EasyBank Statistical Code Review

This report focuses on analyzing the project's codebase using statistical tools, **Sonarlint/Sonarcloud** to identify potential issues and vulnerabilities. Also to make sure the code is optimized to be easily understandable.

Key metrics:

Project's statistical code analysis relies on several key metrics we found helpful to assess the quality and maintainability of the codebase. Sonarlint code analysis found a total of 213 issues in 37 files (2598 lines). These issues were classified based on severity by color codes, ranging from maintain issues (yellow) to security vulnerabilities (red).

Cyclomatic Complexity calculations indicated that the complexity rating of the code was at a risk level of 182, primarily due to the front-end components. This rating reflects that main front-end components are too complex, and maintainability and testing could create issues. Consequently, minimal changes to the base code were possible to be made in the current timetable.

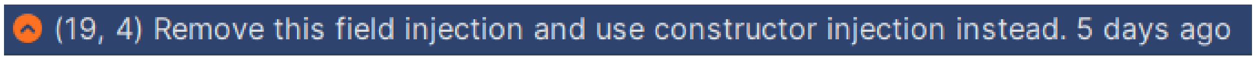
The duplication density was only 2.2%, with 56 duplicated lines identified. Additionally, throughout the coding comment density ratings were lower than desired, which made some of the codes more challenging to understand.

Overall test coverage was executed in 52% of classes, which is moderately good, although there is room for improvement.

Analysis Findings & Refactoring:

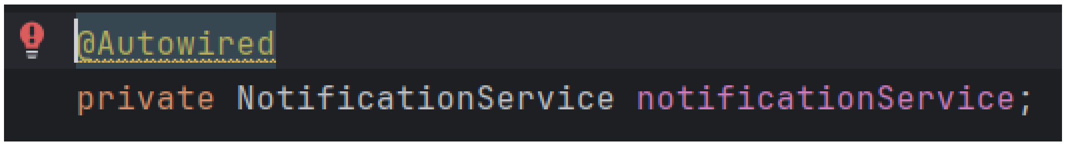
Here are the main conflicts that were detected in multiple occasions:

1. Field injection issues:



```
(19, 4) Remove this field injection and use constructor injection instead. 5 days ago
```

Most of the services and controllers used field injection instead of constructor injection, which is a best practice in Spring Boot because it enforces good software design principles such as **dependency visibility, immutability, and testability.**



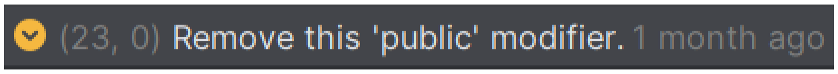
```
@Autowired
private NotificationService notificationService;
```

Sonarlint analysis:

Dependency injection frameworks such as Spring, Quarkus, and others 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.

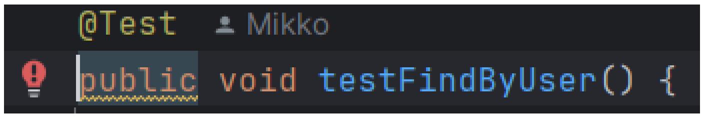
Generally speaking, field injection is discouraged. It allows the creation of objects in an invalid state and makes testing more difficult. The dependencies are not explicit when instantiating a class that uses field injection.

2. Public modifiers:



```
(23, 0) Remove this 'public' modifier. 1 month ago
```

Many test classes had unnecessary ‘public’ modifiers implemented in test methods and class names. It is better to stick with the default package-private visibility to maintain **cleaner code**, **better encapsulation**, and **alignment with modern testing practices.**

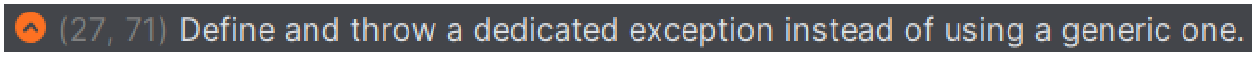


```
@Test   Mikko
public void testFindByUser() {
```

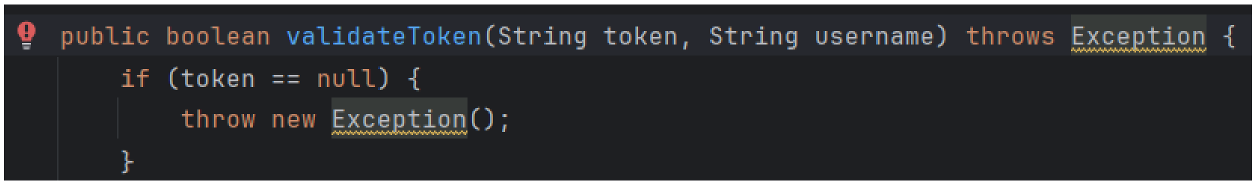
Sonarlint analysis:

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.

3. Generic exceptions:


(27, 71) Define and throw a dedicated exception instead of using a generic one.

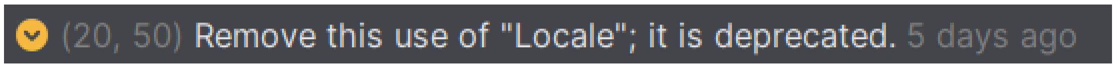
Generic exceptions were thrown out on multiple occasions. Catching **Exception** or **Throwable** hides the actual cause of the error, making it harder to debug and handle correctly.


```
public boolean validateToken(String token, String username) throws Exception {
    if (token == null) {
        throw new Exception();
    }
```

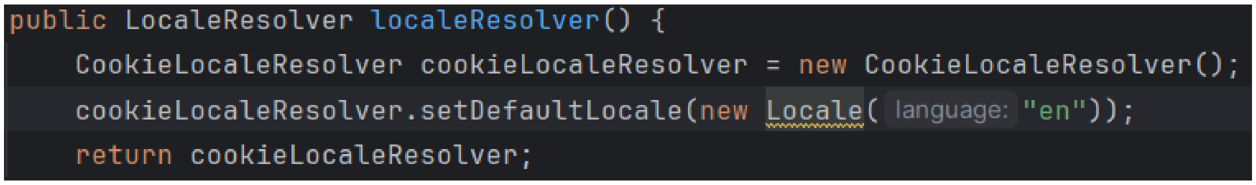
Sonarlint analysis:

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 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.

4. Deprecated Locale


(20, 50) Remove this use of "Locale"; it is deprecated. 5 days ago

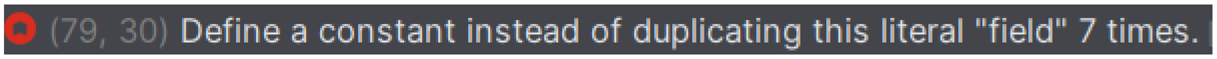
Deprecated objects were used in many cases while implementing localization. Code below does not allow for easy configuration of multiple locales or dynamic handling of locales in a more robust way.


```java
public LocaleResolver localeResolver() {
    CookieLocaleResolver cookieLocaleResolver = new CookieLocaleResolver();
    cookieLocaleResolver.setDefaultLocale(new Locale( language: "en"));
    return cookieLocaleResolver;
```

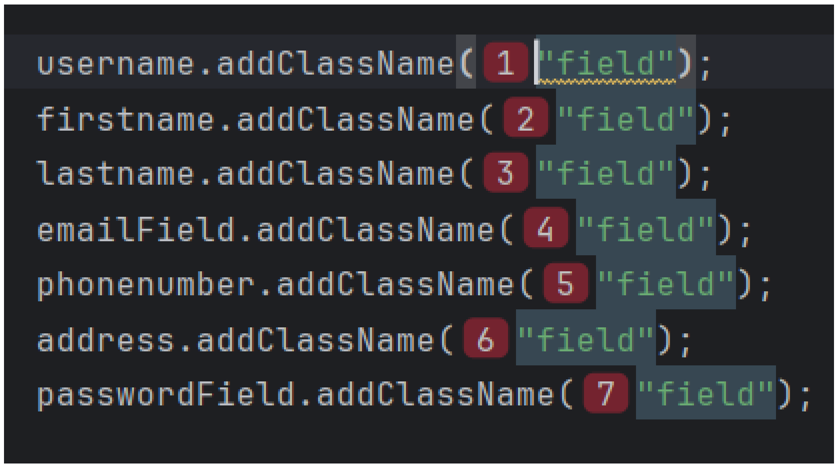
Sonarlint analysis:

Code is sometimes annotated as deprecated by developers maintaining libraries or APIs to indicate that the method, class, or other programming element is no longer recommended for use. This is typically due to the introduction of a newer or more effective alternative. For example, when a better solution has been identified, or when the existing code presents potential errors or security risks.

5. Code Duplication:

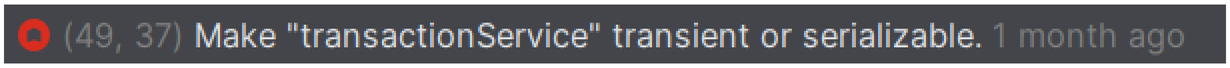

```
(79, 30) Define a constant instead of duplicating this literal "field" 7 times.
```

There were many occasions where the same literal was used instead of a constant. By using constants instead of literals, you improve code maintainability, readability, reduce errors, and facilitate easier changes in the future.


```java
username.addClassName( 1 "field");
firstname.addClassName( 2 "field");
lastname.addClassName( 3 "field");
emailField.addClassName( 4 "field");
phonenumber.addClassName( 5 "field");
address.addClassName( 6 "field");
passwordField.addClassName( 7 "field");
```

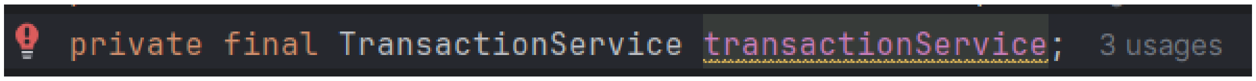
Sonarlint analysis: Duplicated string literals make the process of refactoring complex and error-prone, as any change would need to be propagated on all occurrences.

5. Serialization issues:



```
(49, 37) Make "transactionService" transient or serializable. 1 month ago
```

Some controllers and services were not serialized correctly. When the object is serialized, the service or controller object will not be included in the serialized form, avoiding issues with non-serializable objects.



```
private final TransactionService transactionService; 3 usages
```

Sonarlint analysis: By contract, fields in a Serializable class must themselves be either Serializable or transient. Even if the class is never explicitly serialized or deserialized, it is not safe to assume that this cannot happen. For instance, underload, most J2EE application frameworks flush objects to disk.

Conclusion:

Although there were many small and not so small errors in our project, we managed to sort them out, at a brisk pace and without any issue. In the beginning, this task seemed quite insurmountable as there were over two hundred of these errors, but to our luck, none of them were too serious. This in turn created a cycle that was easy to manage and conclude in a timely manner.

The race was good, but the car was bad. -Lenni Liu 26/11/2024