|  |
| --- |
| pokeapi-soap-service  Branch : main  Version 0.0.1-SNAPSHOT  Code analysis |

|  |
| --- |
| **By: Administrator**  **2024-07-23** |

# Content

[Content 1](#_Toc97156258)

[Introduction 2](#_Toc97156259)

[Configuration 2](#_Toc97156260)

[Synthesis 3](#_Toc97156261)

[Analysis Status 3](#_Toc97156262)

[Quality gate status 3](#_Toc97156263)

[Metrics 3](#_Toc97156264)

[Tests 3](#_Toc97156265)

[Detailed technical debt 3](#_Toc97156266)

[Metrics Range 5](#_Toc97156267)

[Volume 5](#_Toc97156268)

[Issues 6](#_Toc97156269)

[Charts 6](#_Toc97156270)

[Issues count by severity and type 8](#_Toc97156271)

[Issues List 8](#_Toc97156272)

[Security Hotspots 9](#_Toc97156273)

[Security hotspots count by category and priority 9](#_Toc97156274)

[Security hotspots List 9](#_Toc97156275)

# Introduction

This document contains results of the code analysis of pokeapi-soap-service.

A Java Spring Boot application that consumes the PokeAPI REST service and exposes a SOAP web service with methods to retrieve various Pokémon details.

# Configuration

* Quality Profiles
  + Names: Sonar way [Java]; Sonar way [XML];
  + Files: 4d325172-74ca-4ac5-b1ec-2268059b6424.json; b3460f59-c417-47b6-badf-79f3251d11ff.json;
* Quality Gate
  + Name: Sonar way
  + File: Sonar way.xml

# Synthesis

## Analysis Status

|  |  |  |  |
| --- | --- | --- | --- |
| Reliability | Security | Security Review | Maintainability |
| A.png | **A.png** | **A.png** | **A.png** |

## Quality gate status

|  |  |
| --- | --- |
| Quality Gate Status | **OK.png** |



## Metrics

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Coverage | Duplication | Comment  density | Median number of lines of code per file | Adherence to coding standard |
| 0.0 % | **0.0 %** | **1.9 %** | **22.0** | **99.7 %** |

## Tests

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Total | Success Rate | Skipped | Errors | Failures |
| 0 | **0 %** | **0** | **0** | **0** |

## Detailed technical debt

|  |  |  |  |
| --- | --- | --- | --- |
| Reliability | Security | Maintainability | Total |
| - | - | 0d 1h 20min | 0d 1h 20min |

## Metrics Range

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Cyclomatic  Complexity | Cognitive  Complexity | Lines of code per file | Comment  density (%) | Coverage | Duplication (%) |
| Min | 0.0 | 0.0 | 7.0 | 0.0 | 0.0 | 0.0 |
| Max | 44.0 | 3.0 | 909.0 | 7.5 | 0.0 | 0.0 |

## Volume

|  |  |
| --- | --- |
| Language | Number |
| Java | 909 |
| XML | 196 |
| Total | 1105 |

# Issues

## Charts

## Issues count by severity and type

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Type / Severity | INFO | MINOR | MAJOR | CRITICAL | BLOCKER |
| BUG | 0 | 0 | 0 | 0 | 0 |
| VULNERABILITY | 0 | 0 | 0 | 0 | 0 |
| CODE\_SMELL | 7 | 9 | 5 | 0 | 0 |

## Issues List

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Name | Description | Type | Severity | Number |
| JUnit5 test classes and methods should have default package visibility | JUnit5 test classes and methods should generally have package visibility. To fix this issue, change their visibility to the default package visibility. Why is this an issue? JUnit5 is more tolerant regarding the visibility of test classes and methods than JUnit4, which required everything to be public. Test classes and methods can have any visibility except private. It is however recommended to use the default package visibility to improve readability. Test classes, test methods, and lifecycle methods are not required to be public, but they must not be private. It is generally recommended to omit the public modifier for test classes, test methods, and lifecycle methods unless there is a technical reason for doing so – for example, when a test class is extended by a test class in another package. Another technical reason for making classes and methods public is to simplify testing on the module path when using the Java Module System. — JUnit5 User Guide What is the potential impact? The code will be non-conventional and readability can be slightly affected. Exceptions This rule does not raise an issue when the visibility is set to private, because private test methods and classes are systematically ignored by JUnit5, without a proper warning. In this case, there is also an impact on reliability and so it is handled by the rule S5810. How to fix it You can simply change the visibility by removing the public or protected keywords. Code examples Noncompliant code example import org.junit.jupiter.api.Test; public class MyClassTest { // Noncompliant - modifier can be removed @Test protected void test() { // Noncompliant - modifier can be removed // ... } } Compliant solution import org.junit.jupiter.api.Test; class MyClassTest { @Test void test() { // ... } } Resources Documentation JUnit5 User Guide: Test Classes and Methods | CODE\_SMELL | INFO | 7 |
| Generic exceptions should never be thrown | This rule raises an issue when a generic exception (such as Error, RuntimeException, Throwable, or Exception) is thrown. Why is this an issue? Throwing generic exceptions such as Error, RuntimeException, Throwable, and Exception will have a negative impact on any code trying to catch these exceptions. From a consumer perspective, it is generally a best practice to only catch exceptions you intend to handle. Other exceptions should ideally be let to propagate up the stack trace so that they can be dealt with appropriately. When a generic exception is thrown, it forces consumers to catch exceptions they do not intend to handle, which they then have to re-throw. Besides, when working with a generic type of exception, the only way to distinguish between multiple exceptions is to check their message, which is error-prone and difficult to maintain. Legitimate exceptions may be unintentionally silenced and errors may be hidden. For instance, when a Throwable is caught and not re-thrown, it may mask errors such as OutOfMemoryError and prevent the program from terminating gracefully. When throwing an exception, it is therefore recommended to throw the most specific exception possible so that it can be handled intentionally by consumers. Exceptions Generic exceptions in the signatures of overriding methods are ignored, because an overriding method has to follow the signature of the throw declaration in the superclass. The issue will be raised on superclass declaration of the method (or won’t be raised at all if superclass is not part of the analysis). @Override public void myMethod() throws Exception {...} Generic exceptions are also ignored in the signatures of methods that make calls to methods that throw generic exceptions. public void myOtherMethod() throws Exception { doTheThing(); // this method throws Exception } How to fix it To fix this issue, make sure to throw specific exceptions that are relevant to the context in which they arise. It is recommended to either: Raise a specific exception from the Java standard library when one matches. For example an IllegalArgumentException should be thrown when a method receives an invalid argument. Create a custom exception class deriving from Exception or one of its subclasses. Code examples Noncompliant code example void checkValue(int value) throws Throwable { // Noncompliant: signature is too broad if (value == 42) { throw new RuntimeException("Value is 42"); // Noncompliant: This will be difficult for consumers to handle } } Compliant solution void checkValue(int value) { if (value == 42) { throw new IllegalArgumentException("Value is 42"); // Compliant } } Resources Standards CWE - CWE-397 Declaration of Throws for Generic Exception CERT - ERR07-J. Do not throw RuntimeException, Exception, or Throwable Related rules S1181 - Generic exceptions should not be caught | CODE\_SMELL | MAJOR | 1 |
| "Preconditions" and logging arguments should not require evaluation | Why is this an issue? Some method calls can effectively be "no-ops", meaning that the invoked method does nothing, based on the application’s configuration (eg: debug logs in production). However, even if the method effectively does nothing, its arguments may still need to evaluated before the method is called. Passing message arguments that require further evaluation into a Guava com.google.common.base.Preconditions check can result in a performance penalty. That is because whether or not they’re needed, each argument must be resolved before the method is actually called. Similarly, passing concatenated strings into a logging method can also incur a needless performance hit because the concatenation will be performed every time the method is called, whether or not the log level is low enough to show the message. Instead, you should structure your code to pass static or pre-computed values into Preconditions conditions check and logging calls. Specifically, the built-in string formatting should be used instead of string concatenation, and if the message is the result of a method call, then Preconditions should be skipped altogether, and the relevant exception should be conditionally thrown instead. Noncompliant code example logger.log(Level.DEBUG, "Something went wrong: " + message); // Noncompliant; string concatenation performed even when log level too high to show DEBUG messages logger.fine("An exception occurred with message: " + message); // Noncompliant LOG.error("Unable to open file " + csvPath, e); // Noncompliant Preconditions.checkState(a &gt; 0, "Arg must be positive, but got " + a); // Noncompliant. String concatenation performed even when a &gt; 0 Preconditions.checkState(condition, formatMessage()); // Noncompliant. formatMessage() invoked regardless of condition Preconditions.checkState(condition, "message: %s", formatMessage()); // Noncompliant Compliant solution logger.log(Level.DEBUG, "Something went wrong: {0} ", message); // String formatting only applied if needed logger.log(Level.SEVERE, () -&gt; "Something went wrong: " + message); // since Java 8, we can use Supplier , which will be evaluated lazily logger.fine("An exception occurred with message: {}", message); // SLF4J, Log4j LOG.error("Unable to open file {0}", csvPath, e); if (LOG.isDebugEnabled()) { LOG.debug("Unable to open file " + csvPath, e); // this is compliant, because it will not evaluate if log level is above debug. } Preconditions.checkState(arg &gt; 0, "Arg must be positive, but got %d", a); // String formatting only applied if needed if (!condition) { throw new IllegalStateException(formatMessage()); // formatMessage() only invoked conditionally } if (!condition) { throw new IllegalStateException("message: " + formatMessage()); } Exceptions catch blocks are ignored, because the performance penalty is unimportant on exceptional paths (catch block should not be a part of standard program flow). Getters are ignored as well as methods called on annotations which can be considered as getters. This rule accounts for explicit test-level testing with SLF4J methods isXXXEnabled and ignores the bodies of such if statements. | CODE\_SMELL | MAJOR | 1 |
| Format strings should be used correctly | Why is this an issue? A printf--style format string is a string that contains placeholders, usually represented by special characters such as "%s" or "{}", depending on the technology in use. These placeholders are replaced by values when the string is printed or logged. Because printf-style format strings are interpreted at runtime, rather than validated by the compiler, they can contain errors that result in the wrong strings being created. This rule checks whether every format string specifier can be correctly matched with one of the additional arguments when calling the following methods: java.lang.String#format java.util.Formatter#format java.io.PrintStream#format java.text.MessageFormat#format java.io.PrintWriter#format java.io.PrintStream#printf java.io.PrintWriter#printf java.lang.String#formatted (since Java 15) logging methods of org.slf4j.Logger, java.util.logging.Logger, org.apache.logging.log4j.Logger. How to fix it A printf--style format string is a string that contains placeholders, which are replaced by values when the string is printed or logged. Mismatch in the format specifiers and the arguments provided can lead to incorrect strings being created. To avoid issues, a developer should ensure that the provided arguments match format specifiers. Note that MessageFormat is used by most logging mechanisms, for example java.util.logging.Logger, thus the single quote must be escaped by a double single quote. Code examples Noncompliant code example void logging(org.slf4j.Logger slf4jLog, java.util.logging.Logger logger) { String.format("Too many arguments %d and %d", 1, 2, 3); // Noncompliant - the third argument '3' is unused String.format("First {0} and then {1}", "foo", "bar"); //Noncompliant - it appears there is confusion with the use of "java.text.MessageFormat" - parameters "foo" and "bar" will be ignored here slf4jLog.debug("The number: ", 1); // Noncompliant - String contains no format specifiers. logger.log(level, "Can't load library \"{0}\"!", "foo"); // Noncompliant - the single quote ' must be escaped } Compliant solution void logging(org.slf4j.Logger slf4jLog, java.util.logging.Logger logger) { String.format("Too many arguments %d and %d", 1, 2); String.format("First %s and then %s", "foo", "bar"); slf4jLog.debug("The number: {}", 1); logger.log(level, "Can''t load library \"{0}\"!", "foo"); } Resources CERT, FIO47-C. - Use valid format strings java.text.MessageFormat | CODE\_SMELL | MAJOR | 1 |
| Field dependency injection should be avoided | Why is this an issue? Dependency injection frameworks such as Spring support dependency injection by using annotations such as @Inject and @Autowired. These annotations can be used to inject beans via constructor, setter, and field injection. Generally speaking, field injection is discouraged. It allows the creation of objects in an invalid state and makes testing more difficult. The dependencies are not explicit when instantiating a class that uses field injection. In addition, field injection is not compatible with final fields. Keeping dependencies immutable where possible makes the code easier to understand, easing development and maintenance. Finally, because values are injected into fields after the object has been constructed, they cannot be used to initialize other non-injected fields inline. This rule raises an issue when the @Autowired or @Inject annotations are used on a field. How to fix it Use constructor injection instead. By using constructor injection, the dependencies are explicit and must be passed during an object’s construction. This avoids the possibility of instantiating an object in an invalid state and makes types more testable. Fields can be declared final, which makes the code easier to understand, as dependencies don’t change after instantiation. Code examples Noncompliant code example public class SomeService { @Autowired private SomeDependency someDependency; // Noncompliant private String name = someDependency.getName(); // Will throw a NullPointerException } Compliant solution public class SomeService { private final SomeDependency someDependency; private final String name; @Autowired public SomeService(SomeDependency someDependency) { this.someDependency = someDependency; name = someDependency.getName(); } } Resources Articles &amp; blog posts Baeldung - Why Is Field Injection Not Recommended? Baeldung - Constructor Dependency Injection in Spring Oliver Drotbohm - Why field injection is evil | CODE\_SMELL | MAJOR | 2 |
| Exceptions in "throws" clauses should not be superfluous | Why is this an issue? Superfluous exceptions within throws clauses have negative effects on the readability and maintainability of the code. An exception in a throws clause is superfluous if it is: listed multiple times a subclass of another listed exception not actually thrown by any execution path of the method Noncompliant code example void foo() throws MyException, MyException {} // Noncompliant; should be listed once void bar() throws Throwable, Exception {} // Noncompliant; Exception is a subclass of Throwable void boo() throws IOException { // Noncompliant; IOException cannot be thrown System.out.println("Hi!"); } Compliant solution void foo() throws MyException {} void bar() throws Throwable {} void boo() { System.out.println("Hi!"); } Exceptions The rule will not raise any issue for exceptions that cannot be thrown from the method body: in interface default methods in overriding and implementating methods in non-private methods that only throw, have empty bodies, or a single return statement. in overridable methods (non-final, or not member of a final class, non-static, non-private), if the exception is documented with a proper JavaDoc interface MyInterface { default void defaultMethod() throws IOException { System.out.println("Hi!"); } void doSomething() throws IOException; } class A implements MyInterface { @Override void doSomething() throws IOException { System.out.println("Hi!"); } public void emptyBody() throws IOException {} protected void singleThrowStatement() throws IOException { throw new UnsupportedOperationException("This method should be implemented in subclasses"); } Object singleReturnStatement() throws IOException { return null; } /\*\* \* @throws IOException Overriding classes may throw this exception if they print values into a file \*/ protected void overridable() throws IOException { // no issue, method is overridable and the exception has proper javadoc System.out.println("foo"); } } Also, the rule will not raise issues on RuntimeException, or one of its sub-classes, because documenting runtime exceptions which could be thrown can ultimately help users of the method understand its behavior. class B { int possibleDivisionByZero(int a, int b) throws ArithmeticException { return a / b; } } | CODE\_SMELL | MINOR | 4 |
| Local variables should not be declared and then immediately returned or thrown | Why is this an issue? Declaring a variable only to immediately return or throw it is considered a bad practice because it adds unnecessary complexity to the code. This practice can make the code harder to read and understand, as it introduces an extra step that doesn’t add any value. Instead of declaring a variable and then immediately returning or throwing it, it is generally better to return or throw the value directly. This makes the code cleaner, simpler, and easier to understand. How to fix it Declaring a variable only to immediately return or throw it is considered a bad practice because it adds unnecessary complexity to the code. To fix the issue, return or throw the value directly. Code examples Noncompliant code example public long computeDurationInMilliseconds() { long duration = (((hours \* 60) + minutes) \* 60 + seconds) \* 1000; return duration; } Compliant solution public long computeDurationInMilliseconds() { return (((hours \* 60) + minutes) \* 60 + seconds) \* 1000; } Noncompliant code example public void doSomething() { RuntimeException myException = new RuntimeException(); throw myException; } Compliant solution public void doSomething() { throw new RuntimeException(); } | CODE\_SMELL | MINOR | 5 |

# 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 |
| Permission | 0 | 0 | 0 |
| SQL Injection | 0 | 0 | 0 |
| Encryption of Sensitive Data | 0 | 0 | 0 |
| Traceability | 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 |
| Path Traversal Injection | 0 | 0 | 0 |
| HTTP Response Splitting | 0 | 0 | 0 |
| Others | 0 | 0 | 0 |

## Security hotspots List