escanea vulnerabilidades en el código y dependencias (ejemplo Snyk).
7. **Documentación**: Mantén un README.md claro con instrucciones para ejecutar y desplegar la aplicación además de aplicar Markdowns.

5. Desafíos comunes y soluciones

- **Fallas en pruebas**:
  - **Problema**: Pruebas inestables o falsos positivos.
  - **Solución**: Mejora la calidad de las pruebas y usa herramientas de cobertura como JaCoCo.
- **Tiempos largos de ejecución**:
  - **Problema**: Pipelines lentos retrasan la entrega.
  - **Solución**: Optimiza pasos (ejemplo, caché de dependencias) y usa entornos paralelos.
- **Errores de despliegue**:
  - **Problema**: Configuraciones incorrectas en entornos de producción.
  - **Solución**: Usa herramientas de infraestructura como código (ejemplo., Terraform) y prueba en staging.
- **Seguridad**:
  - **Problema**: Vulnerabilidades en dependencias.
  - **Solución**: Integra herramientas como Snyk y actualiza dependencias regularmente.

Aplicación a Desarrollar y probar

Nuestra aplicación se desarrollará usando **Spring Boot** y la librería **Faker** y tendrá los siguientes escenarios:

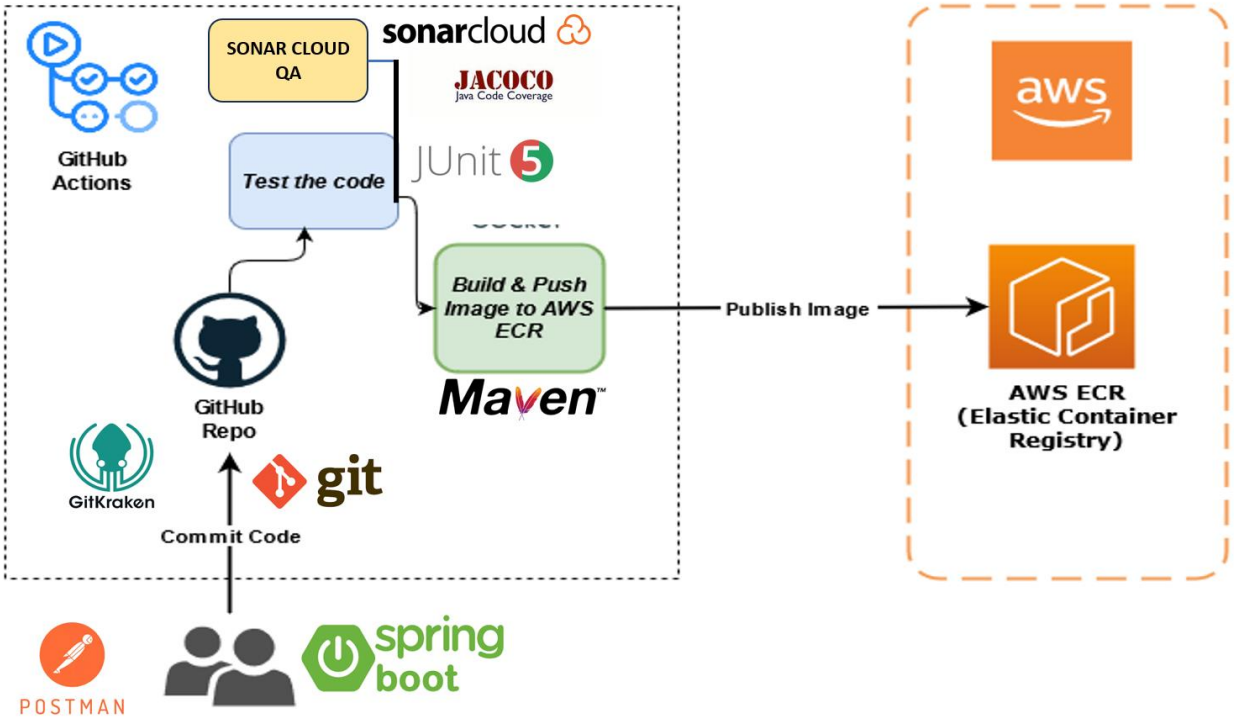
- Obtener naciones aleatorias
- Obtener monedas aleatorias
- Obtener información de vuelos aleatorios.
- Obtener versión de la aplicación
- Chequeo de estado de la App.

Requisitos previos

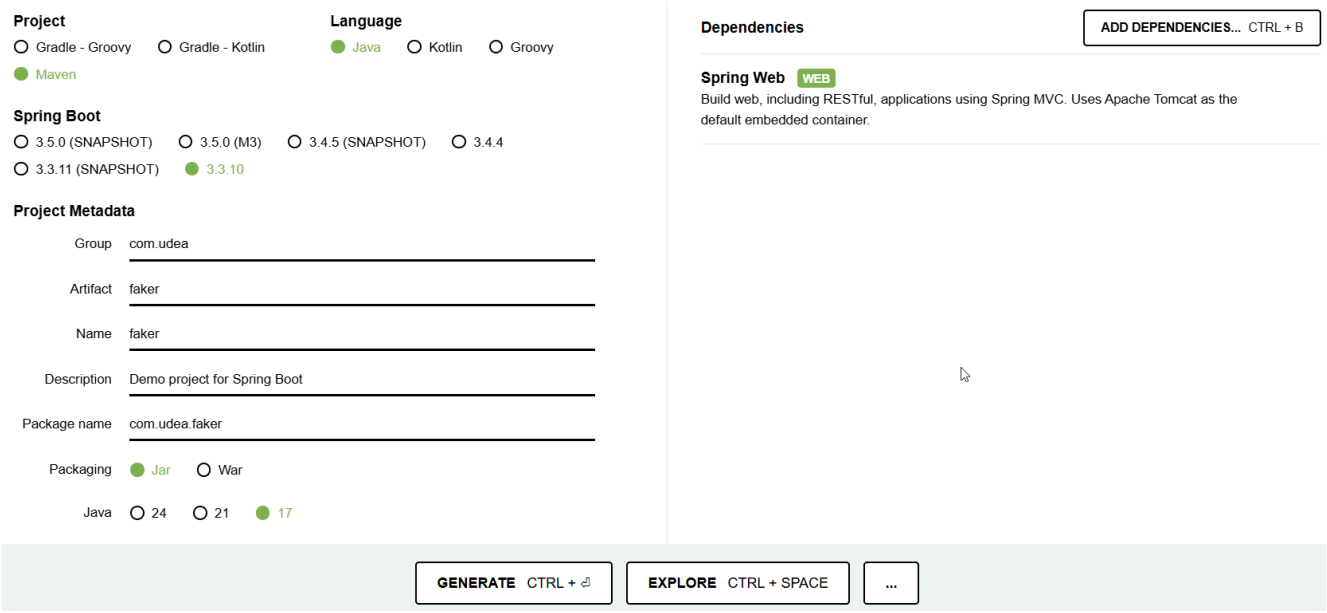
- JDK 11 o superior
- Git
- Spring Boot
- Editor de código como IntelliJ, Netbeans, Sublime, Visual Studio Code, Atom, etc.

- Gitkraken (GUI de GIT), que también nos facilitará enormemente trabajar con los repositorios de GIT.
- Cuenta en GitHub. Se recomienda crearla en GitHub Education con dominio @udea.edu.co.
- Cuenta en algún servicio Cloud como AWS, Azure, Render.

Arquitectura General de la App



Creamos el proyecto en Spring Boot a través de Initializr accediendo al siguiente link https://start.spring.io/



Deje la configuración que aparece arriba. Luego descargue el .ZIP del proyecto (Opción Generar) y descomprímalo en una carpeta de trabajo.

Abra el proyecto con IntelliJ o algún IDE de su preferencia

Usaremos la biblioteca **Java Faker** para obtener datos aleatorios, por lo que debemos agregar la siguiente dependencia a nuestro archivo **pom.xml.**

```
<dependency>
<groupId>com.github.javafaker</groupId>
<artifactId>javafaker</artifactId>
<version>1.0.2</version>
</dependency>
```

Ahora creamos la clase DataController para retornar los datos generados desde Faker con algunos atributos básicos mediante un Controlador REST. Si desea puede explorar otros métodos dados por Faker.

```
@RestController
public class DataController {
    @GetMapping("/")
    public String healthCheck() {
        return "HEALTH CHECK OK!";
    }

    @GetMapping("/version")
    public String version() {
        return "The actual version is 1.0.0";
    }

    @GetMapping("/nations")
    public JsonNode getRandomNations() {
        var objectMapper = new ObjectMapper();
        var faker = new Faker(new Locale("en-US"));
        var nations = objectMapper.createArrayNode();
        for (var i = 0; i < 10; i++) {
            var nation = faker.nation();
            nations.add(objectMapper.createObjectNode()
                    .put("nationality", nation.nationality())
                    .put("capitalCity", nation.capitalCity())
                    .put("bandera", nation.flag())
                    .put("language", nation.language()));
        }
        return nations;
    }

    @GetMapping("/currencies")
    public JsonNode getRandomCurrencies() {
        var objectMapper = new ObjectMapper();
        var faker = new Faker(new Locale("en-US"));
        var currencies = objectMapper.createArrayNode();
        for (var i = 0; i < 20; i++) {
            var currency = faker.currency();
            currencies.add(objectMapper.createObjectNode()
                    .put("name", currency.name())
                    .put("code", currency.code()));
        }
```

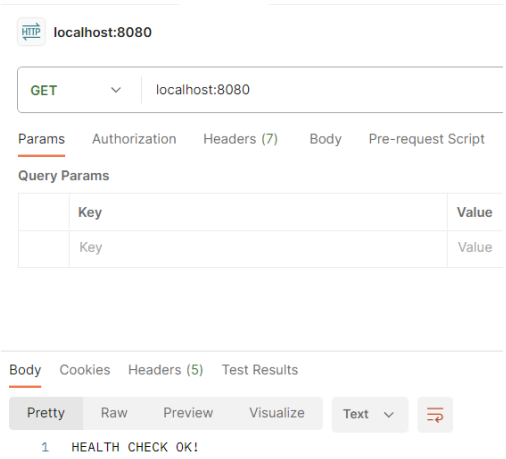
```
        return currencies;
    }

@GetMapping("/aviation")
    public JsonNode getRandomnAviation(){
        var objectMapper = new ObjectMapper();
        var faker = new Faker(new Locale("en-US"));
        var aviations = objectMapper.createArrayNode();
        for(var i=0;i<20;i++){
            var aviation = faker.aviation();
            aviations.add(objectMapper.createObjectNode()
                    .put("aircraft",aviation.aircraft())
                    .put("airport", aviation.airport())
                    .put("METAR", aviation.METAR())

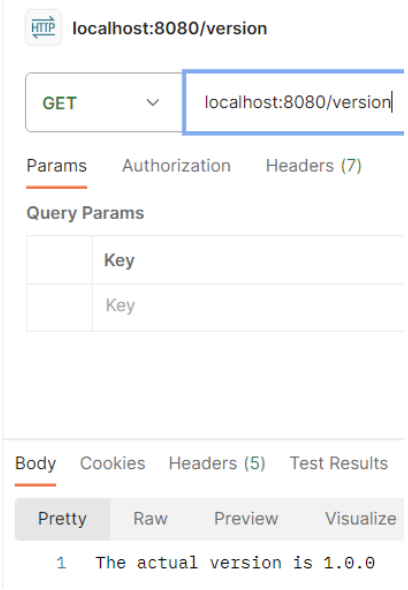
            );
        }
        return aviations;
    }
}
```

Ahora ejecutamos la Aplicación con el comando mvnw spring-boot:run o desde el entorno del IDE para revisar su funcionamiento con POSTMAN u otra herramienta similar (ejemplo curl, insomina, Restclient, etc), haciendo llamadas a las APIs con el método **HTTP GET**

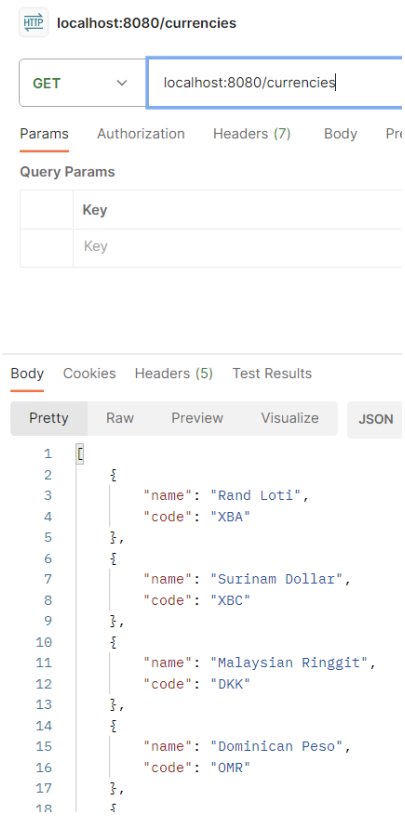
Ver el estado de la aplicación.



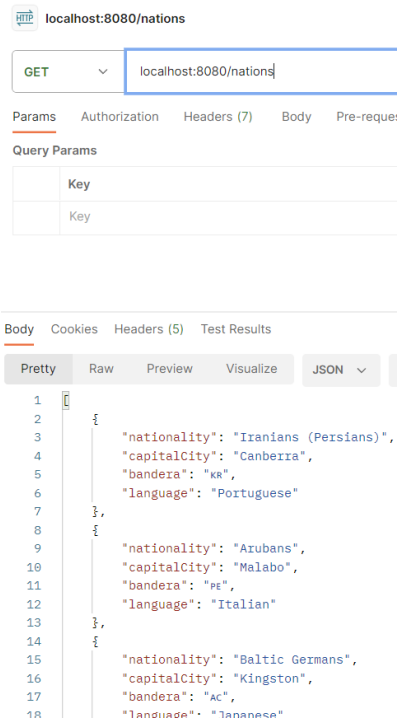
Ver la versión de la aplicación



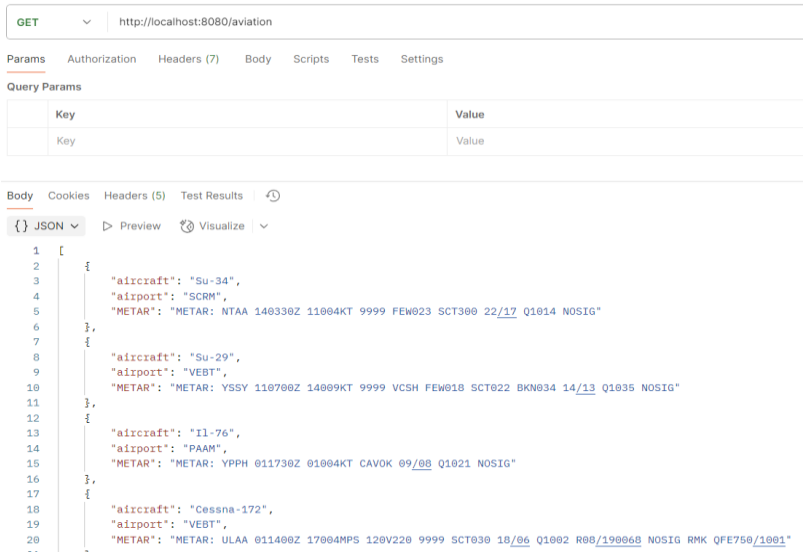
Ver listado de monedas



Ver lista de naciones



Listado de información sobre aviones



Ahora haremos las pruebas unitarias a nuestro REST Controller

En la carpeta Test de nuestro proyecto ubicamos la clase de pruebas y agregamos el siguiente código

```
package com.udea.consulta;

import com.fasterxml.jackson.databind.JsonNode;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
```

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

@SpringBootTest
class ConsultaApplicationTests {

	@Autowired
	DataController dataController;

	@Test
	void health() {
		assertEquals("HEALTH CHECK OK!", dataController.healthCheck());
	}

	@Test
	void version() {
		assertEquals("The actual version is 1.0.0", dataController.version());
	}

	@Test
	void nationLength() {
		Integer nationsLength = dataController.getRandomNations().size();
		assertEquals(10, nationsLength);
	}

	@Test
	void currenciesLength() {
		Integer currenciesLength = dataController.getRandomCurrencies().size();
		assertEquals(20, currenciesLength);
	}

	@Test
	public void testRandomCurrenciesCodeFormat() {
		DataController controller = new DataController();
		JsonNode response = controller.getRandomCurrencies();

		for (int i = 0; i < response.size(); i++) {
		JsonNode currency = response.get(i);
		String code = currency.get("code").asText();
		assertTrue(code.matches("[A-Z]{3}")); // Check for 3 uppercase letters format
		}
	}

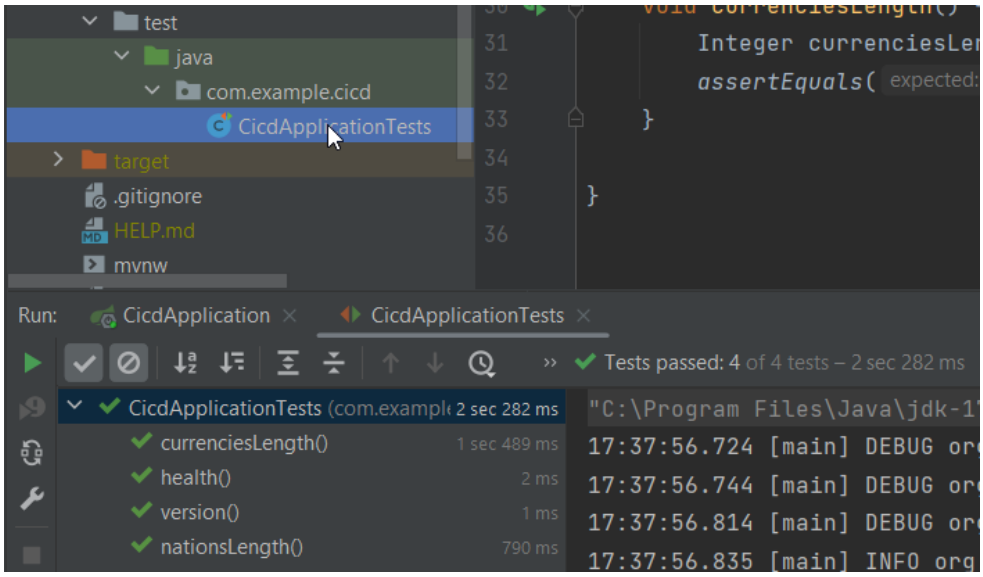
	@Test
	public void testRandomNationsPerformance() {
		DataController controller = new DataController();
		long startTime = System.currentTimeMillis();

		controller.getRandomNations();

		long endTime = System.currentTimeMillis();
		long executionTime = endTime - startTime;
		System.out.println(executionTime);

		// Assert that execution time is within acceptable limits
		assertTrue(executionTime < 2000); // 2 second threshold
	}
	@Test
	void aviationsLength() {
		Integer aviationsLength = dataController.getRandomnAviation().size();
		assertEquals(20, aviationsLength);
	}
}
```

Ejecutamos la clase de Test y miramos la ejecución de cada aserción exitosamente.

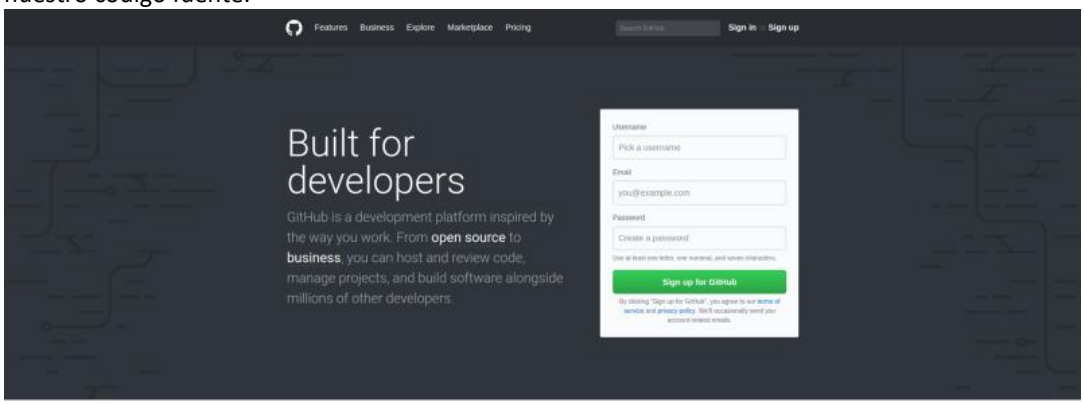


Ahora tenemos una aplicación SpringBoot bien probada, lista para implementar.

Ahora subiremos el proyecto a GitHub

1. **Integración al proyecto en GitHub**

La idea es crear un entorno de desarrollo colaborativo usando cómo repositorio de código GitHub dónde subiremos nuestro código fuente.



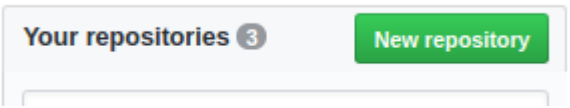
Después, lo integraremos con GitHub Actions y Sonar Cloud que nos van a permitir verificar la calidad del código, hacer procesos automáticos de integración continúa usando pipelines, ejecutar pruebas unitarias y mucho más.

Nota: Se recomienda usar o crear una cuenta con el dominio institucional **@udea.edu.co** para acceder a los beneficios de GitHub Education.

Creación del proyecto en GitHub

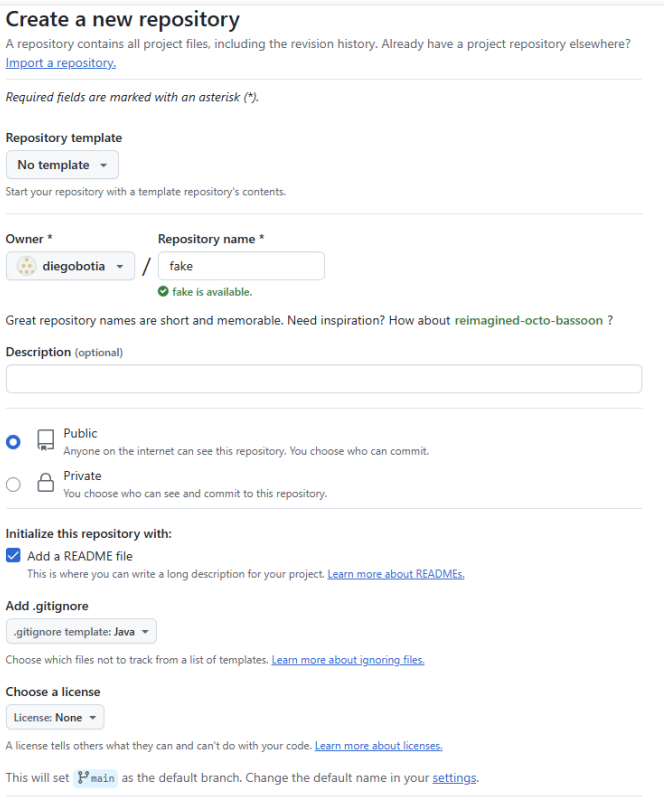
Lo primero que haremos será abrir una cuenta en GitHub.

Ahora creamos un nuevo repositorio en GitHub, para ello basta con pulsar desde la pantalla principal en el botón de **New Repository**.



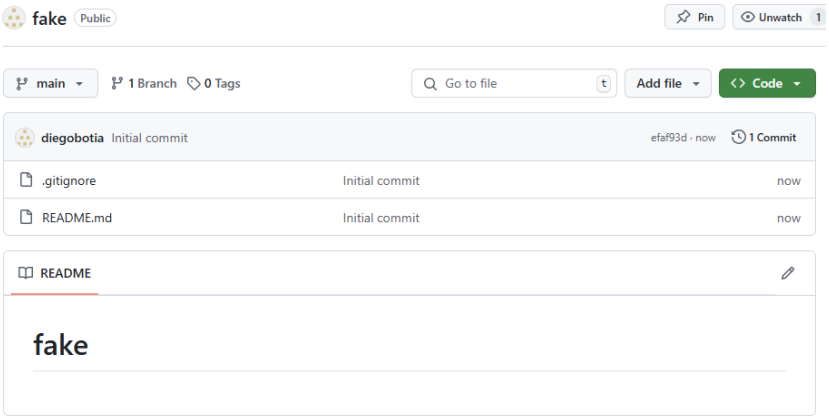
A continuación, llenamos los datos del nombre del proyecto, la descripción, lo dejamos marcado como público y además marcamos la casilla de inicializar el repositorio con un fichero **README**.

También aprovechamos para añadir el archivo **.gitignore** preparado para Java que nos servirá para evitar subir al repositorio archivos que no consideremos fuentes.



Creando el repo en GitHub

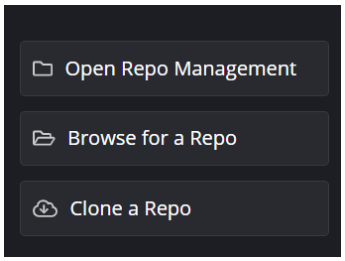
Pulsamos el botón de **Create repository** y ya tendremos nuestro repositorio listo.



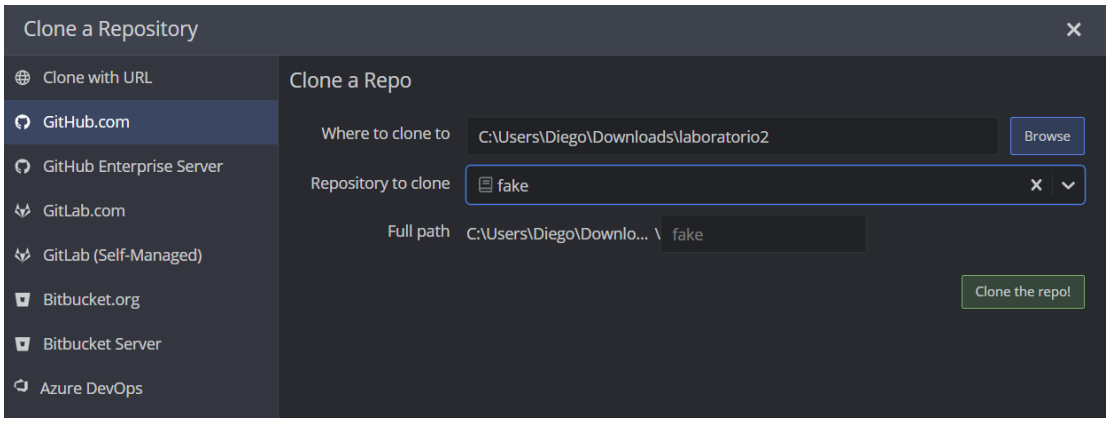
Ahora el repositorio en GitHub está listo para empezar

Clonación del proyecto con GitKraken

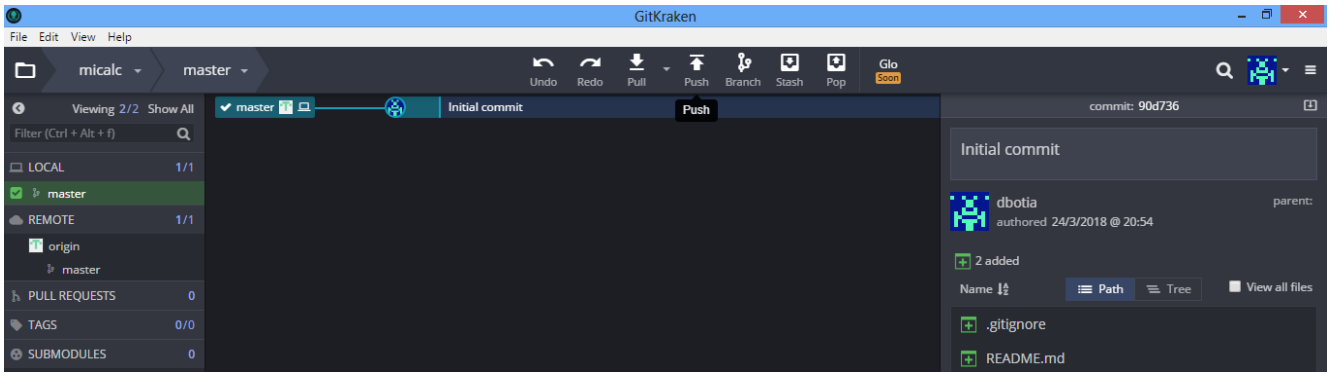
Ahora vamos a **clonar el proyecto,** usando GitKraken. Recuerde que para poder clonar el proyecto de GitHub hay que tenerlo asociarlo a la cuenta de GitHub previamente.



En la nueva ventana seleccione el directorio donde va a clonar el repo y seleccione de su GitHub el repositorio que se clonara:



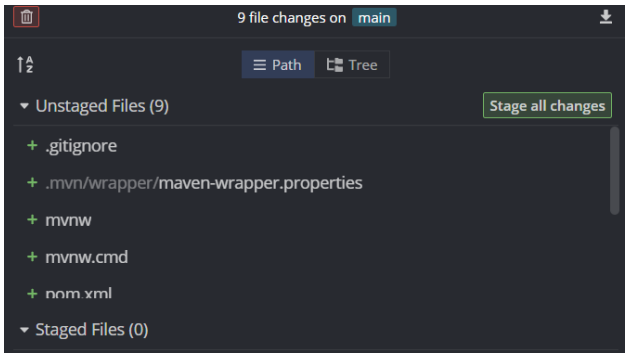
Clonamos el proyecto con GitKraken



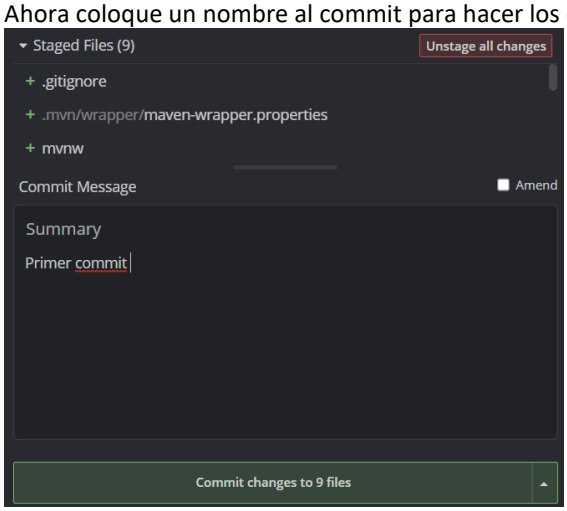
Nuestro repositorio esta clonado con Gitkraken y listo para empezar

Copie la carpeta de su proyecto Spring Boot y péguelo en el nuevo directorio clonado

Ahora que ya tenemos el código principal de nuestro proyecto, lo subimos a nuestro repositorio de GitHub. Para iniciar haga click sobre el botón **Stage all Changes**

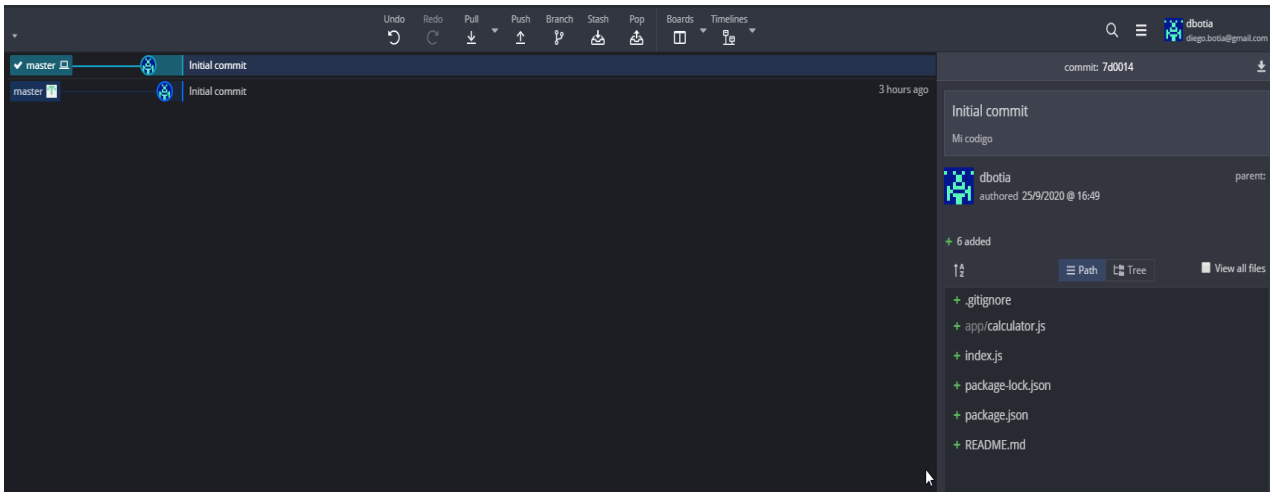


Ahora coloque un nombre al commit para hacer los cambios en el repositiorio GIT local.

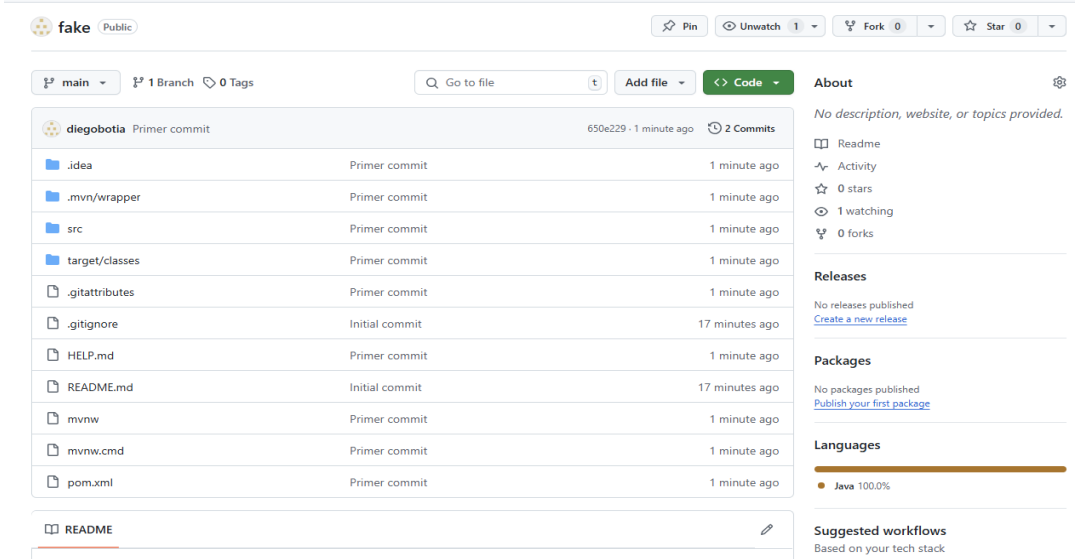


Ahora haga click en el botón **Push** 

el cual enviará los cambios a nuestro repositorio de Github en la rama **main**.



Verifique en su repo de GitHub que ya estén todos los cambios realizados y actualizados.



Creación del Pipeline de CI/CD
Entonces, comencemos a crear el pipeline de CI/CD.

En este Laboratorio, usaremos GitHub Actions para crear nuestro pipeline. GitHub Actions es una herramienta para automatizar su flujo de trabajo desde el código hasta la producción.

GitHub Actions facilita la automatización de todos sus flujos de trabajo de software. Permite crear, probar e implementar su código directamente desde GitHub.

Primero, necesitamos crear el archivo **.github/workflows/build.yml**.

```
#Workflow name
  name: CI/CD Pipeline
  on:
#Manually trigger workflow runs
    workflow_dispatch:
#Trigger the workflow on push from the main branch
    push:
      branches:
```

```
        - main
  jobs:
#Test's job
    tests:
      name: Unit tests
#Run on Ubuntu using the latest version
      runs-on: ubuntu-latest
#Job's steps
      steps:
  #Check-out your repository under $GITHUB_WORKSPACE, so your workflow can access it
        - uses: actions/checkout@v1
  #Set up JDK 11
        - name: Set up JDK
          uses: actions/setup-java@v1
          with:
            java-version: '11'
  #Set up Maven cache
        - name: Cache Maven packages
#This action allows caching dependencies and build outputs to improve workflow execution
time.
          uses: actions/cache@v1
          with:
            path: ~/.m2
            key: ${{ runner.os }}-m2-${{ hashFiles('**/pom.xml') }}
            restore-keys: ${{ runner.os }}-m2
  #Run Tests
        - name: Run Tests
          run: mvn -B test
```

En el archivo anterior, tenemos un pipeline llamado CI/CD Pipeline, se activará manualmente o cuando detecte una inserción en la rama principal. Entre los principales elementos a tener en cuenta dentro del archivo estan:

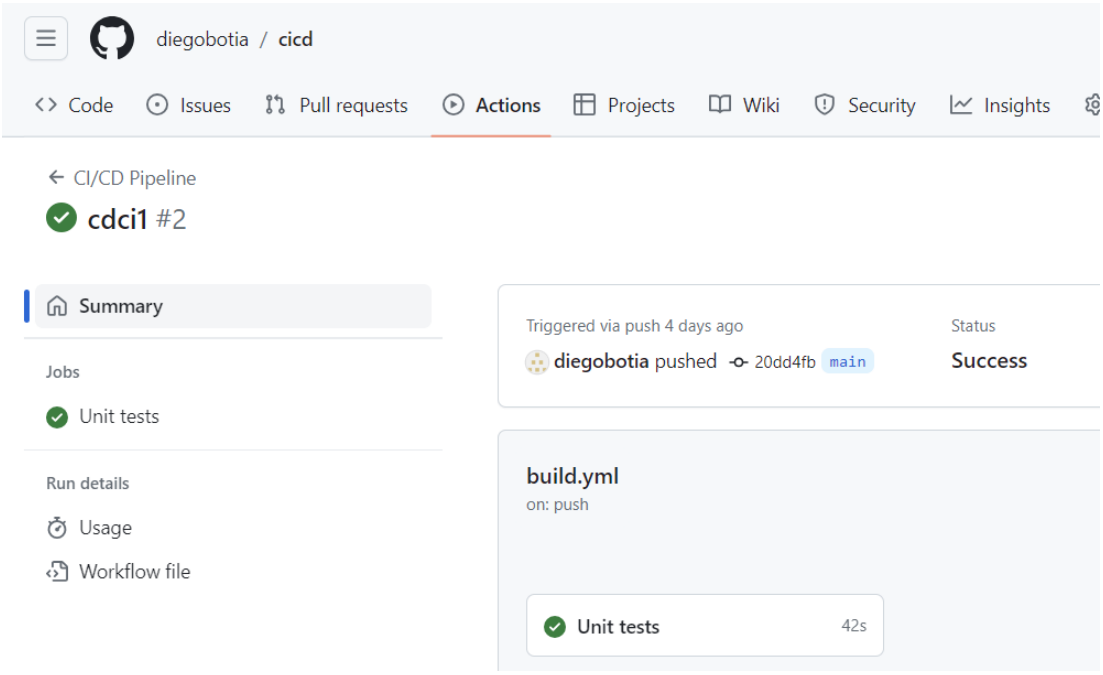
- **name**: Este es el nombre de su flujo de trabajo.
- **on**: especifica los eventos que desencadenan el flujo de trabajo. En este caso, se ejecuta en inserciones a la rama main
- **jobs**: En esta sección se definen los trabajos que se ejecutarán. Cada trabajo se ejecuta en una instancia nueva de un entorno virtual.
- **steps**: Estas son las tareas individuales que se ejecutarán en el trabajo.

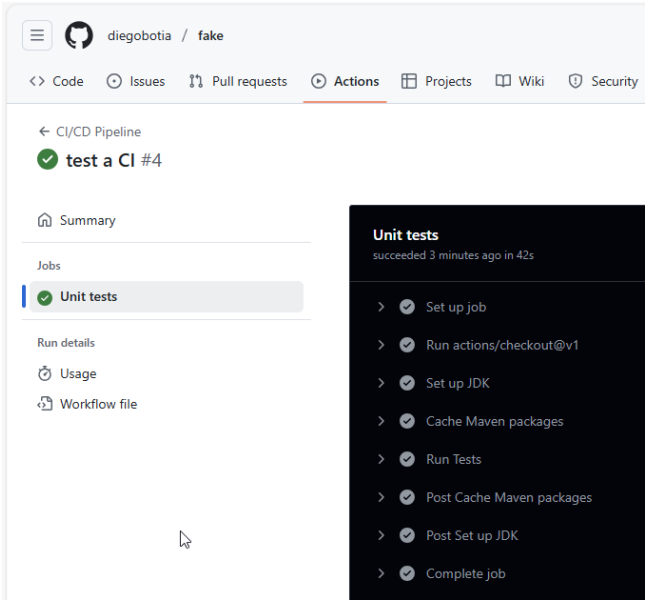
Este pipeline contiene un único trabajo llamado tests con una etiqueta de nombre Unit tests, este trabajo contiene varios pasos. Primero, configurará el JDK 11, luego configurará el caché de Maven y finalmente ejecutará las pruebas de la aplicación.

Ahora probamos nuestra canalización y vemos qué sucede.

Enviaremos el archivo **build.yml** y el flujo de trabajo se ejecutará automáticamente.

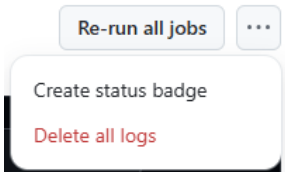
Ahora vaya a su repositorio de Github y haga clic en Acciones. Encontrará el estado del pipeline y los registros de pasos.



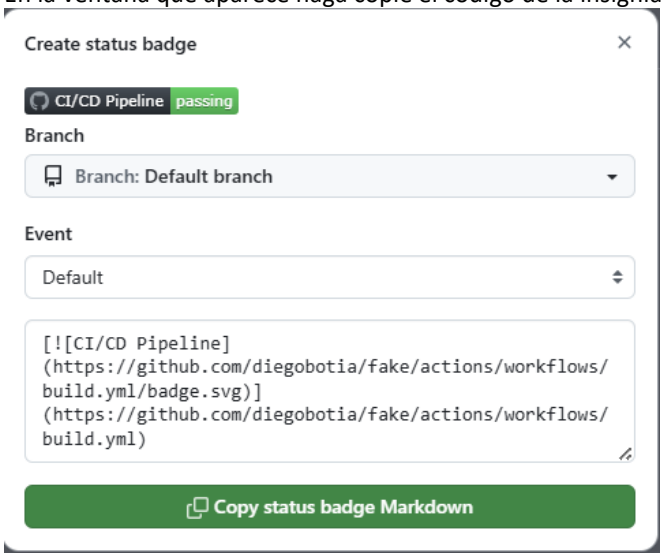


Nota: Para evitar errores en la ejecución y si hay problemas de compatibilidad con alguna versión del JDK de Java o de Spring Boot, usaremos para las pruebas el JDK 11 y la versión 2.7.17 de Spring Boot los cuales debe configurar en el archivo **POM.XML** antes de subir el repo a GitHub.

Ahora, vamos a poner el distintivo del test unitario en nuestro README.md mediante el Markdown y de paso añadir el funcionamiento de nuestra app. En la parte superior derecha haga click en los tres puntos (…) y haga click en la opción **Create Status Badge.**



En la ventana que aparece haga copie el código de la insignia la cual colocaremos dentro del archivo Readme.md



Contenido del archivo README

La estructura del archivo Readme será la siguiente:

```
[![CI/CD
Pipeline](https://github.com/<Usuario>/<Repo>/actions/workflows/build.yml/badge.svg)](https://github.com/<Usuario>/<Repo>/actions/workflows/build.yml)

Implementation of a Simple App with the next operations:

* Get random nations
* Get random currencies
* Get random Aircraft
* Get application version
* health check

Including integration with GitHub Actions, Sonarqube (SonarCloud), Coveralls and
Snyk

### Folders Structure

In the folder `src` is located the main code of the app

In the folder `test` is located the unit tests

### How to install it

Execute:

```shell
```
```

```
$ mvnw spring-boot:run
```
to download the node dependencies

How to test it

Execute:

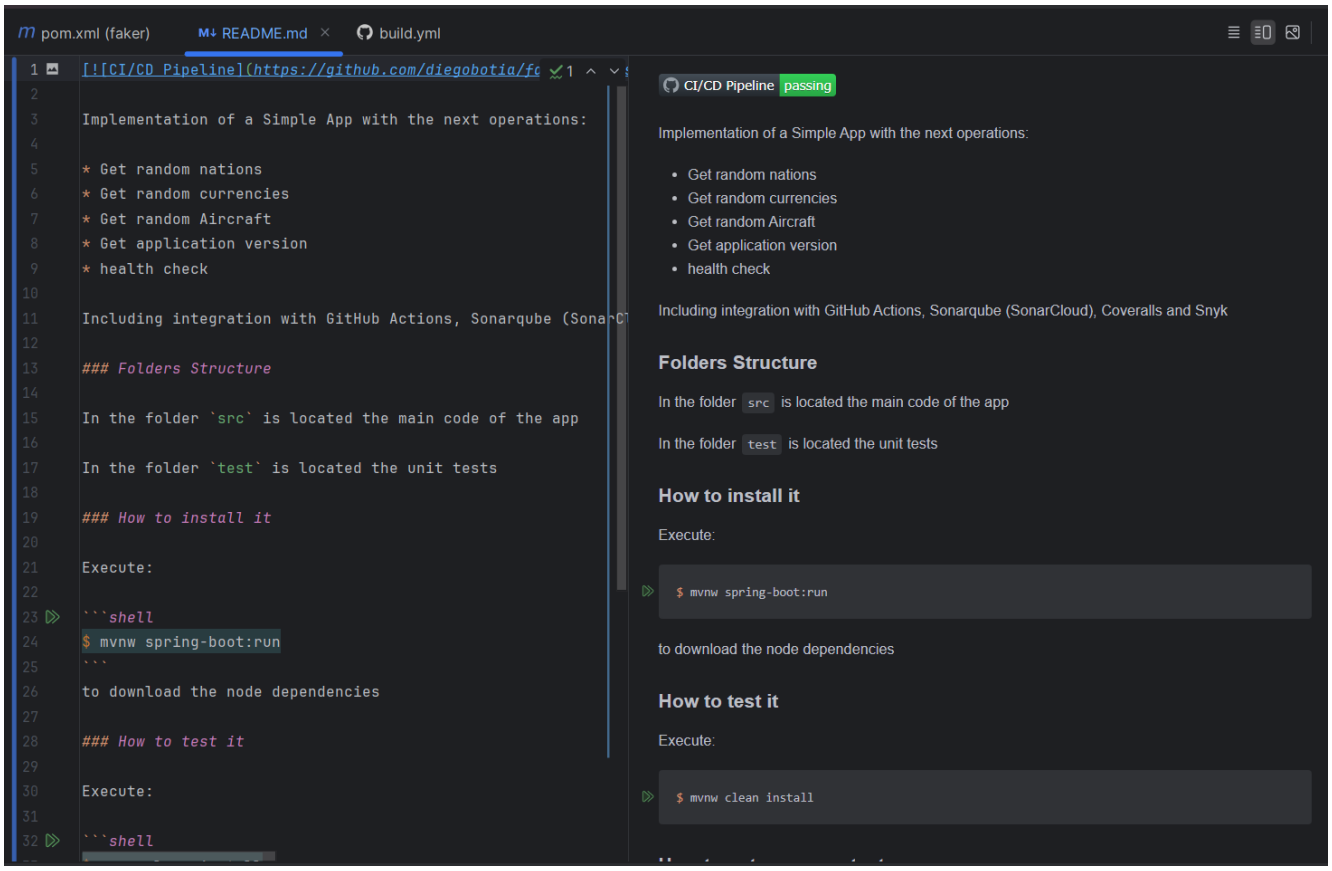
```shell
$ mvnw clean install
```

How to get coverage test

Execute:

```shell
$ mvwn -B package -DskipTests --file pom.xml
```

Observe que en el editor de IntelliJ aparece una vista previa de su archivo con el formato de Markdown



Puede actualizar el archivo Readme a su repo de GitHub para actualizar los cambios

Arquitectura de Software
Universidad de Antioquia

Integración con SonarCloud para Q.A

Ahora integraremos el proyecto con la conocida herramienta de verificación de calidad de código **Sonarqube**, en su versión online Sonarcloud (https://sonarcloud.io/). SonarCloud es el servicio en línea líder para detectar errores y vulnerabilidades de seguridad en sus repositorios de código.

Este es un servicio de análisis de código estático basado en la nube diseñado para medir la calidad del código. Esta herramienta soporta más de 25 lenguajes de programación. Usa diferentes técnicas documentadas de análisis de código estático para detectar problemas. El análisis de código estático ofrece una capa adicional de verificación, diferente a las pruebas automatizadas y la revisión de código por personas.

Esta herramienta funciona con GitHub, Bitbucket Cloud, Azure DevOps y GitLab. Cada vez que se importa una organización se convierte en una organización SonarCloud y cada repositorio importado se convierte en un proyecto SonarCloud.

SonarCloud identifica defectos y hotspots de seguridad en el código.

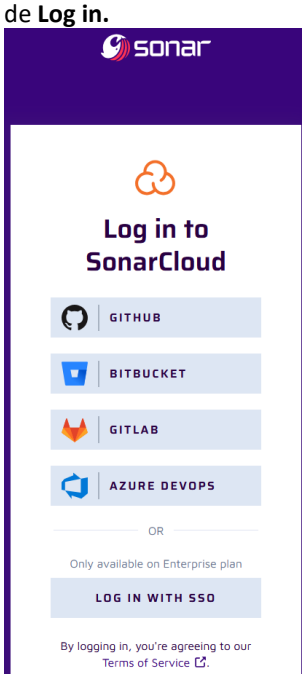
Defectos: Un defecto es un problema en el código que requiere corrección. Los defectos son agrupados así:

1. **Code Smells**: Son características del código que mientas no afectan el funcionamiento apropiado del programa, podría indicar problemas profundos que afectan negativamente la mantenibilidad del código.
2. **Bugs:** Son errores en el código que afectan la fiabilidad del código.
3. **Vulnerabilidades**: Estos son problemas en el código que podrían ser explotados por un tercero para comprometer la seguridad de la aplicación.

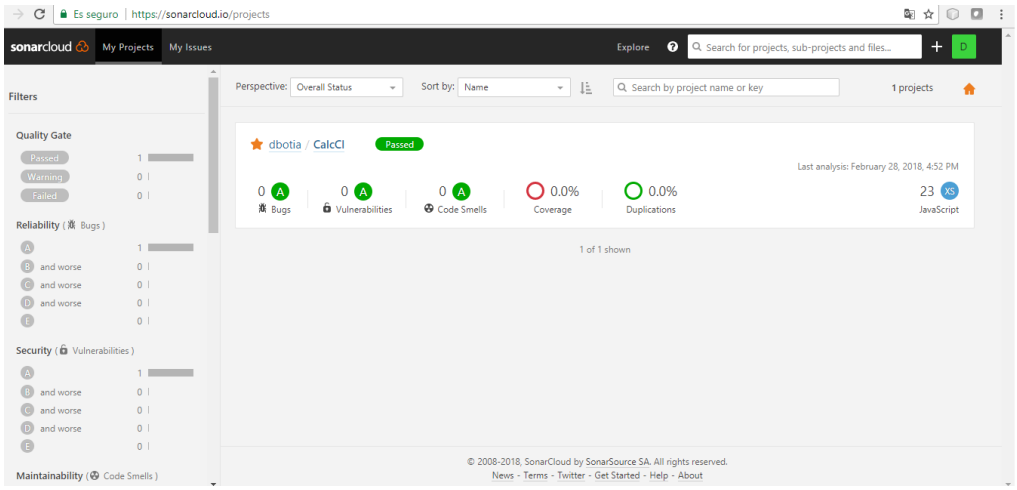
Hotspots de seguridad: Son áreas del código que pueden causar fallas de seguridad y por lo tanto necesitan ser revisados.

Una falla es generalmente un problema real, mientras que un hotspot de seguridad puede convertirse en una falsa alarma (aun así, debe ser revisado).

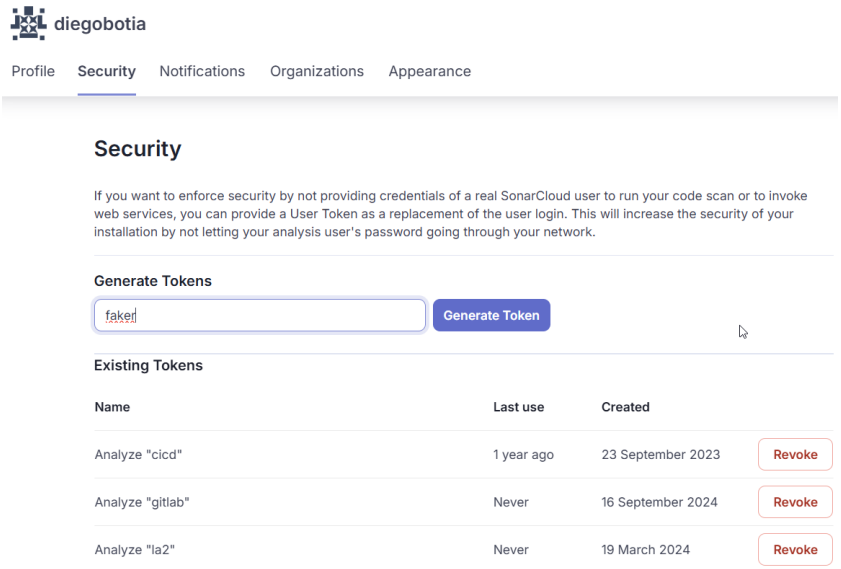
Lo primero que haremos será crear una cuenta en Sonarcloud asociada a nuestra cuenta de GitHub, pulsando en el botón de **Log in.**



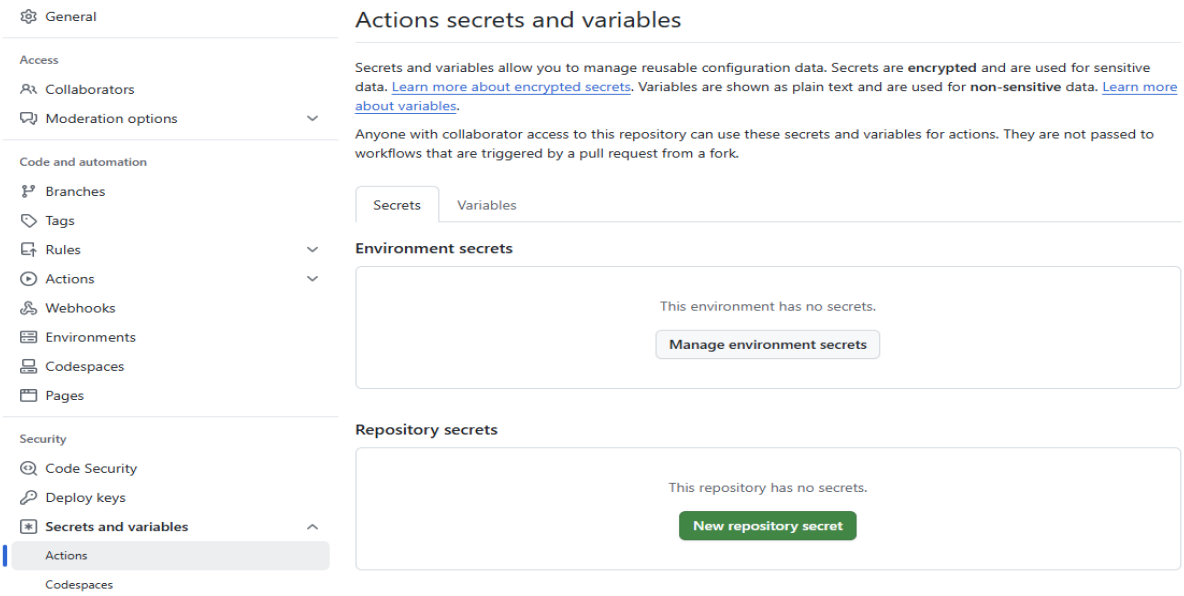
Una vez dentro, se nos muestran algunos proyectos



Ahora hay que integrar GH Actions con SonarCloud. Lo primero que haremos es entrar en nuestra cuenta de SonarCloud, y en la pestaña de **Security** de la opción **MyAccount** generamos un **token.**

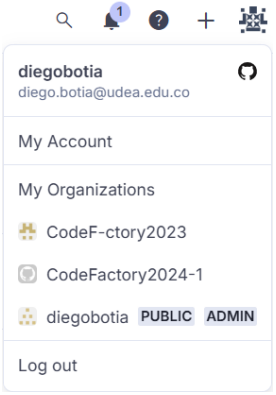


Este token lo copiamos al portapapeles, pulsando en el botón de **Copy**, luego vamos a nuestro repositorio en GitHub y seleccionamos la opción Setting → Secrets and Variables→ Actions.

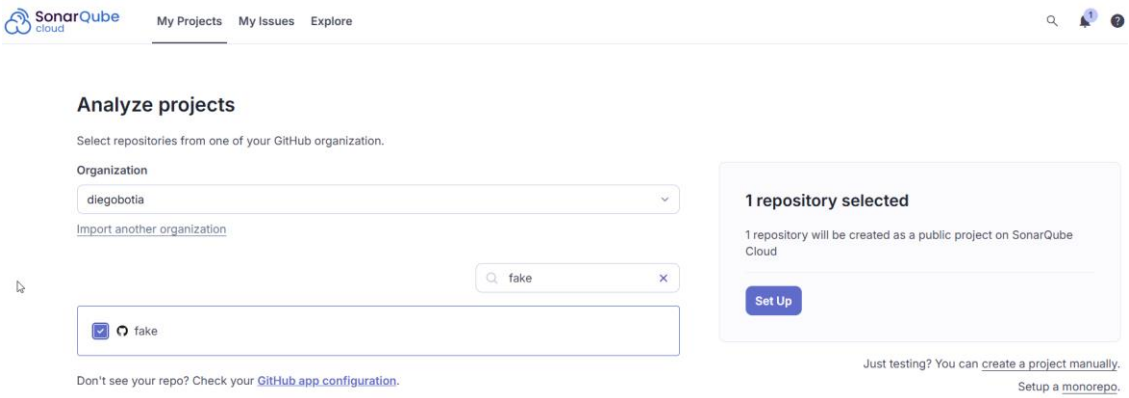


Ahora hacemos click en el botón **New Repository Secret**, luego añadimos una variable de entorno de nombre **SONAR_TOKEN** con el valor que hemos obtenido de SonarCloud y se agrega con el botón **Add Secret**.

Ahora identificamos nuestra organización para esto en la opción de la cuenta debe aparecer el nombre. Este parámetro lo necesitaremos en el archivo de propiedades de sonar en nuestro proyecto que se explicará más adelante.

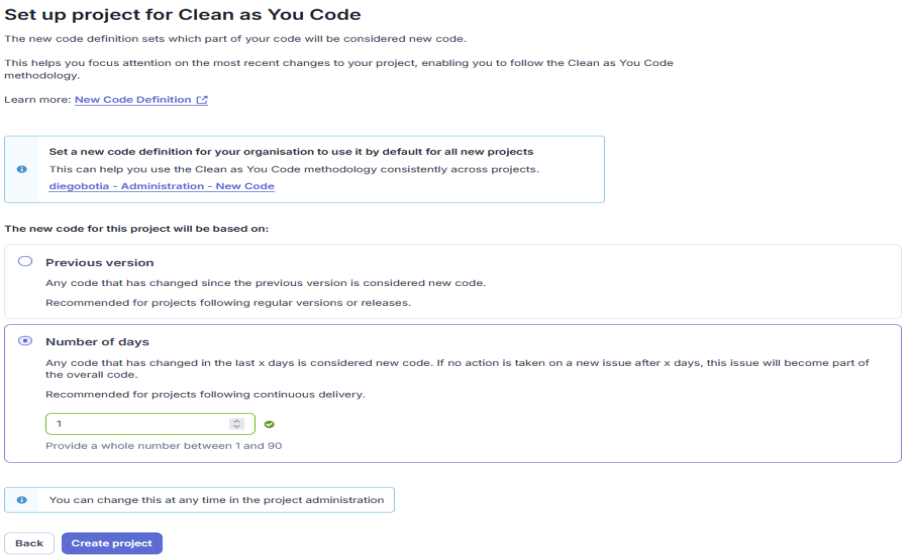


Si creamos un nuevo proyecto en Sonar podremos enlazar el repositorio de GitHub automáticamente.

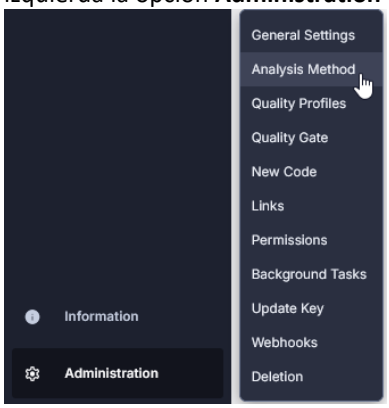


Elegimos nuestro repositorio y luego iniciamos la configuración con el botón **Set Up.**

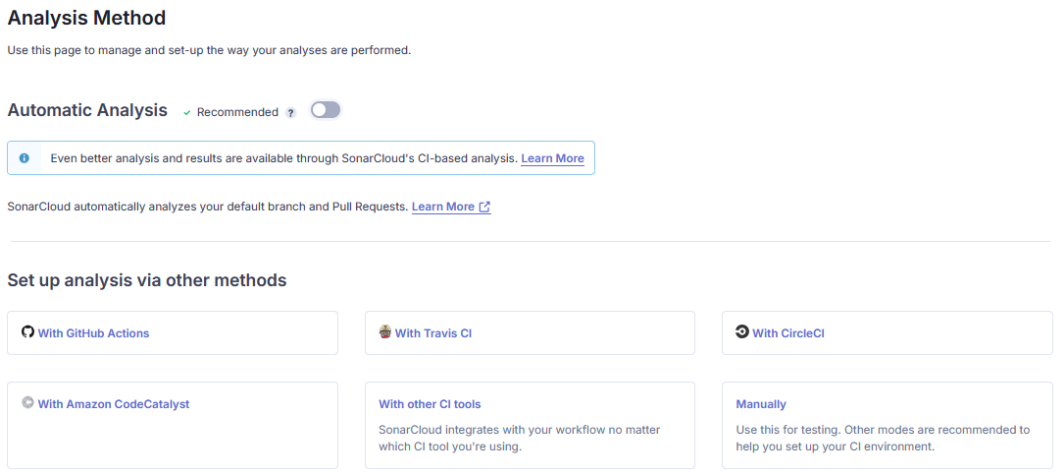
En la ventana llamada **Set up project for Clean as You Code** debemos seleccionar el método para la definición de nuevo código que puede ser basado en Versiones previas o en número de días. Para este ejercicio seleccionaremos la última opción y colocaremos 1 día. Esto obligará a hacer el respectivo escaneo según los últimos cambios del día determinado. Haga click en el bóton **Create Project.**



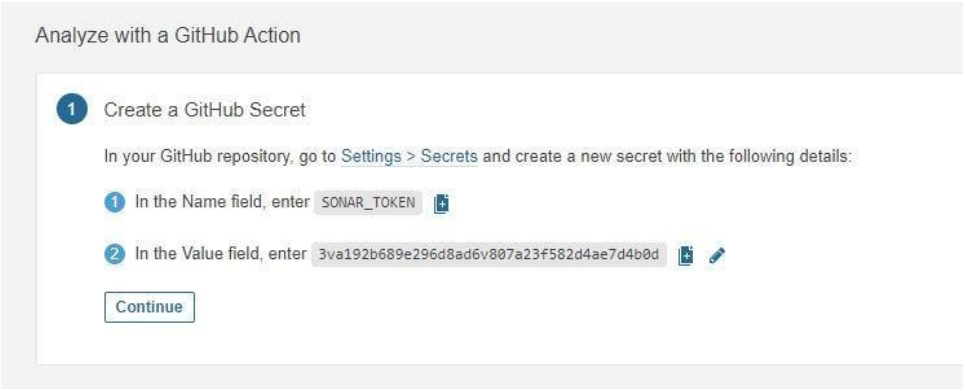
Ahora seleccionamos el Método de Análisis, en nuestro caso será usando Github Actions . Seleccione en el menú de la izquierda la opción **Administration** y luego seleccione Analysis Method.



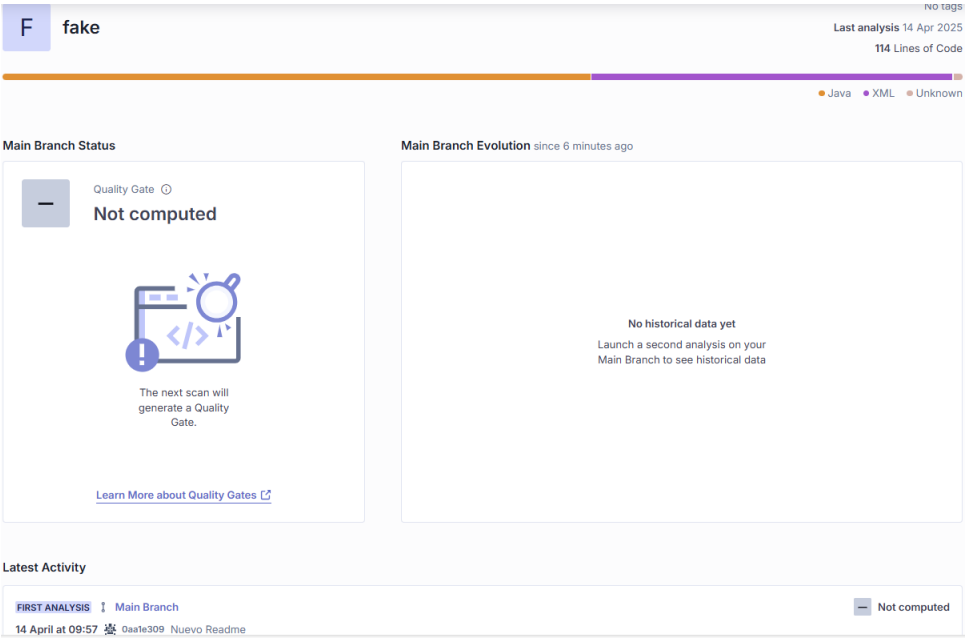
Por el momento desactive la opción **Automatic Analysis** ya que haremos el escaneo del proyecto desde la conexión con GitHub Actions.



Ahora SonarCloud nos ofrece una variable de ambiente llamada **SONAR_TOKEN** que debemos añadir a nuestro repositorio (ver más arriba los pasos).

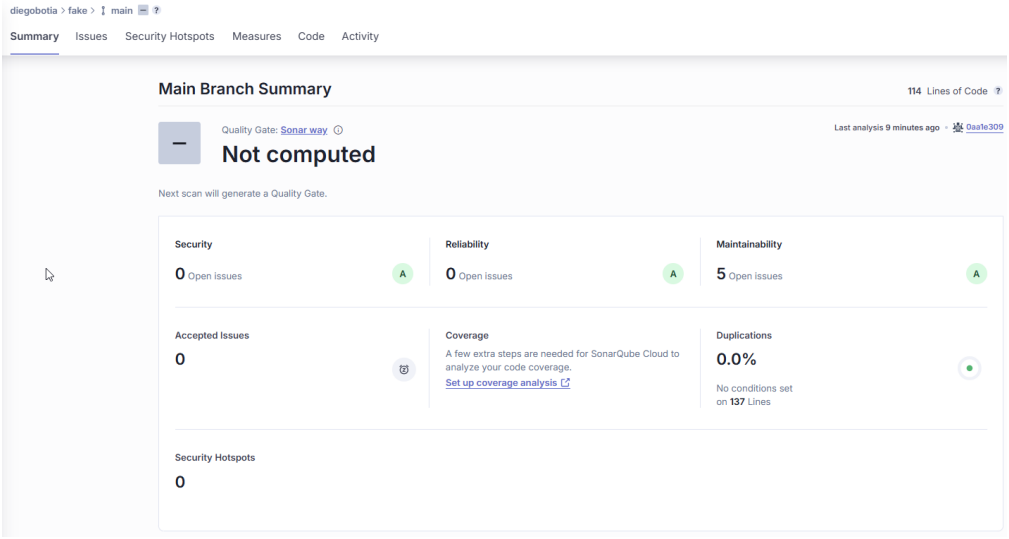


Como puede observar es posible que no tengamos cálculos iniciales. Es necesario esperar que se active el primer escaneo.



Ahora en el menú del lado izquierdo buscamos la opción "**Information**" del proyecto y copiamos el **Project_key** que colocaremos posteriormente en el archivo de propiedades de sonar.

Si hacemos click en el menú del lado izquierdo sobre la opción **Main Branch** observamos que en el primer escaneo aparece un resumen de los Issues y Security Hotspots detectados.. Hay que tener en cuenta que pueden presentarse casos de falsos positivos o de segmentos de código que no necesariamente están errados. En otro caso se puede recomendar hacer uso de técnica de refactorización de código si es necesario.



Una vez hecho esto, pasamos al siguiente paso. SonarCloud nos dará propiedades y configuraciones del pipeline en el POM.XML. Coloque en la sección properties lo siguiente:

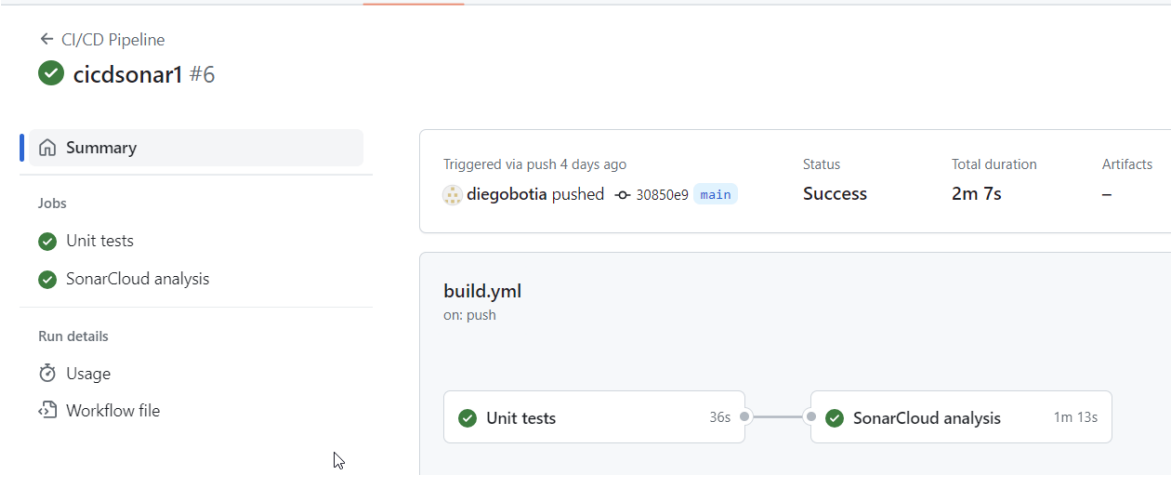
```
<properties>
            <java.version>11</java.version>
            <sonar.projectKey>PROJECT_KEY_SONAR</sonar.projectKey>
            <sonar.organization>PROJECT_ORGANIZATION</sonar.organization>
            <sonar.host.url>https://sonarcloud.io</sonar.host.url>
</properties>
```

En el archivo **build.yml** agregaremos un nuevo trabajo llamado sonar

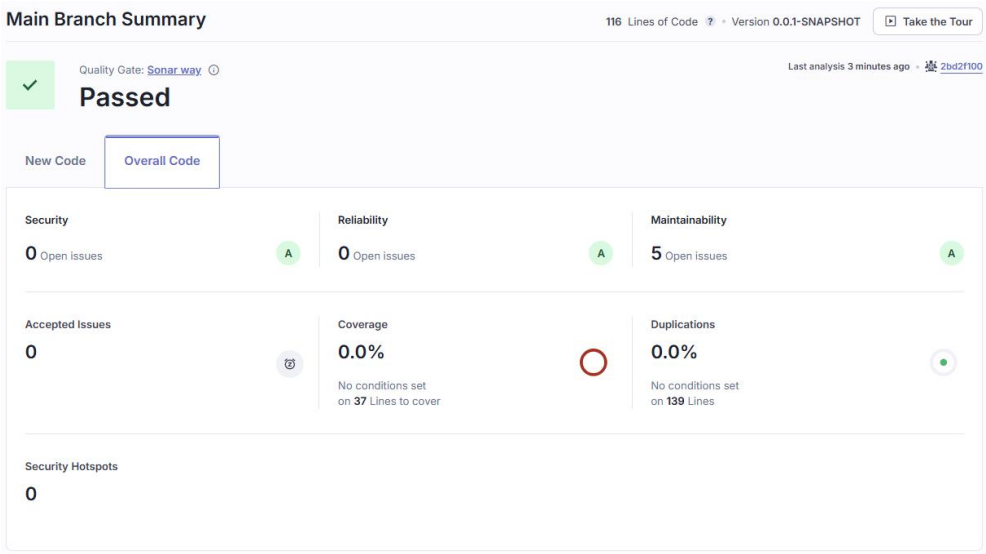
```
#Sonar's Job
sonar:
  #Depends on test's job
  needs: tests
  name: SonarCloud analysis
  #Run on Ubuntu using the latest version
  runs-on: ubuntu-latest
  #Job's steps
```

```
  steps:
    #Check-out your repository under $GITHUB_WORKSPACE, so your workflow can
access it
    - uses: actions/checkout@v1
    #Set up JDK 17
    - name: Set up JDK
      uses: actions/setup-java@v1
      with:
        java-version: '17'
    #Set up SonarCloud cache
    - name: Cache SonarCloud packages
      #This action allows caching dependencies and build outputs to improve
workflow execution time.
      uses: actions/cache@v3
      with:
        path: ~/.sonar/cache
        key: ${{ runner.os }}-sonar
        restore-keys: ${{ runner.os }}-sonar
    #Set up Maven cache
    - name: Cache Maven packages
      #This action allows caching dependencies and build outputs to improve
workflow execution time.
      uses: actions/cache@v3
      with:
        path: ~/.m2
        key: ${{ runner.os }}-m2-${{ hashFiles('**/pom.xml') }}
        restore-keys: ${{ runner.os }}-m2
    #Analyze project with SonarCloud
    - name: Analyze with SonarCloud
      run: mvn -B verify sonar:sonar -Dsonar.projectKey=diegobotia_gitlab -
Dsonar.organization=diegobotia -Dsonar.host.url=https://sonarcloud.io -
Dsonar.login=$SONAR_TOKEN
      env:
        GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
        SONAR_TOKEN: ${{ secrets.SONAR_TOKEN }}
```

Como puede ver, el trabajo llamado sonar depende del trabajo llamado tests, lo que significa que una vez que el tests falla, sonar no se ejecutará. Suba los cambios a su repo y espere que se ejecute el nuevo Job en GH Actions.



Ahora que todo se ejecutó, damos un vistazo a SonarCloud para ver el resultado.



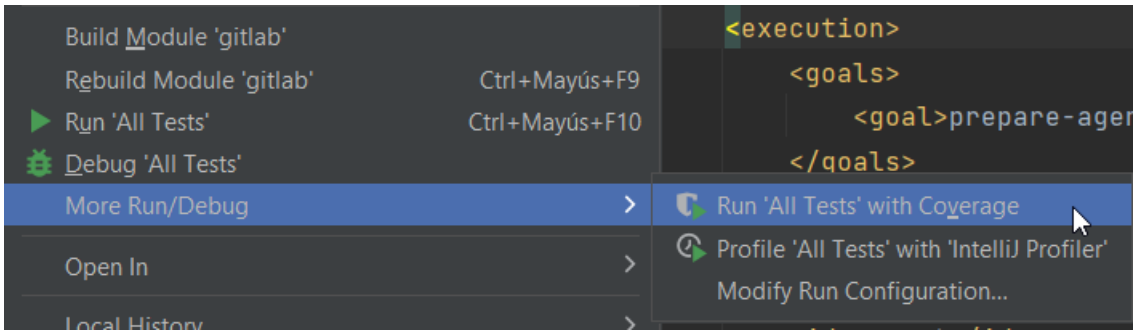
Todo se ve bien, excepto la cobertura, por qué es 0% a pesar de que agregamos pruebas unitarias. Esto es porque necesitamos agregar el complemento JaCoCo a nuestro archivo **pom.xml**.

Cobertura de Código con JaCoCo

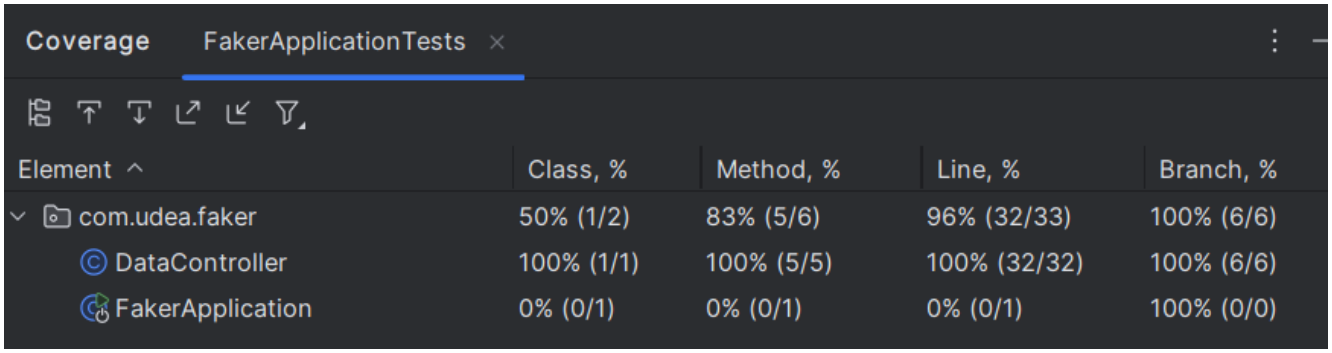
Agregamos el siguiente plugin para tener configurado a JaCoCo (Java Code Coverage) que se usará más adelante para determinar la cobertura de código. Agregue en su archivo POM.XML lo siguiente:

```
<plugin>
	<groupId>org.jacoco</groupId>
	<artifactId>jacoco-maven-plugin</artifactId>
	<version>0.8.10</version>
	<executions>
	<execution>
			<goals>
				<goal>prepare-agent</goal>
			</goals>
		</execution>
		<!-- attached to Maven test phase -->
		<execution>
			<id>report</id>
			<phase>test</phase>
			<goals>
				<goal>report</goal>
			</goals>
		</execution>
	</executions>
</plugin>
```

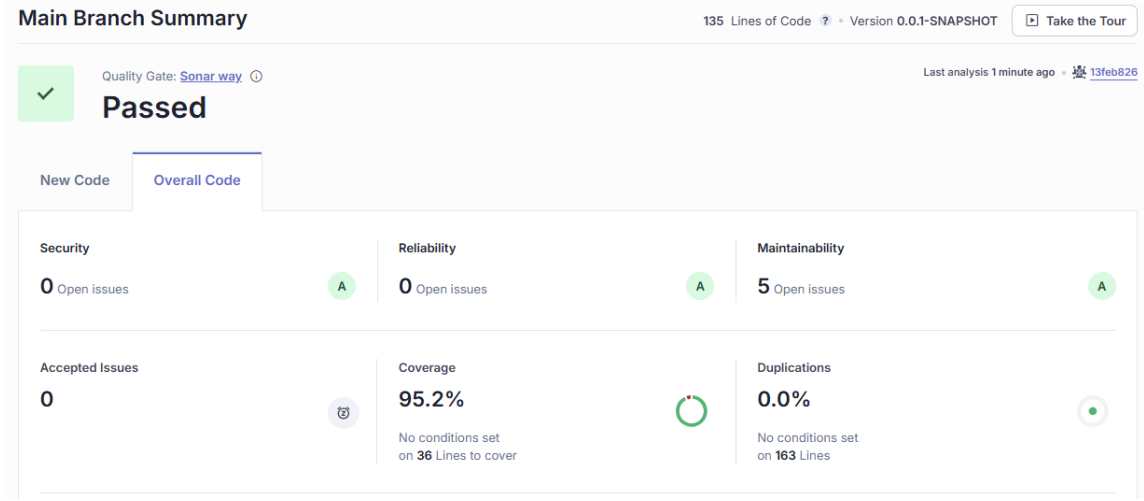
Puede primero probar la ejecución de la cobertura de código local usando la siguiente ruta en IntelliJ:



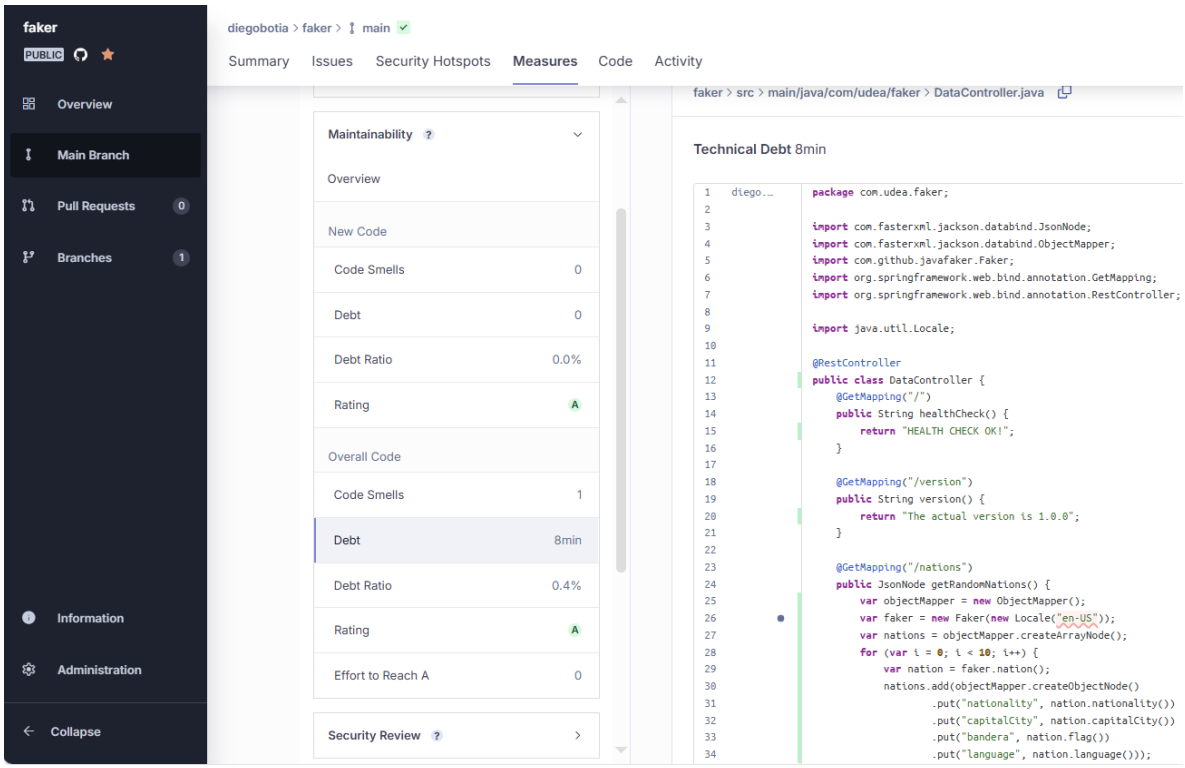
Después de la ejecución se observa la cobertura de código realizada por la clase Test



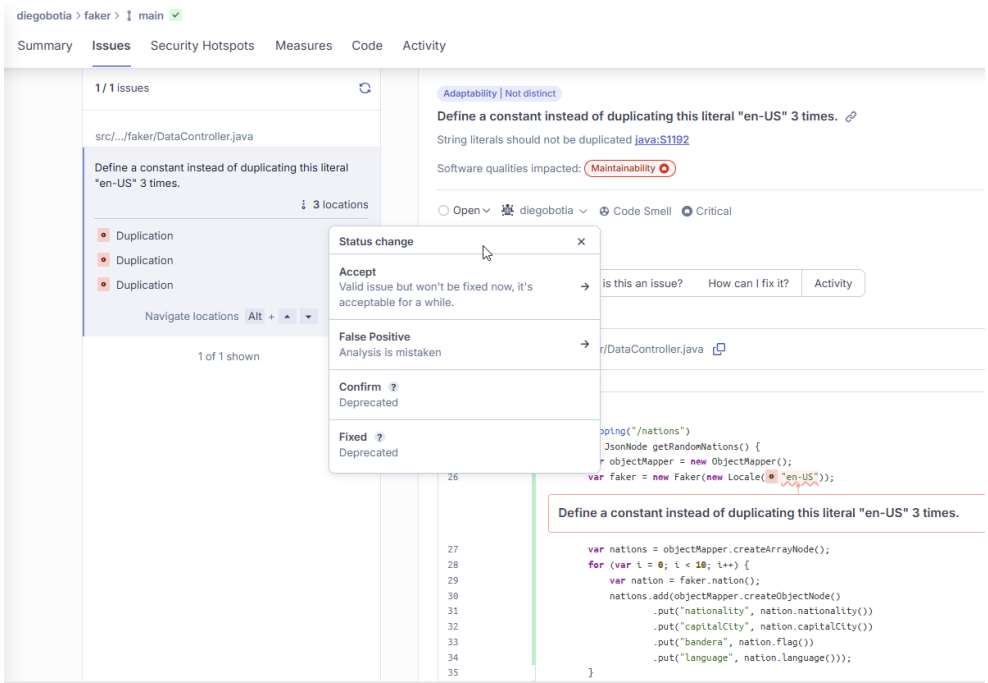
Actualice de nuevo el repo en GitHub con los cambios.
Al momento de revisar de nuevo en SonarCloud ya nos aparece el valor de cobertura de código.



Si observa en la característica de mantenibilidad, Sonar informa que encontró 5 posibles problemas con el código, por lo tanto si vemos el detalle del mismo como se observa en la siguiente figura nos muestra que hay una deuda técnica calculada de 8 minutos y nos señala sobre el código del posible problema encontrado.

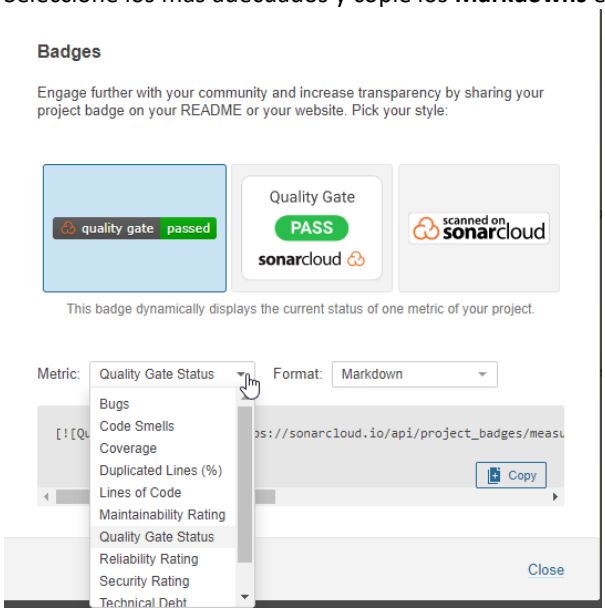


Según este ejemplo encontró en el controlador 3 elementos duplicados, pero si el desarrollador determina que este segmento de código no es un problema puede cambiar el estatus a un Falso Positivo o si es el caso contrario puede aceptar la sugerencia. Ver la siguiente figura.

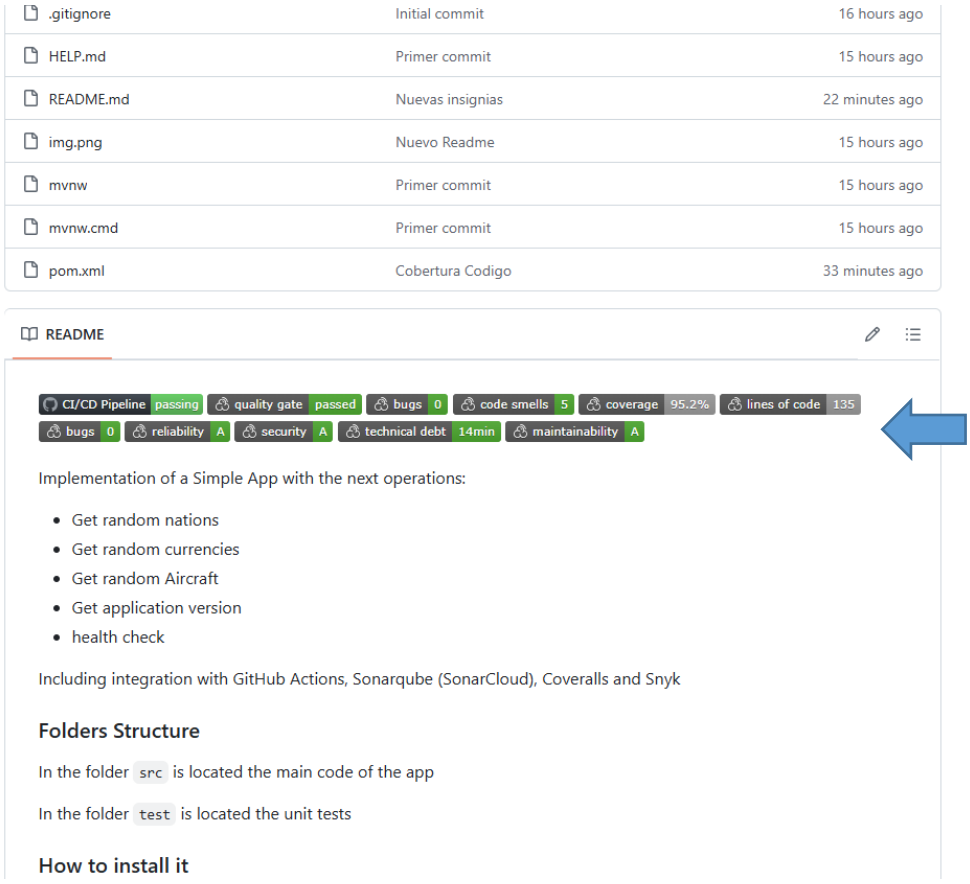


Ya sólo nos faltaría añadir los distintivos de Sonar a nuestro proyecto en GitHub, al igual que hicimos con la integración con GitHub Actions.

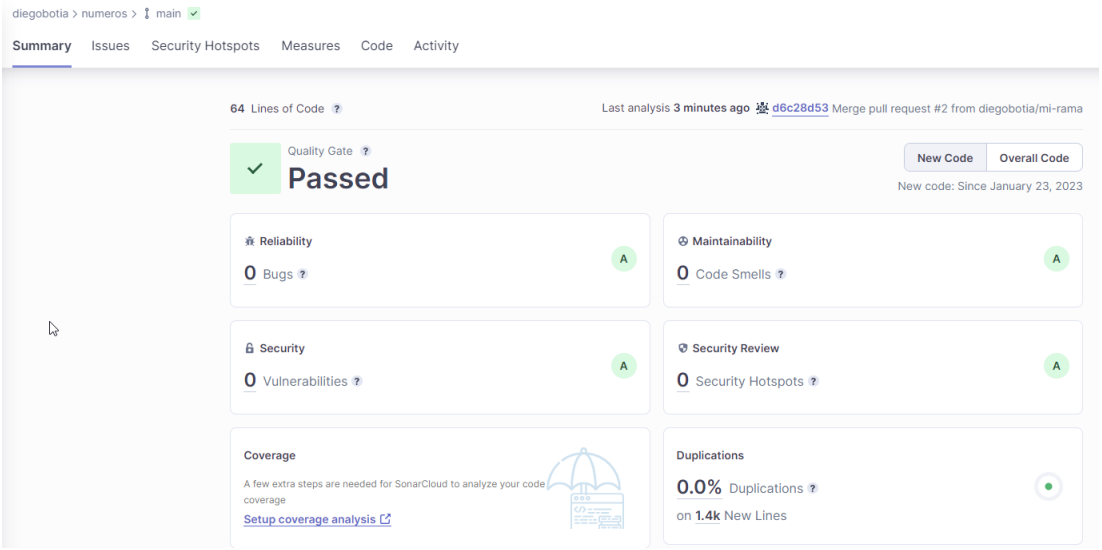
En su proyecto puede encontrarlos en el menú de la izquierda en la opción **Information** y luego en la opción **Badges.** Seleccione los más adecuados y cópie los **Markdowns** en su archivo Readme del proyecto.



Ahora si subimos el proyecto a GitHub, podremos ver los distintivos del test unitario que ya teníamos y los nuevos que hemos añadido de Sonarqube.



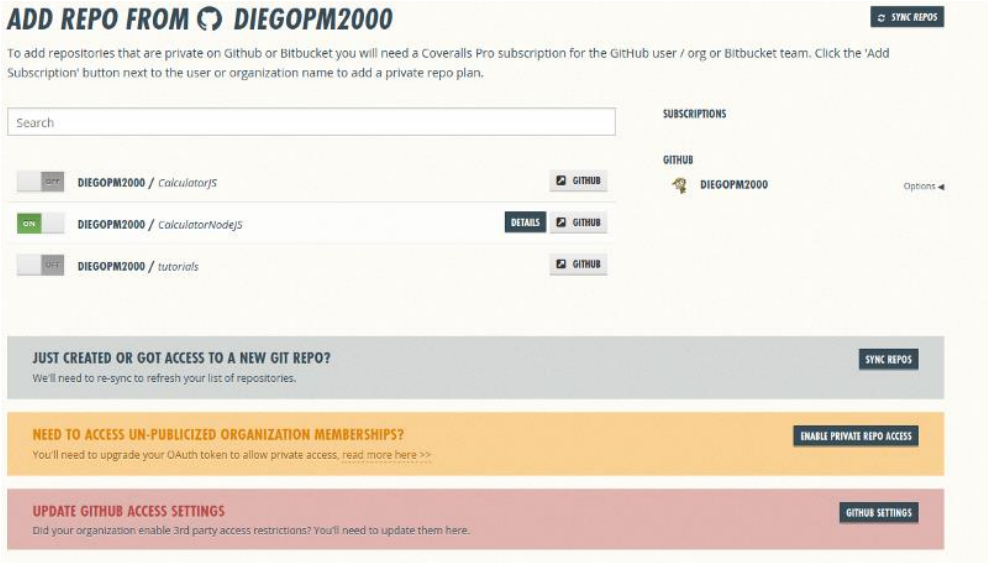
SI hacemos un Merge hacia la rama principal podemos observar que en Sonar ya aparecen los cambios exitosamente y el Quality Gate en estado Passed.



Los test de cobertura, también pueden visualizarse en la plataforma Coveralls.

Esta plataforma nos permite, al igual que Sonarcloud, visualizar online el resultado de nuestros test de cobertura.

Entrando en la plataforma, asociamos nuestra cuenta de GitHub, y asociamos nuestro proyecto.



Tarea: Puede integrar a Coveralls con GitHub Actions

Construcción del artefacto JAR

Después del trabajo de prueba unitaria y el trabajo de sonar, agregaremos otro trabajo para construir nuestra aplicación y generar el archivo jar

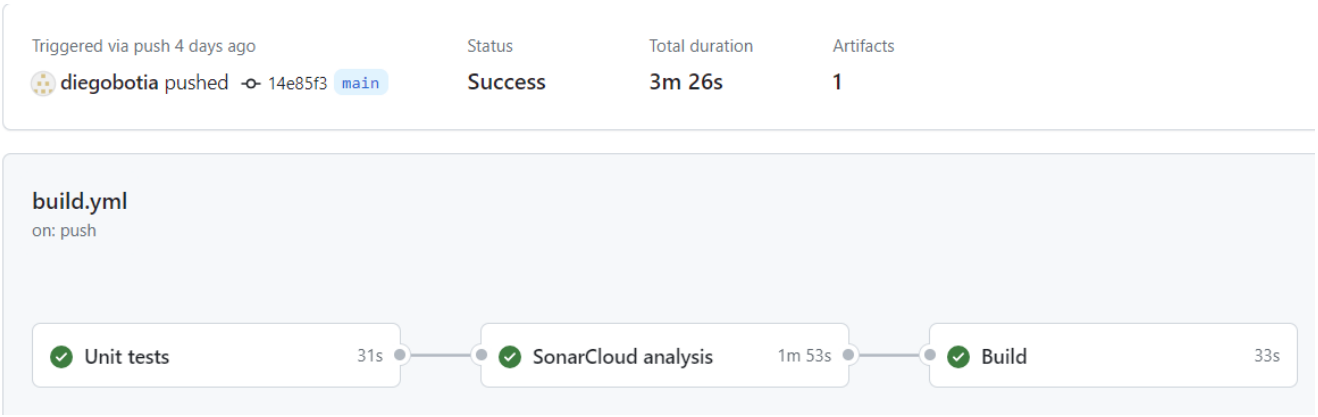
```
#Build's job
  build:
    #Depends on sonar's job
    needs: sonar
    name: Build
    #Run on Ubuntu using the latest version
    runs-on: ubuntu-latest
    steps:
      #Check-out your repository under $GITHUB_WORKSPACE, so your workflow can access it
      - uses: actions/checkout@v1
      #Set up JDK 11
      - name: Set up JDK
        uses: actions/setup-java@v1
        with:
          java-version: '11'
      #Set up Maven cache
      - name: Cache Maven packages
        #This action allows caching dependencies and build outputs to improve workflow
execution time.
        uses: actions/cache@v3
        with:
          path: ~/.m2
          key: ${{ runner.os }}-m2-${{ hashFiles('**/pom.xml') }}
          restore-keys: ${{ runner.os }}-m2
      #Build the application using Maven
      - name: Build with Maven
        run: mvn -B package -DskipTests --file pom.xml
      #Build the application using Maven
      - name: Upload JAR
        #This uploads artifacts from your workflow allowing you to share data between jobs
and store data once a workflow is complete.
        uses: actions/upload-artifact@v4
        with:
          #Set artifact name
```

```
name: artifact
#From this path
path: target/faker-0.0.1-SNAPSHOT.jar
```

Este **build** construirá la aplicación usando **maven** y cargará el archivo jar.

El paso de carga de artefactos nos permite compartir datos entre trabajos y almacenar datos una vez que se completa un flujo de trabajo. En nuestro caso, usaremos el archivo jar cargado en el trabajo Deploy.

Ahora en GH Actions veremos el pipeline completo



Despliegue del artefacto en Docker Hub

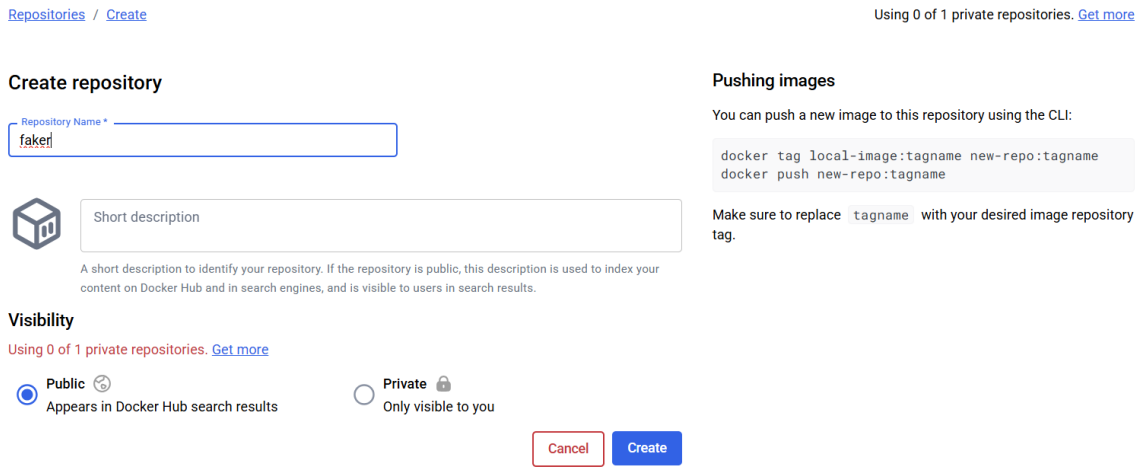
Crear un archivo **Dockerfile** en la raíz de su proyecto y coloque el siguiente contenido. Verifique que el motor de Docker este activo en su PC.

```
FROM openjdk:11
EXPOSE 8080
ADD target/faker.jar faker.jar
ENTRYPOINT ["java","-jar","/faker.jar"]
```

Este Dockerfile se encargará de crear una imagen Docker con su aplicación empaquetada.

Ingrese a Docker Hub (https://hub.docker.com) y cree una nueva cuenta.

Una vez acceda cree un nuevo repositorio



GitHub Secrets: Debe agregar sus credenciales de Docker Hub como secretos en su repositorio de GitHub.

Vaya a su repositorio en GitHub.
Haz clic en Settings > Secrets and variables > Actions.
Agregue los siguientes secretos:
DOCKER_USERNAME: Su nombre de usuario de Docker Hub.
DOCKER_PASSWORD: Su contraseña de Docker Hub.

Configurar GitHub Actions Workflow

Agregue las siguientes líneas a su archivo .yml

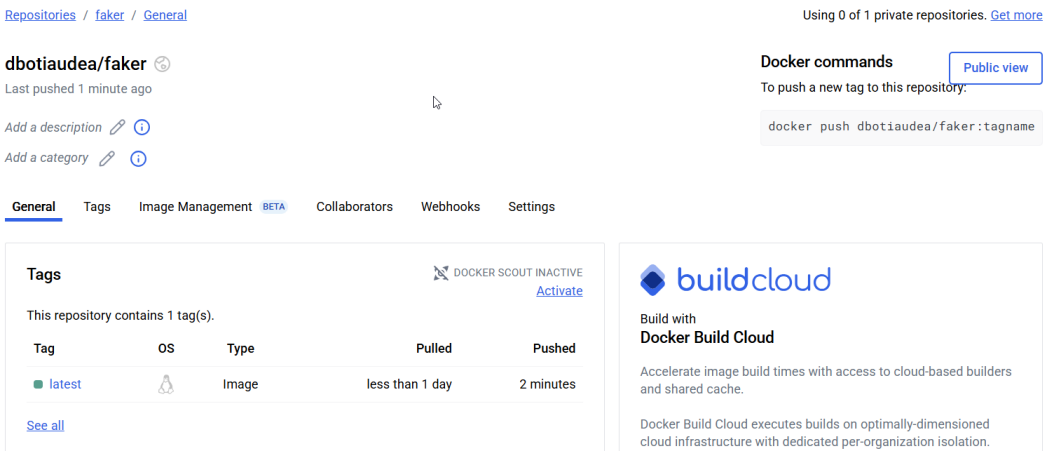
```
- name: Build & push Docker image
  uses: mr-smithers-excellent/docker-build-push@v6
  with:
      image: dbotiaudea/gitlab
      tags: latest
      registry: docker.io
      dockerfile: Dockerfile
      username: ${{ secrets.DOCKER_USERNAME }}
      password: ${{ secrets.DOCKER_PASSWORD }}
```

Este archivo hará lo siguiente:
1. Compilará el JAR con Maven.
2. Construirá la imagen Docker.
3. Subirá la imagen a Docker Hub.

Agregue al archivo pom.xml en la sección plugins al final lo siguiente:

```
    </plugins>
  <finalName>gitlab</finalName>
</build>
```

Proceda a actualizar el repositorio de github con los nuevos cambios. Una vez se ejecute GitHub Actions verifique que en Docker Hub se encuentre cargada la nueva imagen:



Abra una nueva terminal y escriba el siguiente comando para descargar la imagen generada.

C:\Users\Diego\Downloads\laboratorio2\fake>**docker pull dbotiaudea/faker:latest**
latest: Pulling from dbotiaudea/faker
001c52e26ad5: Already exists
d9d4b9b6e964: Already exists
2068746827ec: Already exists
9daef329d350: Already exists
d85151f15b66: Already exists
66223a710990: Already exists
db38d58ec8ab: Already exists
ebe6f9031d47: Pull complete
Digest: sha256:baf70d95b15545df5eb93cf6805172da7938ee08fad05b855e99917877970241
Status: Downloaded newer image for dbotiaudea/faker:latest
docker.io/dbotiaudea/faker:latest

What's next:
View a summary of image vulnerabilities and recommendations → docker scout quickview dbotiaudea/faker:latest

Ejecutar el contenedor desde Docker Hub
Después de que la imagen se haya descargado exitosamente desde Docker Hub, puedes ejecutarla en cualquier máquina que tenga Docker instalado con el siguiente comando:

C:\Users\Diego\Downloads\github-app\github-app>docker run -p 8080:8080 dbotiaudea/faker

```
  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::                (v2.7.17)
```

2024-09-17 17:01:21.279 INFO 1 --- [main] com.udea.consulta.ConsultaApplication : Starting ConsultaApplication v0.0.1-SNAPSHOT using Java 11.0.16 on 8cc786f740b9 with PID 1 (/faker.jar started by root in /)
2024-09-17 17:01:21.284 INFO 1 --- [main] com.udea.consulta.ConsultaApplication : No active profile set, falling back to 1 default profile: "default"

```
2024-09-17 17:01:22.225  INFO 1 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat initialized with port(s): 8080 (http)
2024-09-17 17:01:22.240  INFO 1 --- [           main] o.apache.catalina.core.StandardService   : Starting service [Tomcat]
2024-09-17 17:01:22.241  INFO 1 --- [           main] org.apache.catalina.core.StandardEngine  : Starting Servlet engine: [Apache Tomcat/9.0.82]
2024-09-17 17:01:22.333  INFO 1 --- [           main] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring embedded WebApplicationContext
2024-09-17 17:01:22.333  INFO 1 --- [           main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 981 ms
2024-09-17 17:01:22.676  INFO 1 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port(s): 8080 (http) with context path ''
2024-09-17 17:01:22.687  INFO 1 --- [           main] com.udea.consulta.ConsultaApplication    : Started ConsultaApplication in 2.114 seconds (JVM running for 2.882)
2024-09-17 17:01:46.206  INFO 1 --- [nio-8080-exec-3] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring DispatcherServlet 'dispatcherServlet'
2024-09-17 17:01:46.214  INFO 1 --- [nio-8080-exec-3] o.s.web.servlet.DispatcherServlet        : Initializing Servlet 'dispatcherServlet'
2024-09-17 17:01:46.231  INFO 1 --- [nio-8080-exec-3] o.s.web.servlet.DispatcherServlet        : Completed initialization in 17 ms
```

Esto ejecutará la aplicación empaquetada en Docker

Nota: Verifique con el comando docker image ls que se haya descargado la imagen correctamente.

```
C:\Users\Diego\Downloads\laboratorio2\fake>docker image ls
REPOSITORY                                TAG               IMAGE ID       CREATED          SIZE
dbotiaudea/faker                          latest            7d20dd215f19   10 minutes ago   674MB
```

Integración con GitHub Actions por medio de nuevas ramas (branches).

Vamos a integrar nuestro proyecto en GitHub Actions a través de la creación de ramas que nos permitirá hacer el proceso de integración continua.

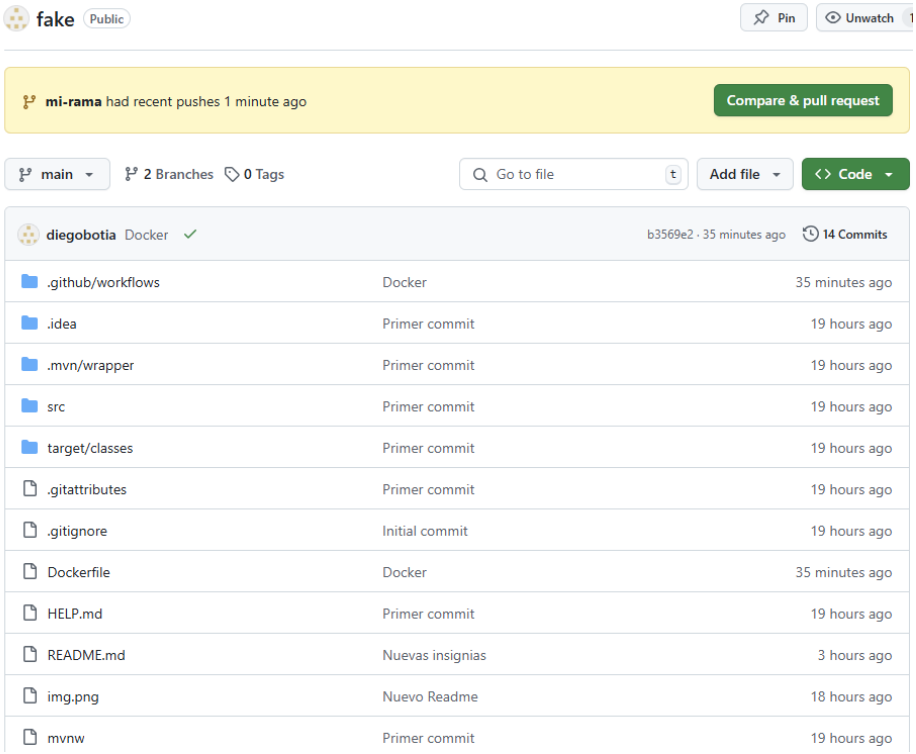
Lo primero que haremos será crear una nueva rama a parte de la rama principal (main). En este caso crearemos por ejemplo la rama **mi-rama**. Si lo desea puede crearla con cualquier otro nombre. Ejecute el siguiente comando:

```
Diego@LAPTOP-MQSPSDUN MINGW64 ~/Downloads/laboratorio2/fake (main)
$ git checkout -b mi-rama
Switched to a new branch 'mi-rama'

Diego@LAPTOP-MQSPSDUN MINGW64 ~/Downloads/laboratorio2/fake (mi-rama)
$
```

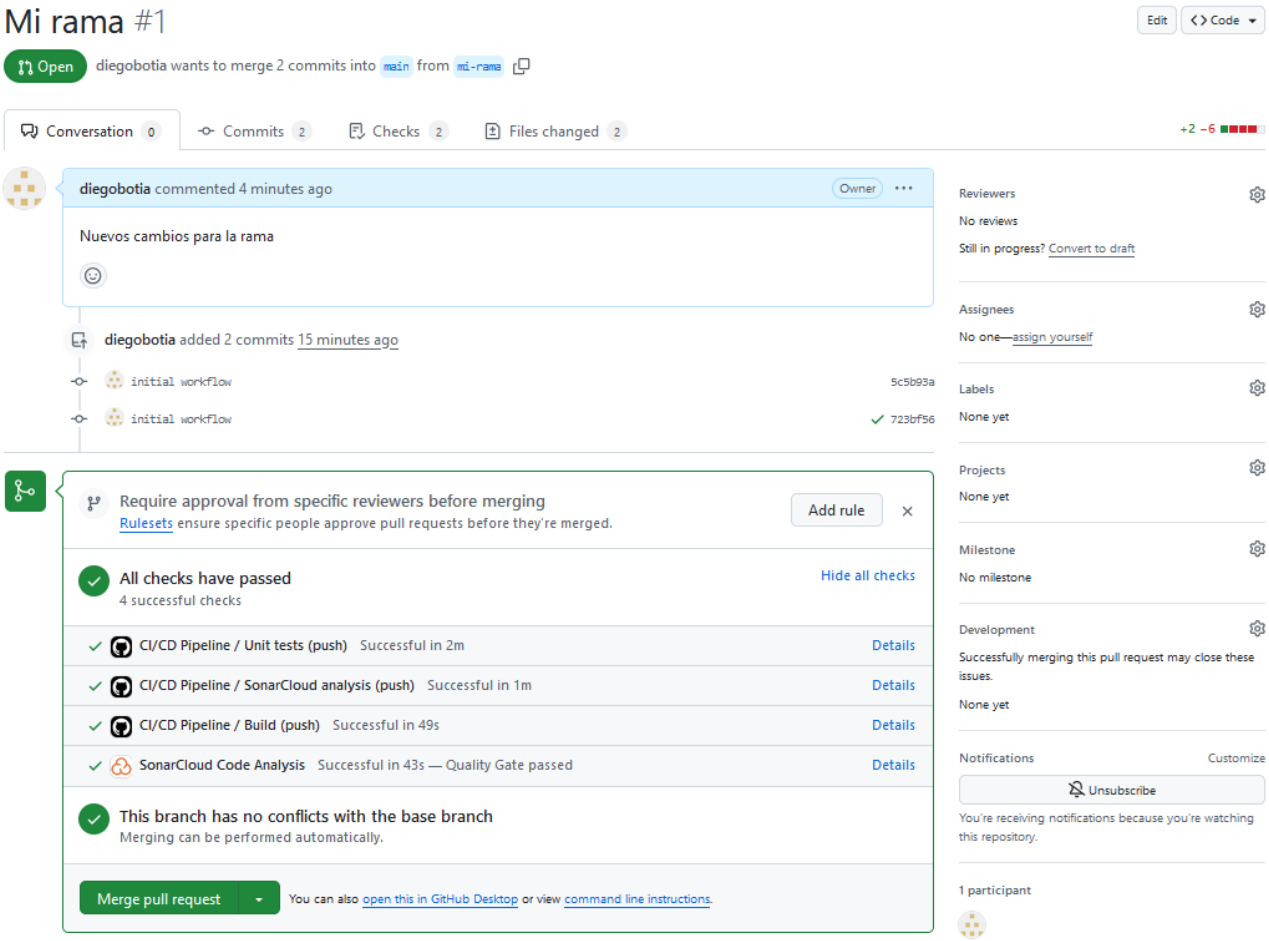
Ahora subiremos los cambios a nuestro repositorio de GitHub usando los siguientes comandos:

```
MINGW64:/c/Users/Diego/Downloads/laboratorio2/fake
Diego@LAPTOP-MQSPSDUN MINGW64 ~/Downloads/laboratorio2/fake (mi-rama)
$ git status
On branch mi-rama
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   pom.xml
        modified:   src/test/java/com/udea/faker/FakerApplicationTests.java

no changes added to commit (use "git add" and/or "git commit -a")

Diego@LAPTOP-MQSPSDUN MINGW64 ~/Downloads/laboratorio2/fake (mi-rama)
$ git commit -am 'initial workflow'
warning: LF will be replaced by CRLF in pom.xml.
The file will have its original line endings in your working directory
warning: LF will be replaced by CRLF in src/test/java/com/udea/faker/FakerApplic
ationTests.java.
The file will have its original line endings in your working directory
[mi-rama 61e10ee] initial workflow
 2 files changed, 1 insertion(+), 3 deletions(-)

Diego@LAPTOP-MQSPSDUN MINGW64 ~/Downloads/laboratorio2/fake (mi-rama)
$ git push --set-upstream origin mi-rama
Enumerating objects: 19, done.
Counting objects: 100% (19/19), done.
Delta compression using up to 8 threads
Compressing objects: 100% (6/6), done.
Writing objects: 100% (10/10), 668 bytes | 668.00 KiB/s, done.
Total 10 (delta 3), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (3/3), completed with 3 local objects.
remote:
remote: Create a pull request for 'mi-rama' on GitHub by visiting:
remote:      https://github.com/diegobotia/fake/pull/new/mi-rama
remote:
To https://github.com/diegobotia/fake.git
 * [new branch]      mi-rama -> mi-rama
branch 'mi-rama' set up to track 'origin/mi-rama'.

Diego@LAPTOP-MQSPSDUN MINGW64 ~/Downloads/laboratorio2/fake (mi-rama)
$ git add .

Diego@LAPTOP-MQSPSDUN MINGW64 ~/Downloads/laboratorio2/fake (mi-rama)
$ git commit -am 'initial workflow'
On branch mi-rama
Your branch is up to date with 'origin/mi-rama'.

nothing to commit, working tree clean

Diego@LAPTOP-MQSPSDUN MINGW64 ~/Downloads/laboratorio2/fake (mi-rama)
$ git push origin mi-rama
Everything up-to-date
```

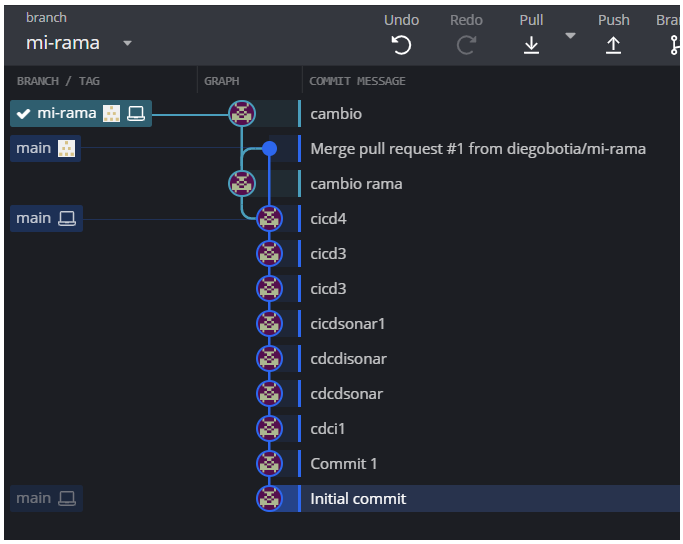
Ahora al revisar los cambios en nuestro repositorio observamos que nos presenta la nueva rama y nos da la opción de **Compare & Pull request** como se presenta a continuación.



Podremos observar que las acciones de GH Actions ya se ejecutaron exitosamente en nuestra aplicación dentro de la nueva rama.



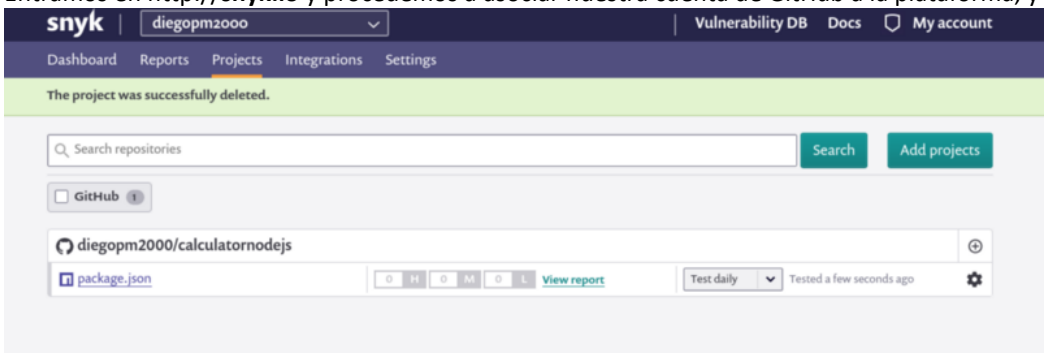
Podrá ver en Git Kraken el resumen de todo lo que se ha trabajo en GH desde el proyecto.



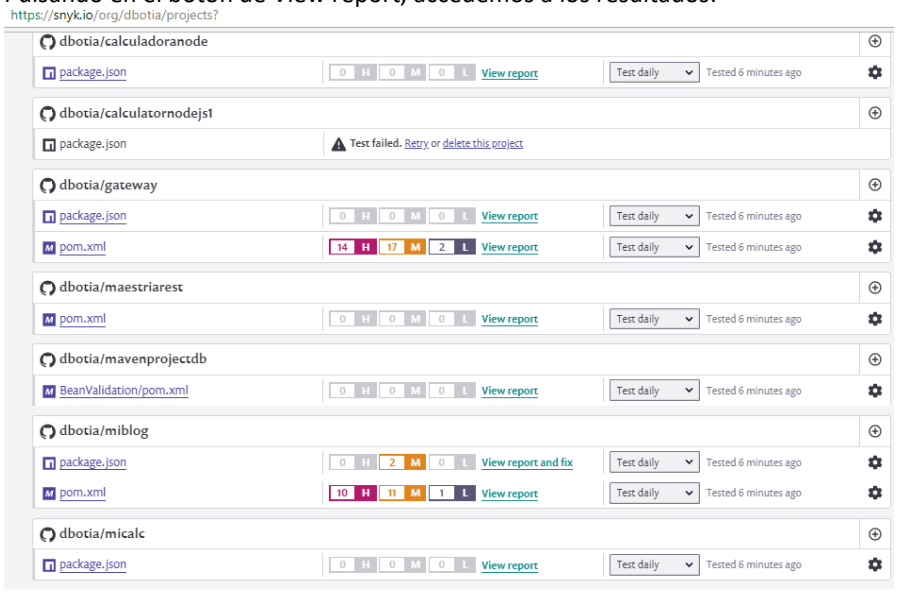
Conexión con Snyk

Para concluir vamos a asociar el proyecto también a **Snyk**, una plataforma que nos va a servir también para detectar posibles vulnerabilidades de seguridad en el código, incluyendo las librerías asociadas.

Entramos en http://**snyk.io** y procedemos a asociar nuestra cuenta de GitHub a la plataforma, y el proyecto.



Pulsando en el botón de View report, accedemos a los resultados:

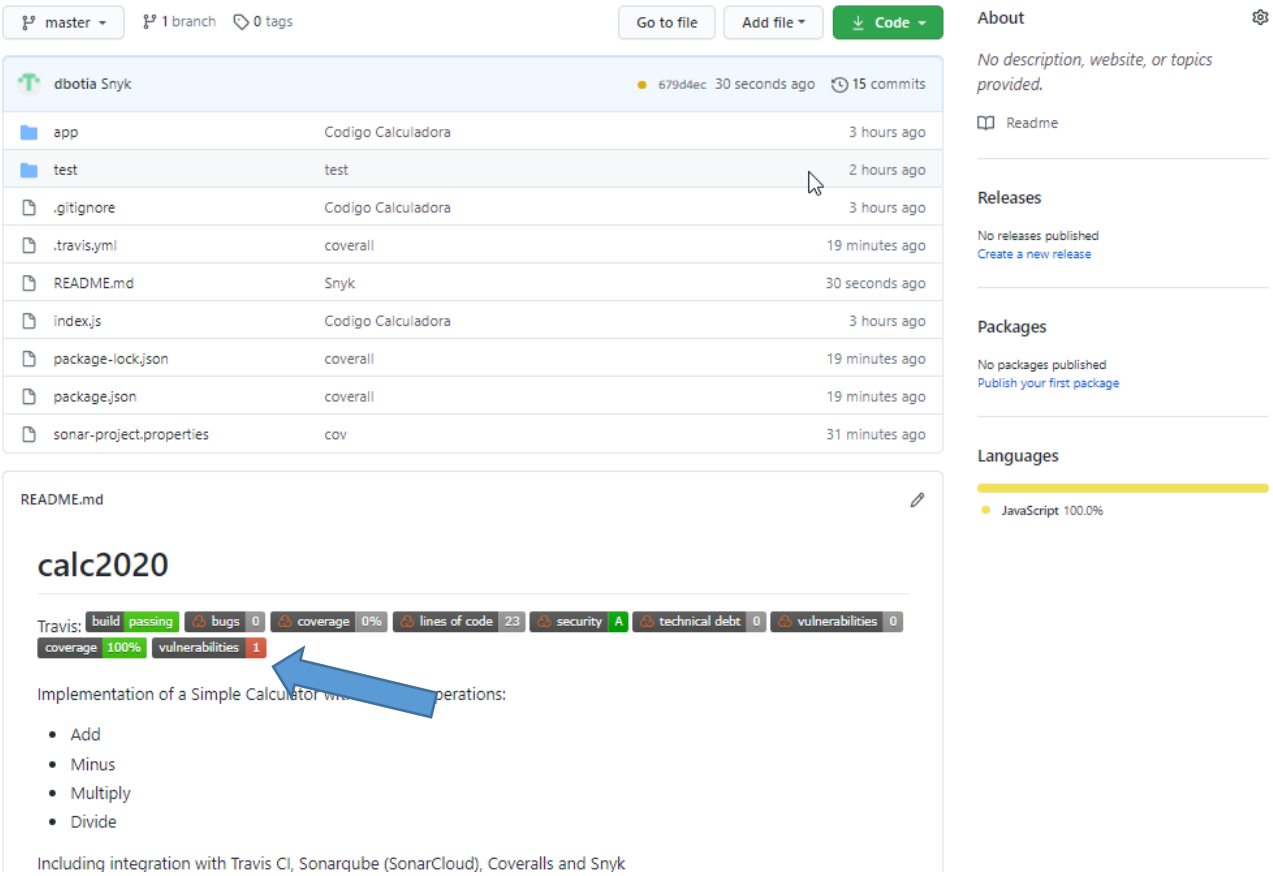


Y desde aquí también podemos obtener el distintivo para Snyk

[![Known Vulnerabilities](https://snyk.io/test/github/dbotia/faker/badge.svg)](https://snyk.io/test/github/dbotia/faker)

Que añadiremos al README.md del proyecto.

Al final tendremos todo conectado así:



Ejercicio de entrega Opcional Fecha de Entrega 24 de Octubre de 2025

De acuerdo al reto de la Fábrica de Software que su grupo esta realizando realice lo siguiente:

- Completar el pipeline de despliegue agregando la conexión a alguno proveedor de nube (AWS / Azure DevOps / GCP, etc).
- Debe modificar el script (build.yml) para que se despliegue el artefacto JAR en un contenedor de aplicaciones según el proveedor Cloud que elija.
- Debe anexar el paso a paso que siguió en el proceso y las pruebas de ejecución del back sobre la nube.