FRAG.JETZT BACKEND

VERSION 1.0

CODE ANALYSIS

By: Administrator

2023-01-25

CONTENT

INTRODUCTION

This document contains results of the code analysis of frag.jetzt Backend.

CONFIGURATION

- Quality Profiles
  - Names: Sonar way [Java]; Sonar way [XML];
  - Files: AYW221OPfGQw7yIQA4ko.json; AYW221R2fGQw7yIQA4v0.json;
- Quality Gate
  - Name: frag.jetzt
  - File: frag.jetzt.xml

SYNTHESIS

ANALYSIS STATUS

Reliability	Security	Security Review	Maintainability
B	A	E	A

QUALITY GATE STATUS

Quality Gate Status	Failed

Metric	Value
Reliability Rating	ERROR (B is worse than A)
Security Rating	OK
Maintainability Rating	OK
Coverage	ERROR (1.7% is less than 80%)
Duplicated Lines (%)	ERROR (8.1% is greater than 3%)

METRICS

Coverage	Duplication	Comment density	Median number of lines of code per file	Adherence to coding standard
1.7 %	8.1 %	0.4 %	36.5	97.5 %

TESTS

Total	Success Rate	Skipped	Errors	Failures

3	**100.0 %**	**0**	**0**	**0**

DETAILED TECHNICAL DEBT

Reliability	Security	Maintainability	Total
1d 2h 5min	-	3d 3h 57min	**4d 6h 2min**

METRICS RANGE

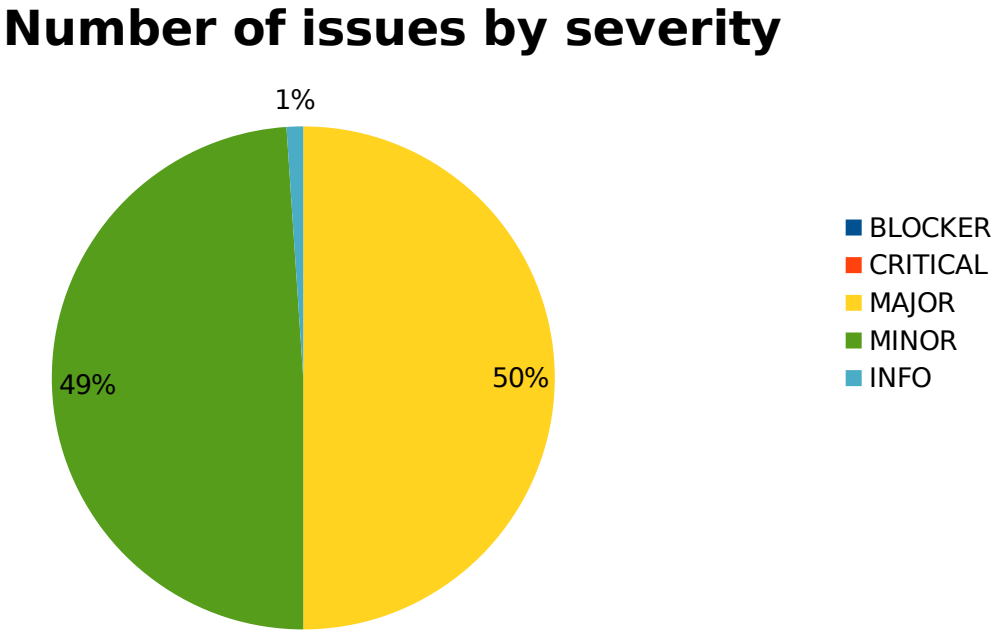
	Cyclomatic Complexity	Cognitive Complexity	Lines of code per file	Comment density (%)	Coverage	Duplication (%)
Min	0.0	0.0	3.0	0.0	0.0	0.0
Max	2701.0	689.0	11270.0	9.4	100.0	56.6

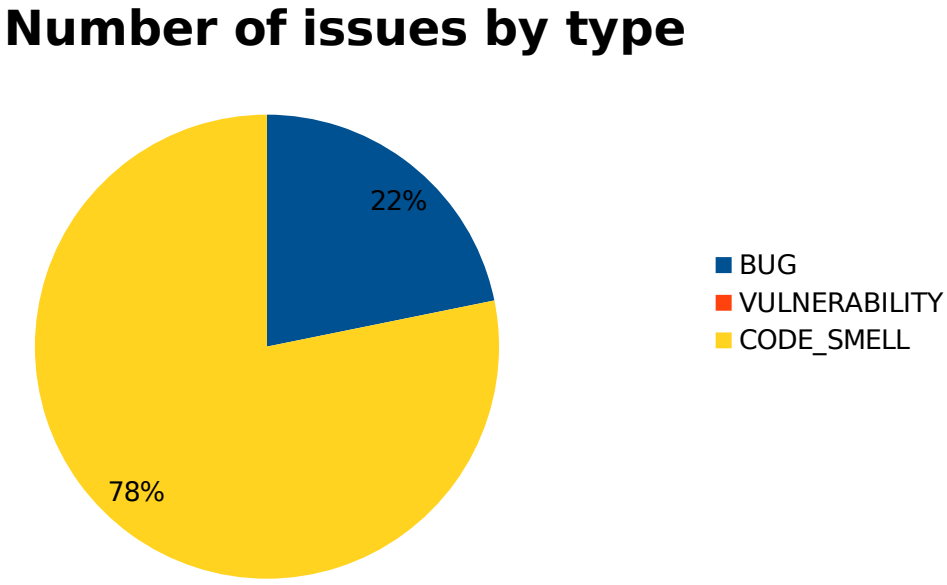
VOLUME

Language	Number
Java	11270
XML	169
Total	11439

ISSUES

CHARTS





ISSUES COUNT BY SEVERITY AND TYPE

Type / Severity	INFO	MINOR	MAJOR	CRITICAL	BLOCKER
BUG	0	41	0	0	0
VULNERABILITY	0	0	0	0	0
CODE_SMELL	2	51	94	0	0

ISSUES LIST

Name	Description	Type	Severity	Number
"equals(Object obj)" and "hashCode()" should be overridden in pairs	According to the Java Language Specification, there is a contract between equals(Object) and hashCode(): If two objects are equal according to the equals(Object) method, then calling the hashCode method on each of the two objects must produce the same integer result. It is not required that if two objects are unequal	BUG	MINOR	40

according to the equals(java.lang .Object) method, then calling the hashCode method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hashtables. In order to comply with this contract, those methods should be either both inherited, or both overridden. Noncompliant Code Example class MyClass { // Noncompliant - should also override "hashCode()" @Override public boolean equals(Object obj) { /* ... */ } } Compliant Solution class MyClass { // Compliant @Override public boolean equals(Object obj) { /* ... */ } @Override public int hashCode() { /* ... */ } } See MITRE, CWE-581 - Object Model

	Violation: Just One of Equals and Hashcode Defined CERT, MET09-J. - Classes that define an equals() method must also define a hashCode() method			
Math operands should be cast before assignment	When arithmetic is performed on integers, the result will always be an integer. You can assign that result to a long, double, or float with automatic type conversion, but having started as an int or long, the result will likely not be what you expect. For instance, if the result of int division is assigned to a floating-point variable, precision will have been lost before the assignment. Likewise, if the result of multiplication is assigned to a long, it may have already overflowed before the assignment. In either case, the result will not be what was expected. Instead, at least	BUG	MINOR	1

one operand should be cast or promoted to the final type before the operation takes place. Noncompliant Code Example float twoThirds = 2/3; // Noncompliant; int division. Yields 0.0 long millisInYear = 1_000*3_600*24*365; // Noncompliant; int multiplication. Yields 1471228928 long bigNum = Integer.MAX_VALUE + 2; // Noncompliant. Yields -2147483647 long bigNegNum = Integer.MIN_VALUE-1; //Noncompliant, gives a positive result instead of a negative one. Date myDate = new Date(seconds * 1_000); //Noncompliant, won't produce the expected result if seconds > 2_147_483 ... public long compute(int factor){ return factor * 10_000; //Noncompliant, won't produce the expected

result if factor > 214_748 } public float compute2(long factor){ return factor / 123; //Noncompliant, will be rounded to closest long integer } Compliant Solution float twoThirds = 2f/3; // 2 promoted to float. Yields 0.6666667 long millisInYear = 1_000L*3_600*24*365; // 1000 promoted to long. Yields 31_536_000_000 long bigNum = Integer.MAX_VALUE + 2L; // 2 promoted to long. Yields 2_147_483_649 long bigNegNum = Integer.MIN_VALUE-1L; // Yields -2_147_483_649 Date myDate = new Date(seconds * 1_000L); ... public long compute(int factor){ return factor * 10_000L; } public float compute2(long factor){ return factor / 123f; } or float twoThirds = (float)2/3; // 2 cast to float long millisInYear = (long)1_000*3_6

	00*24*365; // 1_000 cast to long long bigNum = (long)Integer.MAX_VALUE + 2; long bigNegNum = (long)Integer.MIN_VALUE-1; Date myDate = new Date((long)seconds * 1_000); ... public long compute(long factor){ return factor * 10_000; } public float compute2(float factor){ return factor / 123; } See MITRE, CWE-190 - Integer Overflow or Wraparound CERT, NUM50-J. - Convert integers to floating point for floating-point operations CERT, INT18-C. - Evaluate integer expressions in a larger size before comparing or assigning to that size SANS Top 25 - Risky Resource Management			
Deprecated code should be removed	This rule is meant to be used as a way to track code which is marked as being deprecated. Deprecated code should	CODE_SMELL	INFO	2

	eventually be removed. Noncompliant Code Example class Foo { /** * @deprecated */ public void foo() { // Noncompliant } @Deprecated // Noncompliant public void bar() { } public void baz() { // Compliant } }			
Source files should not have any duplicated blocks	An issue is created on a file as soon as there is at least one block of duplicated code on this file	CODE_SMELL	MAJOR	37
Unused "private" fields should be removed	If a private field is declared but not used in the program, it can be considered dead code and should therefore be removed. This will improve maintainability because developers will not wonder what the variable is used for. Note that this rule does not take reflection into account, which means that issues will be raised on private fields that are only accessed using the reflection API. Noncompliant Code Example	CODE_SMELL	MAJOR	17

public class MyClass { private int foo = 42; public int compute(int a) { return a * 42; } } Compliant Solution public class MyClass { public int compute(int a) { return a * 42; } } Exceptions The Java serialization runtime associates with each serializable class a version number, called serialVersionUID, which is used during deserialization to verify that the sender and receiver of a serialized object have loaded classes for that object that are compatible with respect to serialization. A serializable class can declare its own serialVersionUID explicitly by declaring a field named serialVersionUID that must be static, final, and of type long. By definition those serialVersionUID fields should not be reported by this rule: public class MyClass implements

	java.io.Serializable { private static final long serialVersionUID = 42L; } Moreover, this rule doesn't raise any issue on annotated fields.			
Methods should not have too many parameters	A long parameter list can indicate that a new structure should be created to wrap the numerous parameters or that the function is doing too many things. Noncompliant Code Example With a maximum number of 4 parameters: public void doSomething(int param1, int param2, int param3, String param4, long param5) { ... } Compliant Solution public void doSomething(int param1, int param2, int param3, String param4) { ... } Exceptions Methods annotated with : Spring's @RequestMapping (and related shortcut annotations, like @GetRequest) JAX-RS API	CODE_SMELL	MAJOR	1

	annotations (like @javax.ws.rs.GET) Bean constructor injection with @org.springframework.beans.factory.annotation.Autowired CDI constructor injection with @javax.inject.Inject @com.fasterxml.jackson.annotation.JsonCreator may have a lot of parameters, encapsulation being possible. Such methods are therefore ignored.			
Local variables should not shadow class fields	Overriding or shadowing a variable declared in an outer scope can strongly impact the readability, and therefore the maintainability, of a piece of code. Further, it could lead maintainers to introduce bugs because they think they're using one variable but are really using another. Noncompliant Code Example class Foo { public int myField; public void doSomething() { int myField = 0;	CODE_SMELL	MAJOR	1

... } } See CERT, DCL01-C. - Do not reuse variable names in subscopes CERT, DCL51-J. - Do not shadow or obscure identifiers in subscopes

Utility classes should not have public constructors