Bloque 13. Testing con JUnit (Avanzado)

Testing en SpringBoot - JUnit, Mockito y RestAssured

Esta guía te brindará la asistencia necesaria para desarrollar pruebas unitarias efectivas con JUnit y Mockito en tus proyectos de Spring Boot. Vamos a explorar tres enfoques distintos para la creación de estas pruebas.

Enfoque Básico con JUnit:

Comenzaremos configurando las pruebas unitarias con JUnit en tu proyecto Spring Boot. Para ello, asegúrate de incluir las dependencias necesarias en tu archivo de configuración, como el pom.xml en el caso de Maven. Podremos utilizar notaciones como **@SpringBootTest** para aprovechar las funcionalidades de Spring Boot durante nuestras pruebas.

Posteriormente, procederemos a escribir pruebas unitarias para clases y métodos específicos. Emplearemos las aserciones proporcionadas por JUnit para verificar que el comportamiento de nuestro código cumpla con las expectativas.

También podemos nombrar el orden de test **@TestMethodOrder(MethodOrdered.MethodName.class)** o **@TestMethodOrder(OrderAnnotation.class) @Order(n)**

Enfoque Avanzado con Mockito:

En este siguiente nivel, incorporaremos la biblioteca Mockito para realizar pruebas más avanzadas. Mockito facilita la simulación y control de objetos en nuestras pruebas. Asegúrate de configurar las dependencias necesarias en tu archivo de configuración.

- Utilizaremos anotaciones como **@Mock** y **@InjectMocks** para crear simulacros de objetos y gestionar automáticamente las inyecciones de dependencias. Definiremos el comportamiento esperado de estos objetos simulados mediante métodos como **when()** y **.thenReturn()**, permitiéndonos establecer escenarios específicos para nuestras pruebas.
- Trataremos el manejo de excepciones mediante **@assertThrow** o **try/catch**.
- Por último, mencionaremos los diferentes mecanismos que existen para ejecutar pruebas condicionales o deshabilitar un test. (**@Disabled**, **@EnabledOnOs**, **@EnabledIfSystemProperty**)
- Además, aprenderemos a verificar interacciones utilizando métodos como **verify()**, asegurándonos de que los métodos deseados sean invocados durante la ejecución de nuestras pruebas.
- Aprenderemos a usar **ArgumentCaptor**, que nos servirá para poder obtener los parámetros que se le pasan a un método moqueado.

Uso de una base de datos temporal para tests:

Para realizar un testing más avanzado, crearemos una base de datos temporal, con la que podremos realizar pruebas más completas y sin tener que estar moqueando a mano todos los repositorios que queramos probar.

Enfoque Básico con RestAssured:

Además de Mockito, usaremos RestAssured, para validar el correcto funcionamiento de todos los endpoints que creemos para el proyecto.

Revisaremos algunos de los métodos básicos con los que podremos probar si la información se manda y se recibe correctamente cómo **given()**.

Esta guía abarcará tanto el enfoque básico como el avanzado, brindándote las herramientas necesarias para realizar pruebas unitarias sólidas en tus aplicaciones Spring Boot.

Para los ejemplos descritos usaremos las siguientes dependencias Maven:

```
  <dependency>
     <groupId>org.springframework.boot</groupId>
     <artifactId>spring-boot-starter-data-jpa</artifactId>
   </dependency>
   <dependency>
     <groupId>org.springframework.boot</groupId>
     <artifactId>spring-boot-starter-web</artifactId>
   </dependency>

<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-test</artifactId>
  <scope>test</scope>
</dependency>

   <!-- base de datos temporal-->
   <dependency>
     <groupId>com.h2database</groupId>
     <artifactId>h2</artifactId>
     <scope>test</scope>
   </dependency>

<!-- mockito -->
```

```
    <dependency>
      <groupId>org.mockito</groupId>
      <artifactId>mockito-core</artifactId>
      <version>5.7.0</version>
    </dependency>
<!-- Rest assured -->
    <dependency>
      <groupId>io.rest-assured</groupId>
      <artifactId>rest-assured</artifactId>
      <version>5.2.0</version>
      <scope>test</scope>
    </dependency>
```

JUnit

JUnit destaca como el marco de pruebas de unidades más popular en el mundo Java. En proyectos extensos, que a menudo cuentan con más de 2000 archivos fuente o incluso llegan a los 10000 archivos con un millón de líneas de código, JUnit se convierte en una herramienta fundamental.

Antes de la implementación de pruebas unitarias, la validación de la aplicación solía depender de la ejecución completa y la verificación manual de la interfaz de usuario. Sin embargo, este enfoque resulta ineficiente. Aquí es donde entra en juego el Unit Testing, una práctica centrada en la creación de pruebas automatizadas para evaluar clases y métodos de manera individual.

JUnit se presenta como un marco que facilita la ejecución de métodos y la verificación automática (o afirmación) de que la salida coincide con la esperada. La clave de las pruebas automatizadas radica en su capacidad para ejecutarse con integración continua, permitiendo que las pruebas se realicen tan pronto como se realicen cambios en el código.

Ejemplo de código fuente a probar:

```java
public class MyMath {
  int sum(int[] numbers) {
    int sum = 0;
    for (int i : numbers) {
      sum += i;
    }
    return sum;
  }
}
```

Prueba unitaria para el método de la suma:

```java
class MyMathTest {
  MyMath myMath = new MyMath();

  /**
   * MyMath.sum
   * 1,2,3 => 6
   */
  @Test
  void sum_with3numbers() {
    System.out.println("Test1");
    assertEquals(6, myMath.sum(new int[]{1, 2, 3}));
  }

  @Test
  void sum_with1number() {
    System.out.println("Test2");
    assertEquals(3, myMath.sum(new int[]{3}));
  }

  @Test
  void sum_with0number() {
    System.out.println("Test3");
    assertEquals(0, myMath.sum(new int[]{}));
  }
}
```

Otras anotaciones importantes de JUnit:

Las anotaciones @BeforeEach y @AfterEach en JUnit permiten la ejecución de métodos antes y después de cada método de prueba en la clase, respectivamente. Estas anotaciones son útiles para realizar configuraciones previas a la prueba o limpieza después de la prueba en un entorno controlado.

Anotaciones @BeforeAll y @AfterAll

Por otro lado, las anotaciones **@BeforeAll** y **@AfterAll** se aplican a métodos estáticos y se ejecutan una vez antes y después de la actual clase de prueba. Estos métodos son ideales para llevar a cabo configuraciones o acciones que deben realizarse una sola vez para toda la suite de pruebas de esa clase.

Estas anotaciones proporcionan un control preciso sobre el entorno de prueba, permitiendo la preparación y limpieza adecuadas antes y después de las pruebas unitarias. Utilizarlas con sabiduría puede mejorar la eficiencia y la consistencia de las pruebas en un proyecto JUnit.

```
class MyMathTest {
  MyMath myMath = new MyMath();

  @BeforeAll
  public static void beforeClass() {
     System.out.println("Before Class");
     //initialize database
     //load properties of file or rute
  }

  @AfterAll
  public static void afterClass() {
     System.out.println("After Class");
     //close database connection
  }

  //Rest of code
}
```

También podremos usar la notaciones **@BeforeEach** y **@AfterEach**, que se ejecutan antes de cada test de la clase donde se defina.

Añadiendo los siguientes métodos podremos inicializar para cada test la variable myMath, esto es útil, cuando dentro de la propia variable cambian los datos y necesitamos volver a instanciarlos para otro test.

Ejemplo:

```
MyMath myMath = null;

@BeforeEach
public void beforeEach() {
  System.out.println("Before test");
  //instance new data
  myMath = new MyMath();
}

@AfterEach
public void afterEach() {
  System.out.println("After Test");
  //clean data
  myMath = null;
}
```

Ordenación de tests

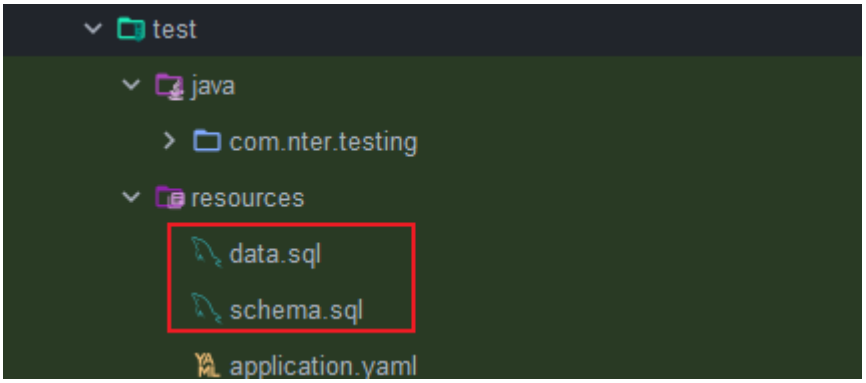
```
@SpringBootTest
@TestMethodOrder(MethodOrderer.OrderAnnotation.class)
class CountryServiceImplOrderTest {

  @Autowired
  private CountryServiceImpl countryService;

  //El order es vital, ya que en una ejecución sin el orden correcto puede provocar que
  se realice la prueba de findByName sobre un objeto que aún no existe.

  @Test
  @Order(2)
  void testSaveCountry() {
     CountryJpa country = new CountryJpa();
     country.setName("Spain");

     CountryJpa savedCountry = countryService.saveCountry(country);
     assertNotNull(savedCountry);
     assertEquals(country.getName(), savedCountry.getName(), "El nombre del país
     guardado no coincide con el esperado");
  }
```

```java
@Test
@Order(3)
void testFindCountryByName() {
    CountryJpa foundCountry = countryService.findCountryByName("Spain");

    assertNotNull(foundCountry);
    assertEquals("Spain", foundCountry.getName(), "El nombre del país encontrado no
    coincide con el esperado");
}

@Test
@Order(1)
void testFindCountryByNameFails() {
    assertThrows(CountryNotFoundException.class, () ->
    countryService.findCountryByName("Spain"));
}
}
```

La etiquetación @Order de JUnit está orientada a organizar el orden de ejecución de test unitarios, ésto es necesario usarlo en casos donde haya test que tengan dependencia de la ejecución de pruebas previamente.

Uso de bbdd temporal en tests

Para poder probar de manera más completa nuestra aplicación, podemos hacer uso de una base de datos temporal, que nos permitirá evitar mockear todos los repositorios, y además probar que nuestro modelo JPA sea correcto.
Para poder crear una base de datos temporal, necesitamos crear los ficheros data.sql y schema.sql, conteniendo en schema.sql la creación de las tablas, precedido de su drop, para regenerar la base de datos en cada ejecución, y en el otro los inserts.
Y se ve de la siguiente manera:



schema.sql

```
DROP TABLE IF EXISTS mstr_country;
CREATE TABLE mstr_country
(
   id INTEGER auto_increment PRIMARY KEY,
   name VARCHAR(10) NOT NULL
);
ALTER TABLE mstr_country ALTER COLUMN id RESTART WITH 1;
```

data.sql

```
INSERT INTO mstr_country (name) VALUES ('India');
INSERT INTO mstr_country (name) VALUES ('France');
INSERT INTO mstr_country (name) VALUES ('Italy');
```

Una vez hecho esto, podemos fijarnos que en el test, usamos la notación **@SpringBootTest** con la que indicamos que cree una instancia de SpringBoot y así creando la base de datos temporal y con ello podemos instanciar el servicio **CountryServiceImpl** como un Bean.
Es importante tener en cuenta que la bbdd se creará al principio, de todos los tests a ejecutar, y se eliminará con el último tests, para poder gestionar y evitar fallos, podemos ordenar los tests que puedan entrar en conflicto, o bien usar DirtiesContext para poder regenerar la instancia de springBoot y con ello los datos de BBDD.

Video tutorial externo - JUnit 5

Mockito

Mockito se destaca como el marco de simulación más ampliamente utilizado en el ámbito de Java.

En el siguiente ejemplo, consideremos el caso de **CountryServiceImpl** que depende de **CountryRepository**. Al escribir una prueba de unidad para **CountryServiceImpl**, surge la necesidad de utilizar un servicio de datos simulado, evitando así la conexión a una base de datos real.

Mockito proporciona la capacidad de crear simulacros (mocks) para objetos, permitiendo la simulación de interacciones con dependencias. En este contexto, podríamos simular el comportamiento del servicio de datos, asegurándonos de que las pruebas se centren exclusivamente en la lógica de **CountryServiceImpl** sin depender de la implementación real de **CountryRepository**.

Este enfoque de simulación con Mockito se convierte en una herramienta valiosa para crear pruebas unitarias más controladas y específicas, mejorando la modularidad y la eficiencia del proceso de prueba en entornos Java.

Uso básico de Mockito

Creamos el Repositorio:

```
@Repository
public interface CountryRepository extends JpaRepository<CountryJpa, Integer> {
}
```

Crearemos un Servicio que tenga de dependencia un Repositorio de JPA:

```
@Service
public class CountryServiceImpl implements ICountryService {

  private final CountryRepository countryRepository;

  public CountryServiceImpl(CountryRepository countryRepository) {
     this.countryRepository = countryRepository;
  }

  @Override
  public List<CountryJpa> findAllCountriesSorted() {
     return countryRepository.findAll(Sort.by(Sort.Direction.ASC, "name"));
  }
}
```

Escribiendo una prueba con Mockito:

```
@ExtendWith(MockitoExtension.class)
class CountryServiceImplTest {

  @Mock
  private CountryRepository countryRepository;

  @InjectMocks
  private CountryServiceImpl countryService;

  @Test
  void findAllCountriesSorted() {
     List<CountryJpa> returnData = List.of(
           new CountryJpa()
     );
     when(countryRepository.findAll(any(Sort.class))).thenReturn(returnData);
     List<CountryJpa> result = countryService.findAllCountriesSorted();
     assertEquals(returnData, result);
     verify(countryRepository, times(1)).findAll(any(Sort.class));
  }
```

```
}
```

En la prueba podemos ver los siguientes puntos:

- Usamos la notación **ExtendWith** para indicar que usamos Mockito.
- La notación **Mock** indica que la variable se debe inicializar como una clase simulada sin necesidad de ser inyectada como bean.
- La notación **InjectMocks** crea el servicio de manera simulada, inyectando las dependencias que tenga como Mocks, y usando el Mock anteriormente declarado.
- Usamos **when()** para cuando se llame al repositorio, en este ejemplo, al método **findAll** con un parámetro Sort, devuelva los datos que hemos definido, usando el método **thenReturn**.
- Con el metodo **verify()** comprobamos el número de ejecuciones que ha tenido el mock countryRepository para el metodo findAll, para la sobrecarga con Sort.

Cuando usamos When(), si la función que queremos simular se llama con parámetros, debemos indicar los valores exactos que sabemos que van a llegar al mock, o usar Matchers como any() o eq(), para indicar cualquier valor y con ello especificar que sobrecarga del método es el que se debe usar.

Uso de @assertThrow y manejo de excepciones en tests

```
@Test
void findCountriesByName() {
  when(countryRepository.findByName(anyString())).thenReturn(Optional.empty());
  // Verifica que se lance la excepción cuando se intenta obtener el país que no existe
  assertThrows(CountryNotFoundException.class, () ->
countryService.findCountryByName("Spain"));
}
```

Uso de Argument captor

ArgumentCaptor sirve para poder capturar los datos que se le pasan a un método simulado, esto es conveniente, cuando queramos obtener el resultado de alguna operación/ casuística, pero no podamos acceder a él, ya que no se devuelva en el método que estamos probando.

```java
@ExtendWith(MockitoExtension.class)
class CountryServiceImplArgumentCaptorTest {

  private final ArgumentCaptor<Sort> argumentCaptor =
ArgumentCaptor.forClass(Sort.class);
  @Mock
  private CountryRepository countryRepository;
  private CountryServiceImpl countryService;

  @BeforeEach
  void initializeService() {
    countryService = new CountryServiceImpl(countryRepository);
  }

  @Test
  void findAllCountriesSorted() {
    List<CountryJpa> returnData = List.of(
        new CountryJpa()
    );
    when(countryRepository.findAll(argumentCaptor.capture())).thenReturn(returnData);
    List<CountryJpa> result = countryService.findAllCountriesSorted();
    Sort sort = argumentCaptor.getValue();
    assertEquals(Sort.by(Sort.Direction.ASC, "name"), sort);
    assertEquals(returnData, result);
  }
}
```

En este ejemplo, para el método de findAll de country repository, le incorporamos el captor, con el método capture(), que previamente hemos declarado como tipo Sort.
Una vez ejecutado el método que queremos probar, y se haya ejecutado el método que hemos simulado, obtenemos el valor con el que se ha llamado, y posteriormente comprobamos que es el correcto.
[Using Mockito ArgumentCaptor | Baeldung]

Inyectando Mockito Mocks en Spring Beans

A continuación, explicaremos cómo emplear la inyección de dependencia para incorporar simulacros de Mockito en Spring Beans con el propósito de realizar pruebas unitarias.

En escenarios del mundo real, donde los componentes a menudo dependen del acceso a sistemas externos, resulta crucial proporcionar un adecuado aislamiento de prueba. Este aislamiento nos permite enfocarnos en verificar la funcionalidad de

una unidad específica sin la necesidad de involucrar toda la jerarquía de clases en cada prueba.

La inyección de un simulacro se presenta como una estrategia limpia para introducir este aislamiento necesario.

Partiendo de la arquitectura y lógica planteada en los casos anteriores, definimos la siguiente clase de configuración.

```
@Profile("test")
@Configuration
public class CountryControllerTestConfiguration {

  @Bean
  @Primary
  public ICountryService nameService() {
     return Mockito.mock(ICountryService.class);
  }
}
```

La anotación @Profile le dice a Spring que aplique esta configuración solo cuando el perfil de "prueba" esté activo. La anotación @Primary está ahí para asegurarse de que esta instancia se use en lugar de una real para el cableado automático. El método en sí mismo crea y devuelve un simulacro de Mockito de nuestra clase NameService .

Ahora podemos escribir la prueba unitaria:

```
@ActiveProfiles("test")
@SpringBootTest
class CountryControllerProfileTest {
  @Autowired
  private CountryController countryController;
  @Test
  public void findAllCountriesSorted() {

        CountryJpa country = new CountryJpa();
    country.setName("Spain");

        List<CountryJpa> countries=new ArrayList<CountryJpa>();
        countries.add(country);

    when(countryController.findAllCountriesSorted()).thenReturn(countries);

    List<CountryJpa> countriesResult=countryController.findAllCountriesSorted();
```

```
        assertEquals(countries.get(0).getName(), countriesResult.get(0).getName());
    }
}
```

Usamos la anotación @ActiveProfiles para habilitar el perfil de "test" y activar la configuración simulada que escribimos anteriormente. Como resultado, Spring conecta automáticamente una instancia real de la clase UserService , pero una simulación de la clase NameService . La prueba en sí es una prueba JUnit+Mockito bastante típica. Configuramos el comportamiento deseado del simulacro, luego llamamos al método que queremos probar y afirmamos que devuelve el valor que esperamos.

También es posible (aunque no recomendado) evitar el uso de perfiles de entorno en dichas pruebas. Para hacerlo, eliminamos las anotaciones @Profile y @ActiveProfiles y agregamos una anotación @ContextConfiguration(classes = NameServiceTestConfiguration.class)

En este breve artículo, aprenderás lo fácil que es inyectar simulacros de Mockito en Spring Beans.

Injecting Mockito Mocks into Spring Beans

Uso de RestAssured

Rest Assured es otra librería con la que podemos realizar tests, enfocada para probar los endpoints de manera más completa, y usando como hemos explicado anteriormente, una base de datos temporal.

Para este ejemplo, creamos el siguiente Controller:

```
@RestController
@AllArgsConstructor
@RequestMapping("/country")
public class CountryController {

  private final ICountryService countryService;

  @GetMapping("/all")
  public List<CountryJpa> findAllCountriesSorted() {
     return countryService.findAllCountriesSorted();
  }

  @GetMapping("/paged")
    public Page<CountryJpa> findAllPaged(@RequestParam(defaultValue = "0", required =
false) int page,
```

```
                            @RequestParam(defaultValue = "10", required = false) int pageSize)
{
    return countryService.findAllCountriesPaged(PageRequest.of(page, pageSize));
  }
}
```

y a continuación, creamos los tests:

```
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
@DirtiesContext(classMode = DirtiesContext.ClassMode.BEFORE_CLASS)
class CountryControllerTest {


  @LocalServerPort
  private int port;


  @Test
  void pagedWithParamsTests() {
    given().port(port).get("country/paged?page=1&pageSize=2").then().assertThat()
          .statusCode(200)
          .body(
                "content[0]", notNullValue(),
                "pageable.pageNumber", is(1),
                "pageable.pageSize", is(2),
                "totalElements", is(3))
          .log();
  }
}
```

A tener en cuenta:

- con la notación **SpringBootTest** indicamos que genere una instancia Spring, para los tests, con ello también generará una bbdd temporal, adicionalmente, indicamos que genere la instancia en un puerto aleatorio.
- **DirtiesContext** indica que regenerará la instancia de Spring antes de la clase, esto se ha realizado, para asegurarnos de que los datos que usemos sean los autogenerados, por ejemplo si en un tests, eliminamos un registro que se genere por defecto, y en otro tests le hacemos findById, el test fallará.
- **LocalServerPort** con esta notación recuperamos el puerto al que se le ha asignado a la instancia temporal.
- con **given()** iniciamos un tests usando RestAssured, indicamos el puerto, y con .get() indicamos que debe hacer una petición GET al servicio indicado, y

pasándole los argumentos, también se pueden realizar posts con body, incluso modificando headers.

- la primera validación que se realiza, es que el servicio devuelva 200, con este método: **statusCode**
- Luego validamos que el body que nos devuelve el servicio, tenga el formato esperado de la siguiente manera:
    - Primero se indica en formato string, que propiedad queremos validar, en nuestro caso, tenemos una propiedad content, que es una lista, al que podemos acceder a sus items, o incluso a sus funciones, usando content.size(), para comprobar su tamaño.
    - En el segundo parámetro, indicamos que matcher tendrá, en el ejemplo hemos usado is() para comprobar literales y notNullValue para indicar que haya un objeto, aunque también podemos usar validadores como equalTo() para validar strings.

Todos los ejemplos de código están disponibles en [EvaluaNter / Bloque 13 · GitLab (nfqsolutions.es)]

Diferentes métodos para hacer tests

[Testeando Spring]

SonarQube

En el mundo del desarrollo de software, la calidad es esencial. Contar con herramientas que evalúen nuestro código se vuelve crucial para asegurar un desarrollo correcto y la aplicación de buenas prácticas. Hoy exploraremos qué es SonarQube, una plataforma popular que ayuda a los desarrolladores a escribir código más limpio y seguro.

¿Qué es SonarQube?

SonarQube es una plataforma de código abierto que realiza una inspección continua de la calidad del código mediante diversas herramientas de análisis estático. Ofrece métricas que mejoran la calidad del código, permitiendo a los equipos de desarrollo hacer un seguimiento, detectar errores y vulnerabilidades de seguridad para mantener un código limpio. Es esencial en las fases de prueba y auditoría del ciclo de desarrollo, siendo perfecto para guiar a los equipos durante las revisiones de código. Soporta una inspección continua y es compatible con 29 lenguajes de programación, ampliables mediante complementos.

¿Por qué usar SonarQube?

Esta herramienta se centra en el nuevo código, facilitando la detección y solución rápida de problemas, lo que ayuda a mantener el código limpio, sencillo y fácil de leer. Los desarrolladores confían en SonarQube para lograr la integración e implementación continuas del código, no solo para detectar problemas, sino también para rastrear, controlar y verificar la calidad continua del código.

Características

- Admite los lenguajes de programación más populares.
- Realiza revisiones automáticas con análisis de código estático.
- Facilita informes objetivos sobre la calidad del proyecto con métricas y gráficos avanzados.
- Se integra con la cadena de herramientas de DevOps.
- Ampliable mediante complementos.

Beneficios de SonarQube

1. Alerta automáticamente a los desarrolladores sobre errores antes de la implementación.
2. Proporciona información detallada sobre reglas de codificación, cobertura de pruebas, duplicaciones, complejidad y arquitectura.
3. Ayuda al equipo a mejorar habilidades como programadores al hacer un seguimiento de problemas de calidad.
4. Permite la creación de paneles y filtros personalizables para centrarse en áreas clave.
5. Favorece la productividad al reducir la complejidad del código y acortar tiempos y costos adicionales al evitar cambios constantes.

Cómo Empezar a Usar SonarQube

Ahora exploraremos los pasos para instalar una instancia local de SonarQube y analizar un proyecto. La instalación local permite poner en marcha la plataforma de manera rápida y sencilla.

Para llevar a cabo la instalación de SonarQube, tienes dos opciones. En primer lugar, puedes optar por una instalación tradicional utilizando el archivo zip. Alternativamente, también puedes desplegar SonarQube mediante la ejecución de un contenedor Docker. Ambas opciones son válidas y te ofrecen flexibilidad según tus preferencias y requisitos:

- Desde el archivo zip
- Desde imagen Docker / Imagen Docker

1. Inicia el servidor ejecutando:

docker run -d --name sonarqube -e SONAR_ES_BOOTSTRAP_CHECKS_DISABLE=true -p 9000:9000 sonarqube:latest

2. Una vez que tu instancia esté en funcionamiento, inicia sesión en

http://localhost:9000 con las credenciales de administrador del sistema:

- inicio de sesión: admin
- contraseña: admin

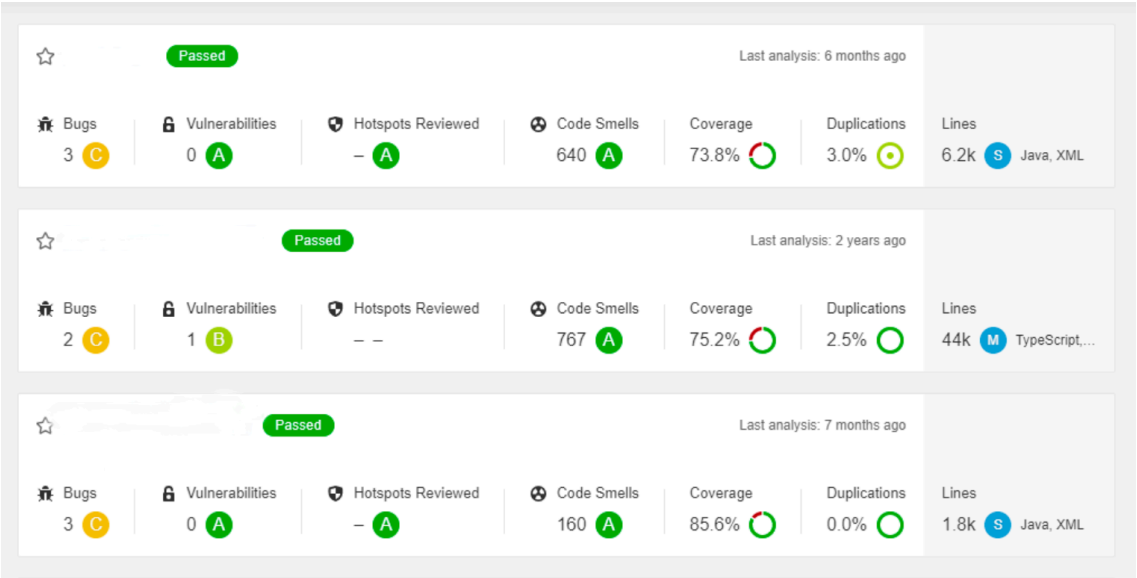
Cómo analizar un proyecto

Una vez que hayas iniciado sesión en la instancia local de SonarQube, puedes comenzar a analizar un proyecto siguiendo estos pasos:

1. Haz clic en el botón "Crear nuevo proyecto".
2. Asigna una clave de proyecto y un nombre para mostrar a tu proyecto, luego haz clic en el botón "Configurar".
3. En la sección "Proporcionar un token", elige la opción "Generar un token". Asigna un nombre a tu token, haz clic en "Generar" y luego en "Continuar".
4. Selecciona el idioma principal de tu proyecto en la sección "Ejecutar análisis en tu proyecto" y sigue las instrucciones para analizar el proyecto. En este punto, podrás descargar y ejecutar un escáner en tu código (si estás utilizando Maven o Gradle, el escáner se descargará automáticamente).

Siguiendo estos sencillos pasos, SonarQube analizará tu proyecto y proporcionará información detallada sobre la calidad del código, ayudándote a mejorar y mantener un código limpio y seguro.

Una vez realizado el escáner del código, podrás ver el primer análisis en SonarQube:



Métricas Clave de SonarQube

SonarQube organiza las métricas en diversas categorías esenciales:

1. Complejidad: Refleja la Complejidad Ciclomática, calculada según la cantidad de caminos a través del código observados normalmente a nivel de métodos o funciones individuales.

2. Duplicados: Indica el número de bloques de líneas duplicados, ayudando a evitar discrepancias en operaciones similares.

3. Evidencias: Son fragmentos nuevos de código en un proyecto que incumplen alguna de las reglas establecidas.

4. Mantenibilidad: Se relaciona con el recuento total de problemas de Code Smell.

5. Umbrales de Calidad: Define los requisitos del proyecto antes de su lanzamiento a producción, como la ausencia de evidencias bloqueantes o una cobertura de código superior al 80% en el código nuevo.

6. Tamaño: Proporciona una visión general del volumen del proyecto en términos generales.

7. Pruebas: Verifican el correcto funcionamiento de una unidad de código y su integración.

La calidad del software es cada vez más crucial en el proceso de desarrollo. SonarQube se presenta como una plataforma integral para el análisis de código estático, fundamental para mejorar la calidad del código, reducir la deuda técnica y abordar vulnerabilidades. La implementación de esta herramienta de

automatización, gracias a su sencillez, se convierte en un elemento clave para desarrollar software de alta calidad.