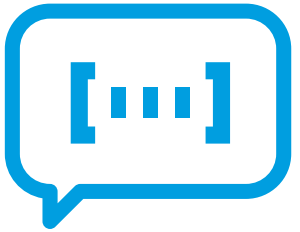
netmind
a bts company

Spring Boot

Testing Spring Boot

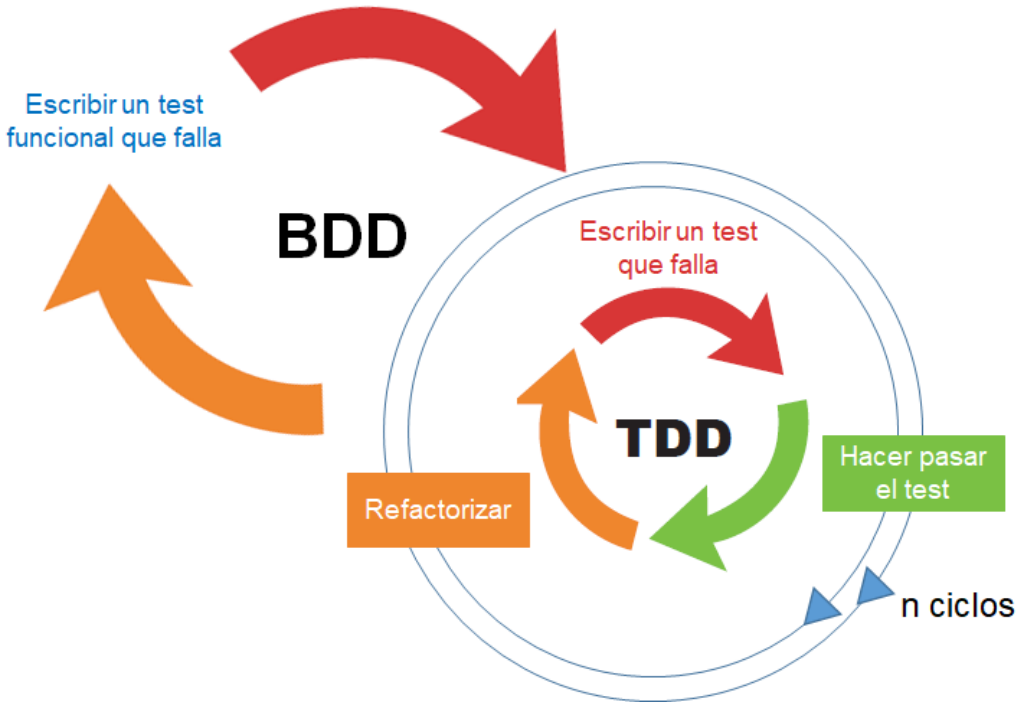
netmind
a bts company



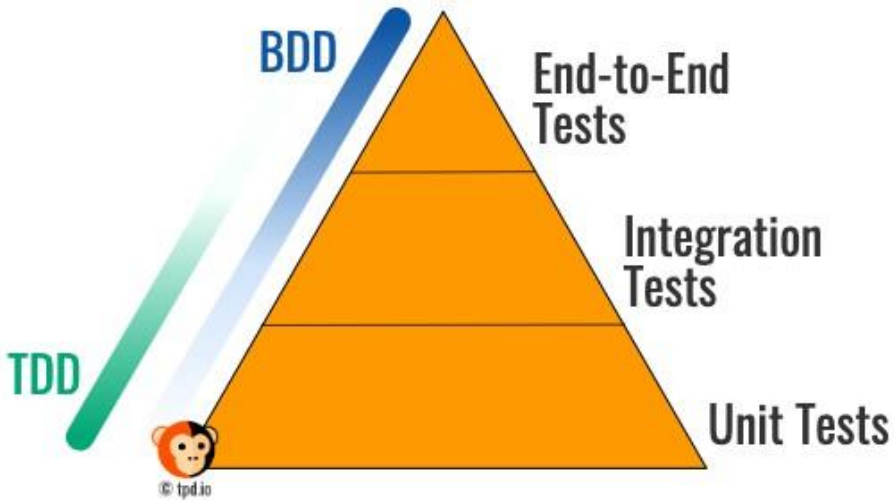
Testing en Spring Boot

01

BDD y TDD



- El Desarrollo guiado por pruebas (**TDD**: Test Driven Development) y el Desarrollo guiado por comportamiento (**BDD**: Behavior Driven Development) son metodologías complementarias que tienen como objetivo asegurar la calidad del software desde la fuente.
- Hemos visto que BDD puede servir para definir los servicios, basándonos en su comportamiento.



- Spring Boot provee herramientas para usar estos métodos incluyendo test automáticos en el delivery de nuestras aplicaciones.

Dependencias Maven

- Spring-boot-starter-test es la dependencia principal que contiene la mayoría de los elementos necesarios para nuestras pruebas.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
    <version>2.5.0</version>
</dependency>
```

- **Junit 4:** A partir de Spring Boot 2.4, el antiguo motor de JUnit se eliminó de spring-boot-starter-test. Si aún queremos usar JUnit 4, debemos agregar la siguiente dependencia:

```
<dependency>
    <groupId>org.junit.vintage</groupId>
    <artifactId>junit-vintage-engine</artifactId>
    <scope>test</scope>
    <exclusions>
        <exclusion>
            <groupId>org.hamcrest</groupId>
            <artifactId>hamcrest-core</artifactId>
        </exclusion>
    </exclusions>
</dependency>
```

Pruebas de integración con @SpringBootTest

- Las pruebas de integración se enfocan en integrar diferentes capas de la aplicación.
- Idealmente, deberíamos mantener las pruebas de integración separadas de las pruebas unitarias y no deberíamos ejecutarlas juntas.
- Podemos hacer esto usando un perfil diferente para ejecutar solo las pruebas de integración.
- Las razones para hacer esto es que:
  - las pruebas de integración consumen mucho tiempo y
  - pueden necesitar una base de datos real para ejecutarse.

Pruebas de integración con @SpringBootTest

- La anotación @SpringBootTest es útil cuando necesitamos arrancar todo el contenedor. La anotación funciona creando el **ApplicationContext** que se utilizará en nuestras pruebas.
  - Lo que significa que podemos @Autowire cualquier bean que haya sido recogido por el escaneo de componentes en nuestra prueba:
- Podemos usar el atributo **webEnvironment** de @SpringBootTest para configurar nuestro entorno de tiempo de ejecución; se usa WebEnvironment.MOCK para que el contenedor funcione en un entorno de servlet simulado.
- La anotación **@TestPropertySource** ayuda a configurar las ubicaciones de los archivos de propiedades específicos de nuestras pruebas.
  - Tener en cuenta que el archivo de propiedades cargado con @TestPropertySource anulará el archivo application.properties existente.

```
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.MOCK,
        classes = ProductsServiceApplication.class)
@AutoConfigureMockMvc
@TestPropertySource(locations = "classpath:application-integrationtest.properties")
public class ProductServiceControllerIntegrationTest {

    @Autowired
    private MockMvc mvc;

    @Autowired
    private ProductsRepository repository;

    // test cases here
}
```

Pruebas de integración con @SpringBootTest

- Los casos de prueba para las pruebas de integración pueden parecerse a las pruebas unitarias de la capa del controlador:

```
@Test
public void givenProducts_whenGetProducts_thenStatus200() throws Exception {
    Product nuevoProd = new Product();
    nuevoProd.setName("Nuevo Prod");
    repository.save(nuevoProd);

    mvc.perform(get("/products").contentType(MediaType.APPLICATION_JSON))
            .andExpect(status().isOk())
            .andExpect(content().contentTypeCompatibleWith(MediaType.APPLICATION_JSON))
            .andExpect(jsonPath("$[0].name", is("Nuevo Prod")));
}
```

- La diferencia con las pruebas unitarias de la capa del controlador es que aquí no se simula nada y se ejecutan escenarios de extremo a extremo.

Configuración de prueba con @TestConfiguration

- Es posible que deseemos evitar el arranque del contexto de la aplicación real y usar una configuración de prueba especial. Podemos lograr esto con la anotación @TestConfiguration .
- Hay dos formas de usar la anotación

(1). Una **clase interna estática** en la misma clase de prueba donde queremos @Autowire the bean:

```
@ExtendWith(SpringExtension.class)
public class ProductServiceImplIntegrationTest {

    @TestConfiguration
    static class ProductServiceImplTestContextConfiguration {
        @Bean
        public ProductsService productsService() {
           return new EmployeeService() {
                // implement methods
            };

        }
    }
}
```

Configuración de prueba con @TestConfiguration

(2). Crear una **clase de configuración de prueba** separada:

```
@TestConfiguration
public class ProductServiceImplTestContextConfiguration {
    @Bean
    public ProductsService productsService() {
       return new EmployeeService() {
            // implement methods
        };

    }
}
```

- Las clases de configuración anotadas con @TestConfiguration están excluidas del análisis de componentes, por lo tanto, debemos importarlas explícitamente en cada prueba en la que queramos hacer @Autowire .

```
@ExtendWith(SpringExtension.class)
@Import(ProductServiceImplTestContextConfiguration.class)
public class ProductServiceImplIntegrationTest {

    @Autowired
    private ProductsService productsService;

    // remaining class code
}
```

Mocking con con @MockBean

- Nuestro código de capa de Servicio usualmente depende de nuestro Repositorio.
- Sin embargo, para probar la capa de servicio , no necesitamos saber cómo se implementa la capa de persistencia ni preocuparnos por ella. Idealmente, deberíamos poder escribir y probar nuestro código de capa de servicio sin cableado en nuestra capa de persistencia completa.
- Para lograr esto, podemos usar el soporte de simulación proporcionado por Spring Boot Test.

```
@ExtendWith(SpringExtension.class)
public class ProductServiceImplIntegrationTest {

    @TestConfiguration
    static class ProductServiceImplTestContextConfiguration {
        @Bean
        public ProductsService productsService() {
            return new ProductsService();
        }
    }

    @Autowired
    private ProductsService productsService;

    @MockBean
    private ProductsRepository productsRepository;

    // write test cases here

}
```

Mocking con @MockBean

- @MockBean crea un simulacro para el Bean de repositorio , que se puede usar para omitir la llamada al repositorio real.

```
public void setUp() {
    List<Product> products = Arrays.asList(
        new Product(1L,"Fake product")
    );
    Mockito.when(productsRepository.findByNameContaining("Fake"))
            .thenReturn(products);
}
```

- El caso de prueba será más simple:

```
@Test
public void whenValidText_thenProductsShouldBeFound() {
    String text = "Fake";
    List<Product> found = productsService.
        getProductsByText(text);

    assertThat(found).isNotEmpty();
    assertThat(found.get(0).getName()).contains("Fake");
}
```

Pruebas de integración con @DataJpaTest

- **@ExtendWith(SpringExtension.class)** proporciona un puente entre las funciones de prueba de Spring Boot y JUnit.
  - Siempre que estemos usando cualquier característica de prueba de Spring Boot en nuestras pruebas JUnit, se requerirá esta anotación.
- **@DataJpaTest** proporciona una configuración estándar necesaria para probar la capa de persistencia:
  - configurar H2, una base de datos en memoria
  - configurar Hibernate, Spring Data y DataSource
  - realizando un @EntityScan
  - activar el registro de SQL
- Para llevar a cabo operaciones de base de datos, necesitamos algunos registros que ya estén en nuestra base de datos.
  - Para configurar estos datos, podemos usar **TestEntityManager**.
  - Spring Boot TestEntityManager es una alternativa al JPA EntityManager estándar que proporciona métodos comúnmente utilizados al escribir pruebas.

Pruebas de integración con @DataJpaTest

```
@ExtendWith(SpringExtension.class)
@DataJpaTest
public class ProductRepositoryIntegrationTest {
    @Autowired
    private TestEntityManager entityManager;

    @Autowired
    private ProductsRepository productsRepository;

    @Test
    public void whenFindByName_thenReturnProduct() {
        // given
        Product aProduct = new Product(null, "Fake Product");
        entityManager.persist(aProduct);
        entityManager.flush();

        // when
        Product found = productsRepository.findByName(aProduct.getName());

        // then
        assertThat(found.getName())
                .isEqualTo(aProduct.getName());
    }

}
```

Pruebas unitarias con @WebMvcTest

- Para probar los controladores , podemos usar **@WebMvcTest**.
  - Configurará automáticamente la infraestructura Spring MVC para nuestras pruebas unitarias.
- En la mayoría de los casos, @WebMvcTest se limitará a arrancar un solo controlador.
- También podemos usarlo junto con @MockBean para proporcionar implementaciones simuladas para cualquier dependencia requerida.
- @WebMvcTest también configura automáticamente MockMvc , que ofrece una forma poderosa de probar fácilmente los controladores MVC sin iniciar un servidor HTTP completo.

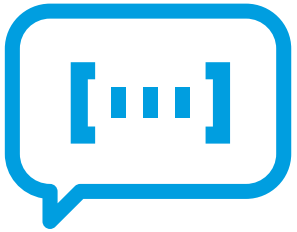
Pruebas unitarias con @WebMvcTest

```
@ExtendWith(SpringExtension.class)
@WebMvcTest(ProductServiceController.class)
public class ProductServiceControllerIntegrationTest {

    @Autowired
    private MockMvc mvc;

    @MockBean
    private ProductsRepository repository;

    @Test
    public void givenProducts_whenGetProducts_thenReturnJsonArray() throws Exception {

        Product aProduct = new Product(1L, "Fake product");

        List<Product> allProducts = Arrays.asList(aProduct);

        given(repository.findAll()).willReturn(allProducts);

        mvc.perform(get("/products")
                        .contentType(MediaType.APPLICATION_JSON))
                .andExpect(status().isOk())
                .andExpect(jsonPath("$", hasSize(1)))
                .andExpect(jsonPath("$[0].name", is(aProduct.getName())));
    }

}
```

Pruebas configuradas automáticamente

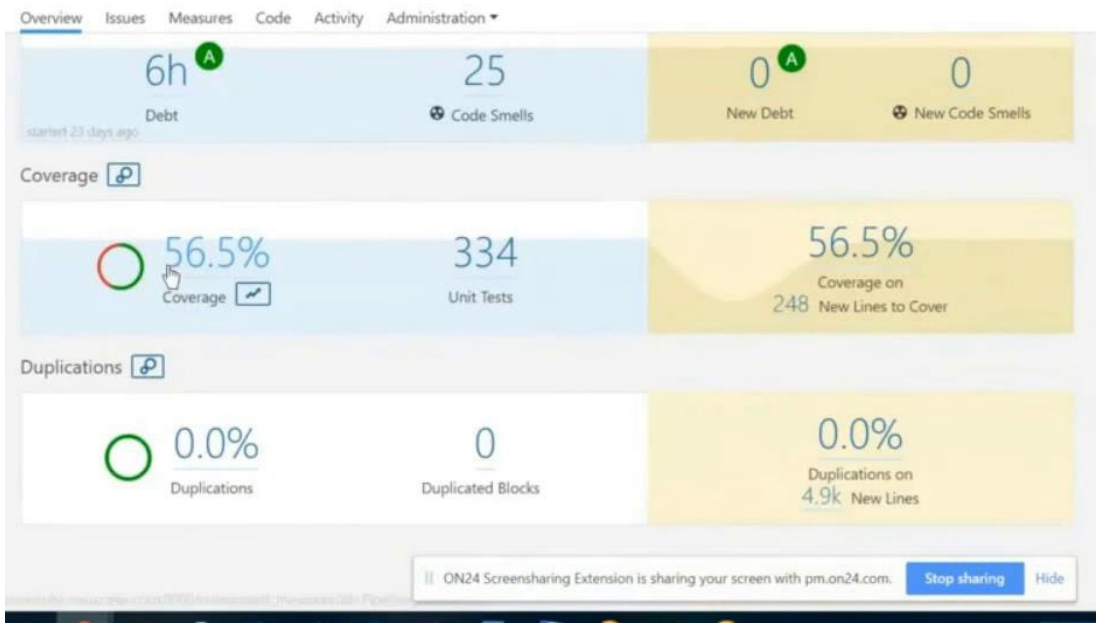
- Además de las anotaciones anteriores, algunas anotaciones ampliamente utilizadas:
  - **@WebFluxTest**: probar los controladores Spring WebFlux. A menudo se usa junto con @MockBean para proporcionar implementaciones simuladas para las dependencias requeridas.
  - **@JdbcTest**: probar aplicaciones JPA, pero es para pruebas que solo requieren un DataSource. La anotación configura una base de datos incrustada en memoria y un JdbcTemplate.
  - **@JooqTest**: pruebas relacionadas con jOOQ. Configura un DSLContext.
  - **@DataMongoTest**: para probar aplicaciones MongoDB. De manera predeterminada, configura un MongoDB incrustado en memoria si el controlador está disponible a través de dependencias, configura un MongoTemplate, busca clases de @Document y configura repositorios de Spring Data MongoDB.
  - **@DataRedisTest**: prueba de aplicaciones de Redis. Busca clases de @RedisHash y configura repositorios Spring Data Redis de forma predeterminada.
  - **@DataLdapTest**: configura un LDAP incrustado en la memoria (si está disponible), configura un LdapTemplate , busca clases de @Entry y configura repositorios Spring Data LDAP de manera predeterminada.
  - **@RestClientTest**: probar clientes REST. Configura automáticamente diferentes dependencias, como la compatibilidad con Jackson, GSON y Jsonb; configura un RestTemplateBuilder; y agrega soporte para MockRestServiceServer de forma predeterminada.
  - **@JsonTest**: Inicializa el contexto de la aplicación Spring solo con los beans necesarios para probar la serialización JSON.

02



SonarQube

Cobertura de código



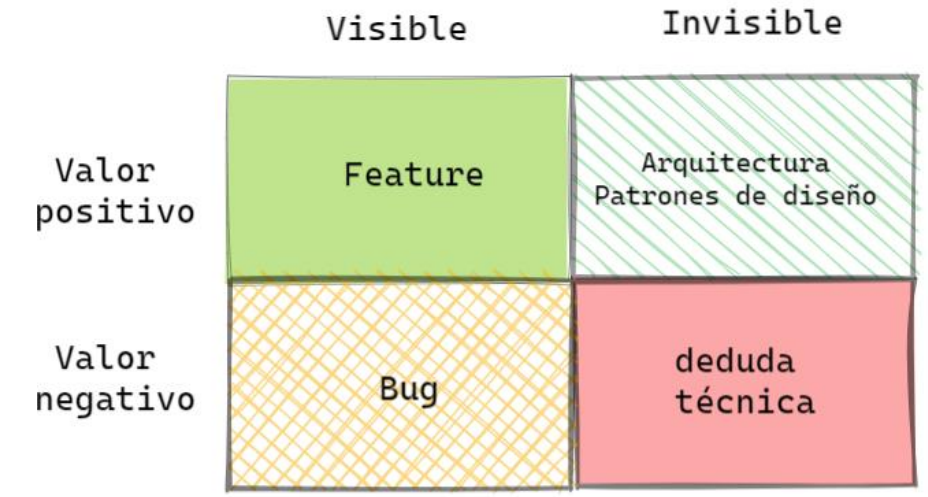
- **% de código** que está cubierto por pruebas automatizadas.
- Un sistema de cobertura de código recopila información sobre el programa en ejecución y luego lo combina con la información de origen para generar un informe sobre la cobertura de código del **conjunto de pruebas**.
- La cobertura del código es parte de un ciclo de feedback en el proceso de desarrollo.

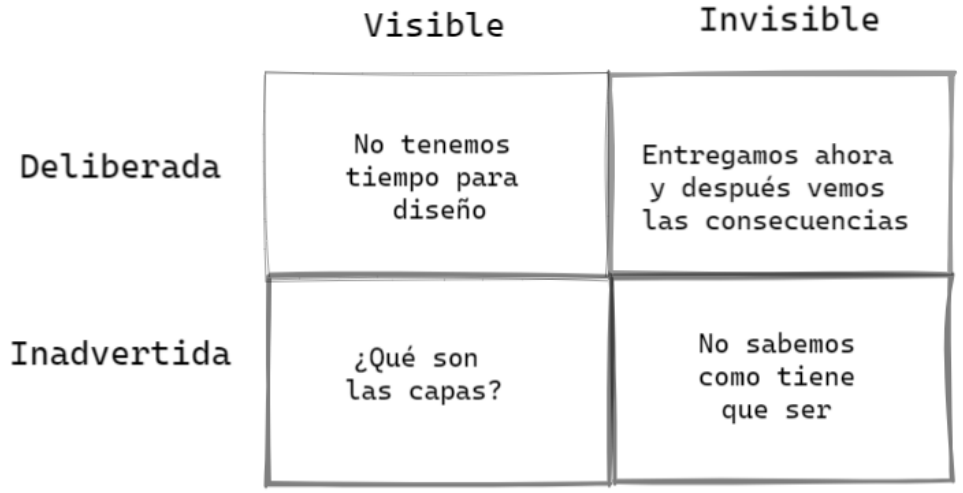
Aproximaciones:

- Instrumentación de código fuente
- Instrumentación de código intermedio
- Recolección de información de tiempo de ejecución

Bugs y deuda técnica

- Es una analogía para poder explicar algo tan complejo como la necesidad de hacer "refactoring" o de **por qué invertir en la "calidad de código"**.
  - "Hoy lo entrego sin haber hecho todas esas prácticas de calidad, pero me quedo con la deuda de hacerlas en algún momento (cuando tenga tiempo)"
- El cálculo se basa en la metodología SQALE (Software Quality Assessment based on Lifecycle Expectations).

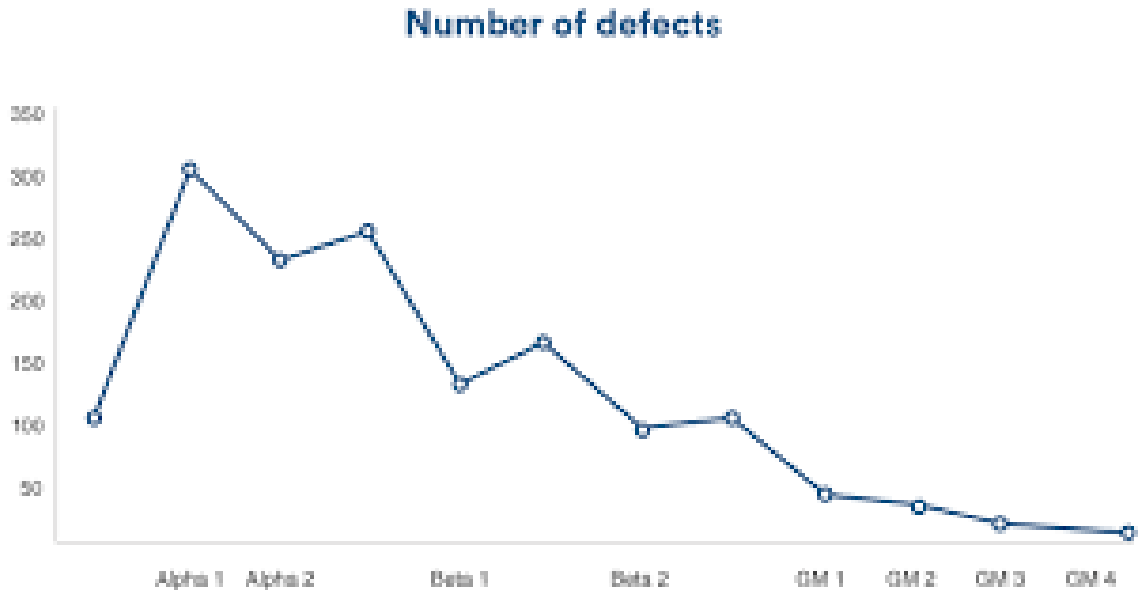




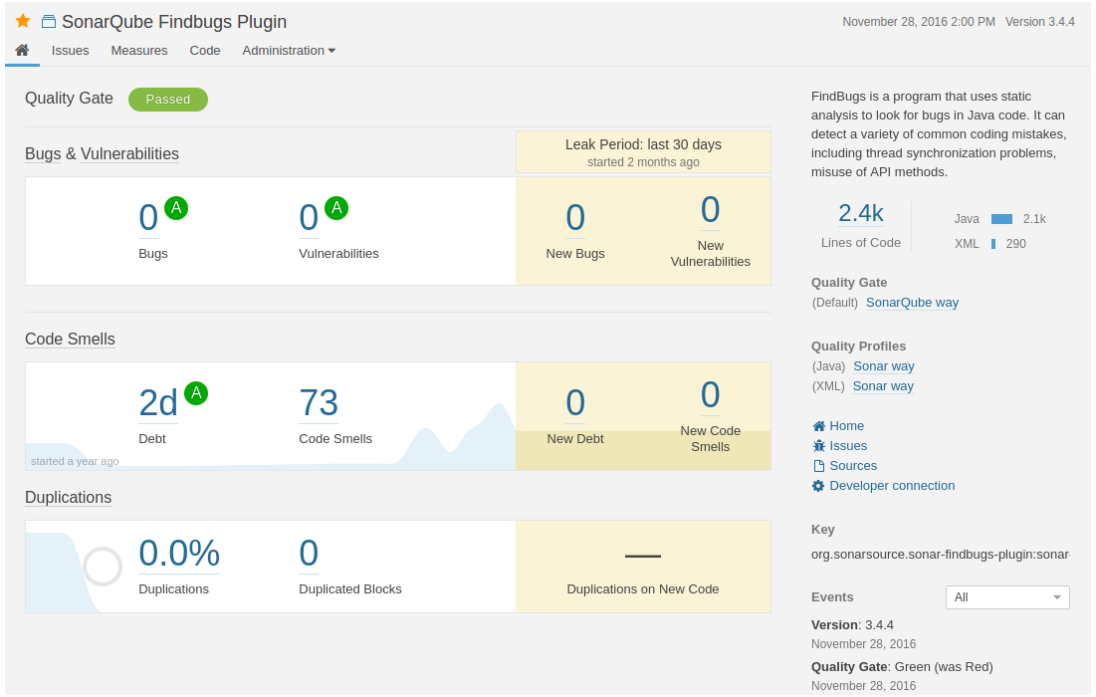
Deuda técnica

La métrica de deuda técnica se mide en horas:

- Estima la cantidad de días y horas que se deberían invertir para poder pagar la deuda.
- Toma en cuenta:
  - Los bloques duplicados
  - Las pruebas unitarias falladas
  - Ramas cubiertas por las pruebas unitarias insuficientes
  - Densidad de comentarios insuficientes
  - Cobertura de líneas cubierta por pruebas unitarias insuficientes
  - Pruebas unitarias omitidas



SonarQube



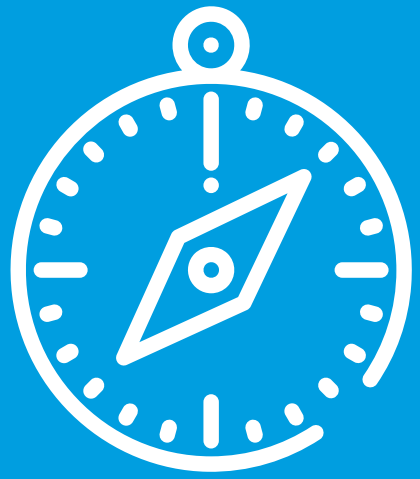
- Plataforma de código abierto para la **inspección continua de calidad de código, revisiones automáticas con análisis estático del código para detectar errores, olores de código y vulnerabilidades** de seguridad en más de 20 lenguajes de programación.
- Informes sobre código duplicado, estándares de codificación, pruebas unitarias, cobertura de código, complejidad de código, comentarios, errores y vulnerabilidades de seguridad.
- **Análisis e integración totalmente automatizados** con Maven, Ant, Gradle, MSBuild y herramientas de integración continua (Atlassian Bamboo, Jenkins, Hudson, etc.)

- https://www.sonarqube.org/

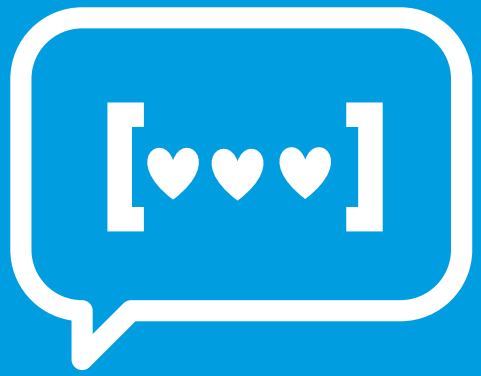
Instalación y análisis con SonarQube

- Instala SonarQube:
  - https://docs.sonarsource.com/sonarqube/latest/analyzing-source-code/scanners/sonarscanner-for-maven/
- Integra con un proyecto Spring Boot
  - http://blog.hadsonpar.com/2023/01/configuracion-y-ejecucion-de-sonarqube.html

30 min



Next steps



We would like to know your opinion!

Please, let us know what you think about the content.
From Netmind we want to say thank you, we appreciate time and effort you have taking in answering all of that is important in order to improve our training plans so that you will always be satisfied with having chosen us
quality@netmind.es

netmind
a bts company

Thanks!

Follow us: