USERDETAILMANAGEMENT

VERSION 0.0.1-SNAPSHOT

Code analysis

By: Administrator

2023-04-07

CONTENT

INTRODUCTION

This document contains results of the code analysis of UserDetailManagement.

Demo project for Spring Boot Mockito

CONFIGURATION

- Quality Profiles
  - Names: javaCustomProfile [Java]; Sonar way [XML];
  - Files: AYUVAI0bQUyjOCvxQ-HS.json; AYMbsodqSwijhRfWmTsQ.json;
- Quality Gate
  - Name: Sonar way
  - File: Sonar way.xml

SYNTHESIS

ANALYSIS STATUS

Reliability	Security	Security Review	Maintainability
A	D	A	A

QUALITY GATE STATUS

Quality Gate Status	Passed

METRICS

Coverage	Duplication	Comment density	Median number of lines of code per file	Adherence to coding standard
100.0 %	0.0 %	0.6 %	26.0	99.9 %

TESTS

Total	Success Rate	Skipped	Errors	Failures
7	100.0 %	0	0	0

DETAILED TECHNICAL DEBT

Reliability	Security	Maintainability	Total
-	0d 0h 10min	0d 0h 12min	0d 0h 22min

UserDetailManagement

UserDetailManagement

METRICS RANGE

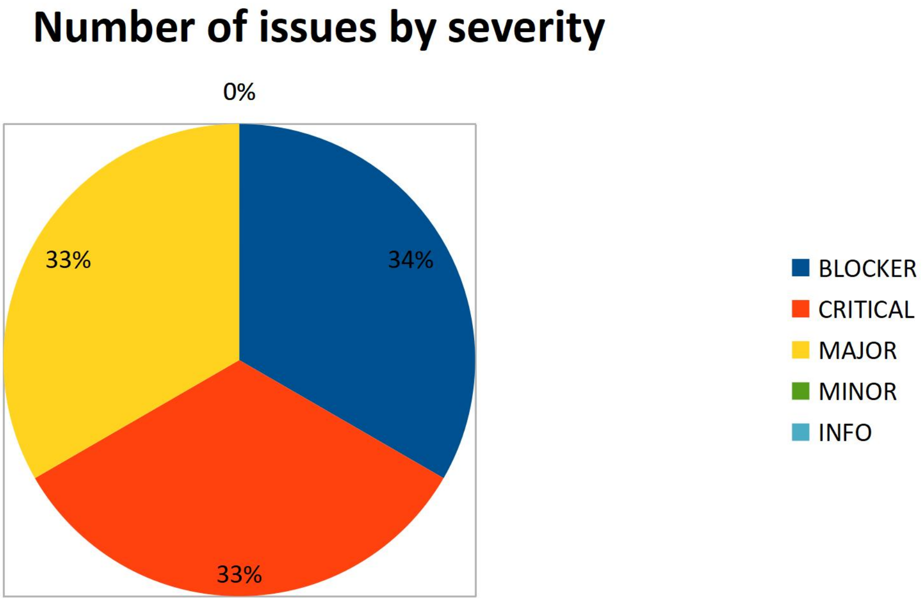
	Cyclomatic Complexity	Cognitive Complexity	Lines of code per file	Comment density (%)	Coverage	Duplication (%)
Min	0.0	0.0	8.0	0.0	100.0	0.0
Max	11.0	0.0	103.0	1.3	100.0	0.0

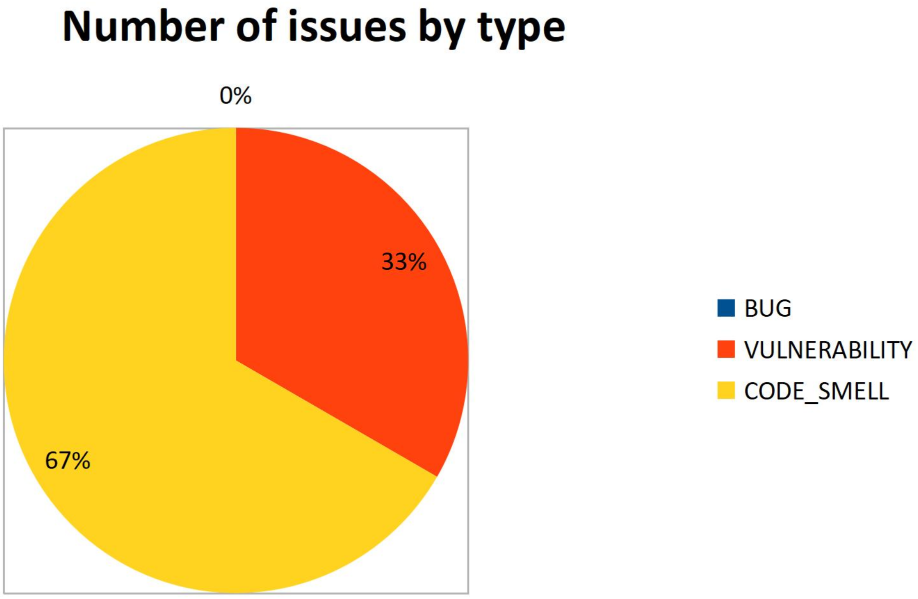
VOLUME

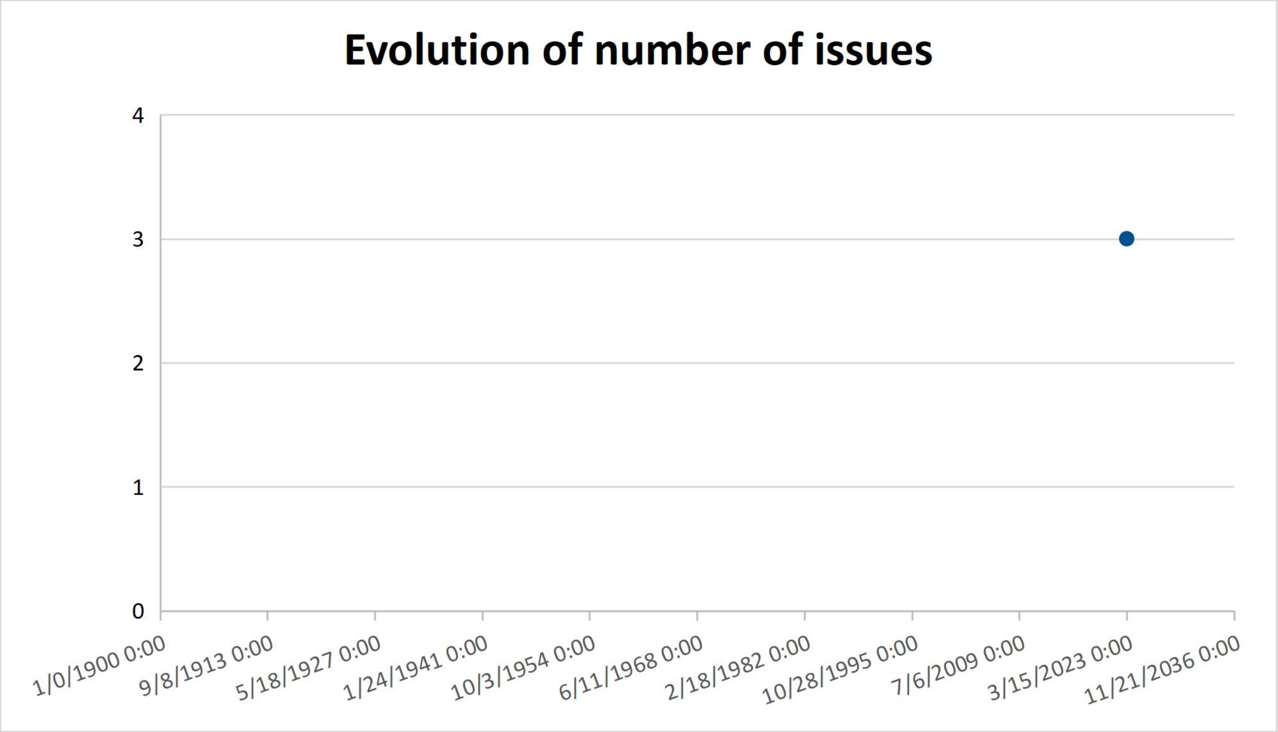
Language	Number
Java	103
XML	75
Total	178

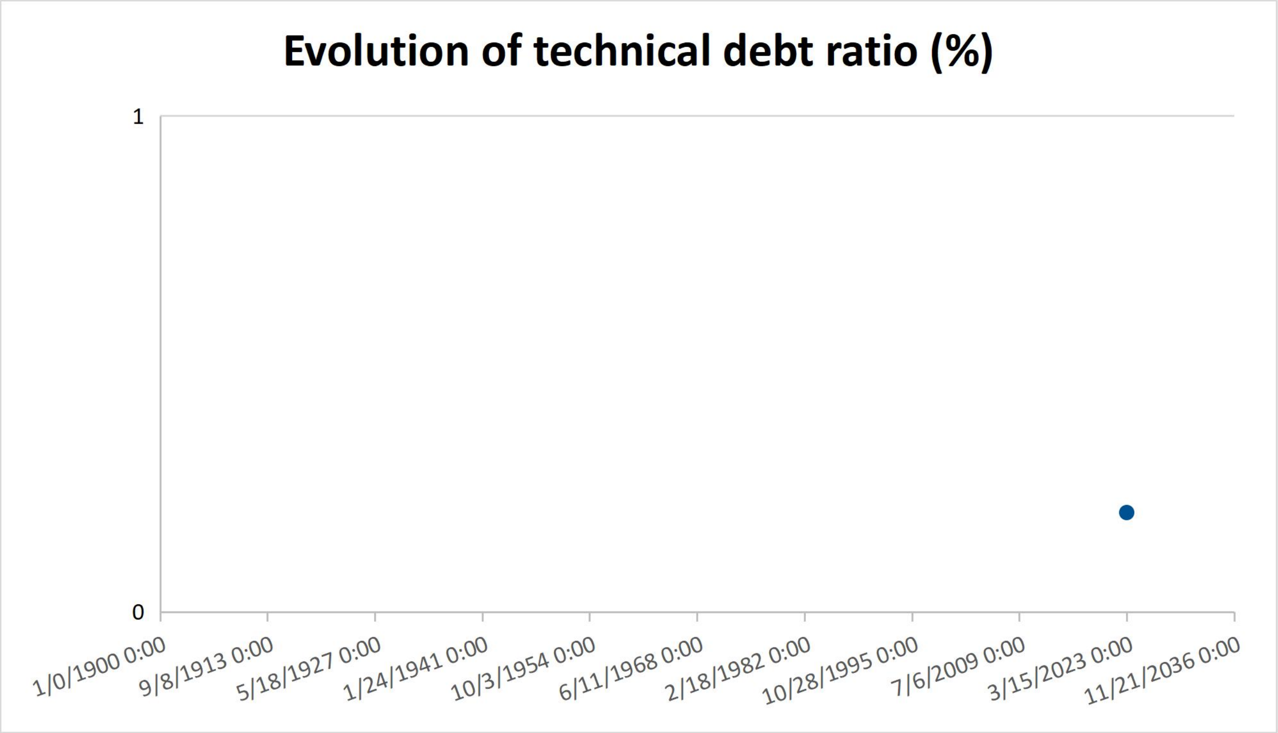
ISSUES

CHARTS






Evolution of number of issues
4
3
2
1
0
1/0/1900 0:00
9/8/1913 0:00
5/18/1927 0:00
1/24/1941 0:00
10/3/1954 0:00
6/11/1968 0:00
2/18/1982 0:00
10/28/1995 0:00
7/6/2009 0:00
3/15/2023 0:00
11/21/2036 0:00


Evolution of technical debt ratio (%)
1
0
1/0/1900 0:00
9/8/1913 0:00
5/18/1927 0:00
1/24/1941 0:00
10/3/1954 0:00
6/11/1968 0:00
2/18/1982 0:00
10/28/1995 0:00
7/6/2009 0:00
3/15/2023 0:00
11/21/2036 0:00

ISSUES COUNT BY SEVERITY AND TYPE

Type / Severity	INFO	MINOR	MAJOR	CRITICAL	BLOCKER
BUG	0	0	0	0	0
VULNERABILITY	0	0	0	1	0
CODE_SMELL	0	0	1	0	1

ISSUES LIST

Name	Description	Type	Severity	Number
Tests should include assertions	A test case without assertions ensures only that no exceptions are thrown. Beyond basic runnability, it ensures nothing about the behavior of the code under test. This rule raises an exception when no assertions from any of the following known frameworks are found in a test: AssertJ Awaitility EasyMock Eclipse Vert.x Fest 1.x and 2.x Hamcrest JMock JMockit JUnit Mockito Rest-assured 2.x, 3.x and 4.x RxJava 1.x and 2.x Selenide Spring's org.springframework.test.web.servlet.ResultActions.andExpect() and org.springframework.test.web.servlet.ResultActions.andExpectAll() Truth Framework WireMock Furthermore, as new or custom assertion frameworks may be used, the rule can be parametrized to define specific methods that will also be considered as assertions. No issue will be raised when such methods are found in test cases. The parameter value should have the following format &lt;FullyQualifiedClassName&gt;#&lt;MethodName&gt;, where MethodName can end with the wildcard character. For constructors, the pattern should be &lt;FullyQualifiedClassName&gt;#&lt;init&gt;. Example:&nbsp;com.company.CompareToTester#compare*,com.company.CustomAssert#customAssertMethod,com.company.CheckVerifier#&lt;init&gt;. Noncompliant Code Example @Test public void testDoSomething() { // Noncompliant MyClass myClass = new MyClass(); myClass.doSomething(); } Compliant Solution Example when com.company.CompareToTester#compare* is used as parameter to the rule. import com.company.CompareToTester; @Test public void testDoSomething() { MyClass myClass = new MyClass(); assertNull(myClass.doSomething()); // JUnit assertion assertThat(myClass.doSomething()).isNull(); // Fest assertion } @Test public void testDoSomethingElse() { MyClass myClass = new MyClass(); new CompareToTester().compareWith(myClass); // Compliant - custom assertion method defined as rule parameter CompareToTester.compareStatic(myClass);	CODE_SMELL	BLOCKER	1

	// Compliant }			
Assertion arguments should be passed in the correct order	The standard assertions library methods such as org.junit.Assert.assertEquals, and org.junit.Assert.assertSame expect the first argument to be the expected value and the second argument to be the actual value. For AssertJ, it’s the other way around, the argument of org.assertj.core.api.Assertions.assertThat is the actual value, and the subsequent calls contain the expected values. Swap them, and your test will still have the same outcome (succeed/fail when it should) but the error messages will be confusing. This rule raises an issue when the actual argument to an assertions library method is a hard-coded value and the expected argument is not. Supported frameworks: JUnit4 JUnit5 AssertJ Noncompliant Code Example org.junit.Assert.assertEquals(runner.exitCode(), 0, "Unexpected exit code"); // Noncompliant; Yields error message like: Expected:&lt;-1&gt;. Actual:&lt;0&gt;. org.assertj.core.api.Assertions.assertThat(0).isEqualTo(runner.exitCode()); // Noncompliant Compliant Solution org.junit.Assert.assertEquals(0, runner.exitCode(), "Unexpected exit code"); org.assertj.core.api.Assertions.assertThat(runner.exitCode()).isEqualTo(0);	CODE_SMELL	MAJOR	1
Persistent entities should not be used as arguments of "@RequestMapping" methods	On one side, Spring MVC automatically bind request parameters to beans declared as arguments of methods annotated with @RequestMapping. Because of this automatic binding feature, it’s possible to feed some unexpected fields on the arguments of the @RequestMapping annotated methods. On the other end, persistent objects (@Entity or @Document) are linked to the underlying database and updated automatically by a persistence framework, such as Hibernate, JPA or Spring Data MongoDB. These two facts combined together can lead to malicious attack: if a persistent object is used as an argument of a method annotated with @RequestMapping, it’s possible from a specially crafted user input, to change the content of unexpected fields into the database. For this reason, using @Entity or @Document objects as arguments of methods annotated with @RequestMapping should be avoided. In addition to @RequestMapping, this rule also considers the annotations introduced in Spring Framework 4.3: @GetMapping, @PostMapping, @PutMapping, @DeleteMapping, @PatchMapping. Noncompliant Code Example import javax.persistence.Entity; @Entity public class Wish { Long productId; Long quantity; Client client; } @Entity public class Client { String clientId; String name; String password; } import org.springframework.stereotype.Controller; import org.springframework.web.bind.annotation.RequestMapping; @Controller public class WishListController { @PostMapping(path = "/saveForLater") public String saveForLater(Wish wish) { session.save(wish); } @RequestMapping(path = "/saveForLater", method = RequestMethod.POST) public String saveForLater(Wish wish) { session.save(wish); } } Compliant Solution public class WishDTO { Long productId; Long quantity; Long clientId; } import org.springframework.stereotype.Controller; import org.springframework.web.bind.annotation.RequestMapping; @Controller public class PurchaseOrderController { @PostMapping(path = "/saveForLater") public String saveForLater(WishDTO wish) { Wish persistentWish = new Wish(); // do the mapping between "wish" and "persistentWish" [...] session.save(persistentWish); }	VULNERABILITY	CRITICAL	1

@RequestMapping(path = "/saveForLater", method = RequestMethod.POST) public String saveForLater(WishDTO wish) { Wish persistentWish = new Wish(); // do the mapping between "wish" and "persistentWish" [...] session.save(persistentWish); } } Exceptions No issue is reported when the parameter is annotated with @PathVariable from Spring Framework, since the lookup will be done via id, the object cannot be forged on client side. See OWASP Top 10 2021 Category A8 - Software and Data Integrity Failures OWASP Top 10 2017 Category A5 - Broken Access Control MITRE, CWE-915 - Improperly Controlled Modification of Dynamically-Determined Object Attributes Two Security Vulnerabilities in the Spring Framework's MVC by Ryan Berg and Dinis Cruz

SECURITY HOTSPOTS

SECURITY HOTSPOTS COUNT BY CATEGORY AND PRIORITY

Category / Priority	LOW	MEDIUM	HIGH
LDAP Injection	0	0	0
Object Injection	0	0	0
Server-Side Request Forgery (SSRF)	0	0	0
XML External Entity (XXE)	0	0	0
Insecure Configuration	0	0	0
XPath Injection	0	0	0
Authentication	0	0	0
Weak Cryptography	0	0	0
Denial of Service (DoS)	0	0	0
Log Injection	0	0	0
Cross-Site Request Forgery (CSRF)	0	0	0
Open Redirect	0	0	0
SQL Injection	0	0	0
Buffer Overflow	0	0	0
File Manipulation	0	0	0
Code Injection (RCE)	0	0	0
Cross-Site Scripting (XSS)	0	0	0
Command Injection	0	0	0

UserDetailManagement

Path Traversal Injection	0	0	0
HTTP Response Splitting	0	0	0
Others	0	0	0

SECURITY HOTSPOTS LIST