ESCUELA COLOMBIANA DE INGENIERIA
JULIO GARAVITO

IT SECURITY AND PRIVACY
GROUP 1L

LABORATORY 10

SUBMITTED BY:
JUAN PABLO FERNANDEZ GONZALES
MARIA VALENTINA TORRES MONSALVE

SUBMITTED TO:
Eng. DANIEL ESTEBAN VELA LOPEZ

BOGOTÁ D.C.
DATE:
31/03/2025

Introduction

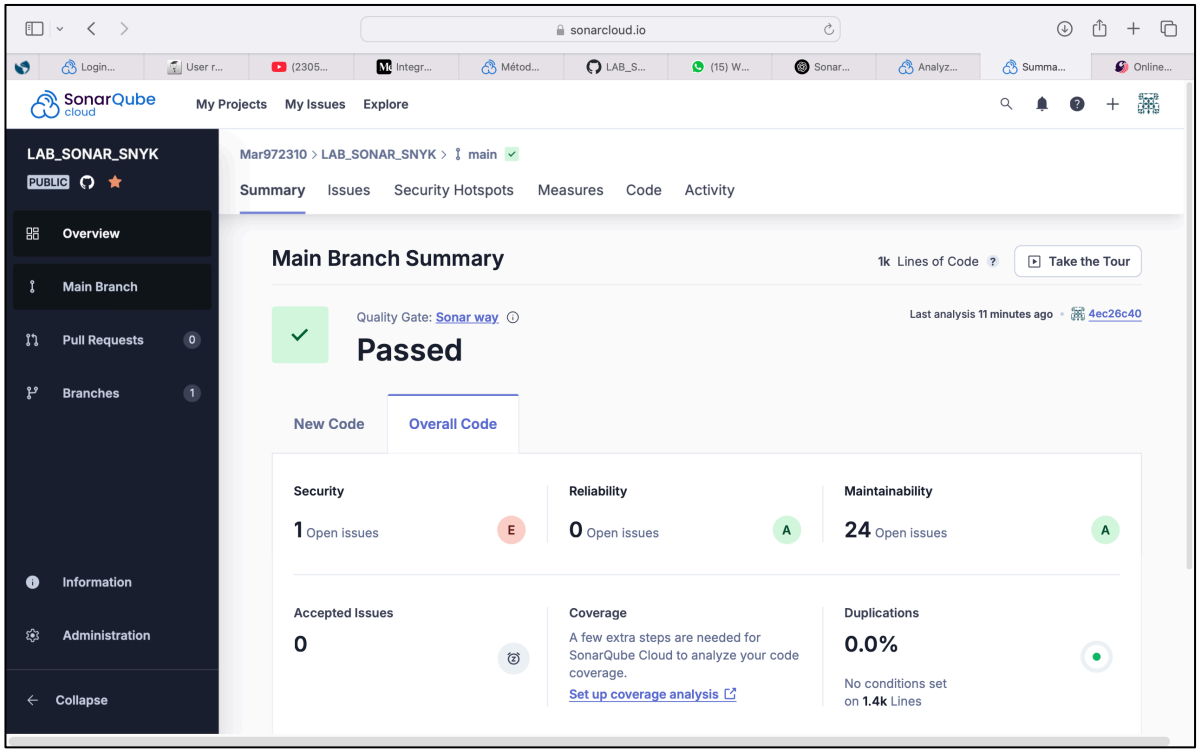
This report analyzes the source code of a real estate property management system to identify security vulnerabilities and ensure clean, maintainable code. The project implements a **CRUD (Create, Read, Update, Delete) system** using a **backend in Spring Boot** and a **frontend developed with HTML, CSS and JavaScript**. The data is stored in a **MySQL database**.

To assess code security and quality, **SonarCloud** is used to detect code smells, bugs, and vulnerabilities, and **Snyk** to identify security risks in dependencies and configurations. The goal is to improve project reliability, performance, and safety, addressing potential weaknesses and following development best practices.

The application we will be using for the analysis is in the repository: ***https://github.com/Mar972310/LAB_SONAR_SNYK.git***

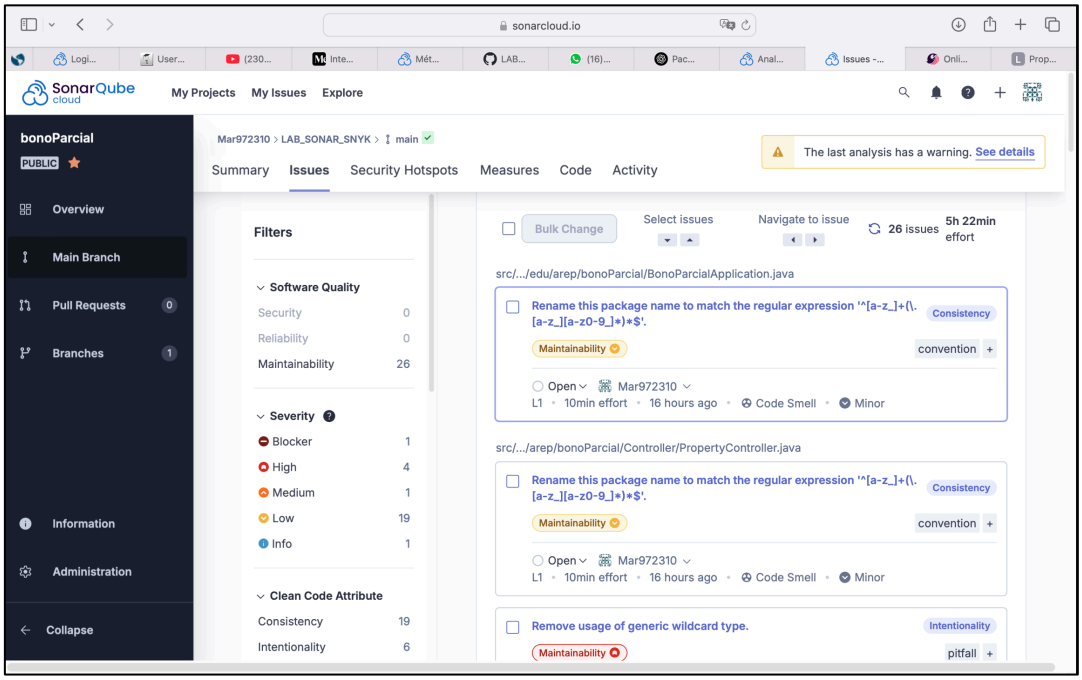
SonarCloud Analysis

The analysis carried out with SonarCloud to the Mar972310/LAB_SONAR_SNYK repository, gave us as an answer after having analyzed the code of the application that 24 maintainability issues and one security issue.



- **Correcting the naming of packages according to Java conventions**

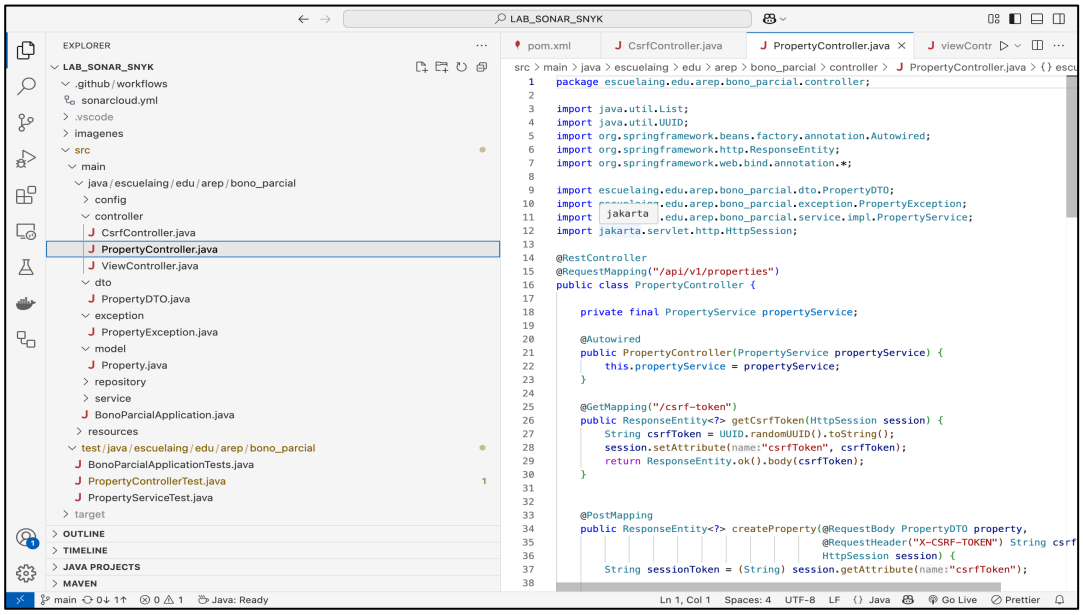
Among the maintainability issues we have that 9 of those refer ***to Rename this package name to match the regular expression '^[a-z_]+(\.[a-z_][a-z0-9_]*)*$'.*** This issue indicates that the name of the packets does not comply with the convention set in the regular expression ^[a-z_]+(\.[a-z_][a-z0-9_]*)*$, which requires package names to begin with lowercase letters or underscores and that subpackages, separated by periods, also begin with a lowercase letter or underscore, allowing only lowercase letters, numbers, and underscores in the rest of the name. No capitalization, hyphens (-), or spaces are allowed. For example, com. ExampleApp is incorrect because it uses all caps, whereas com.example_app would be a valid version.



In the case of our application we have that the names of the packages are as follows:

- escuelaing.edu.arep.bonoPartial
- escuelaing.edu.arep.bonoPartial.Controller
- escuelaing.edu.arep.bonoPartial.DTO
- escuelaing.edu.arep.bonoPartial.Exception
- escuelaing.edu.arep.bonoPartial.Repository
- escuelaing.edu.arep.bonoPartial.Service.Impl
- escuelaing.edu.arep.bonoPartial.Service
- escuelaing.edu.arep.bonoPartial.model

What we will do is correct ***PartialBonus*** for ***bono_parcial***, the subpackages that start with a capital letter, for example, Controller, DTO, Exception, Repository, Service and Impl we will leave them as follows: controller, dto, exception, repository, service and impl

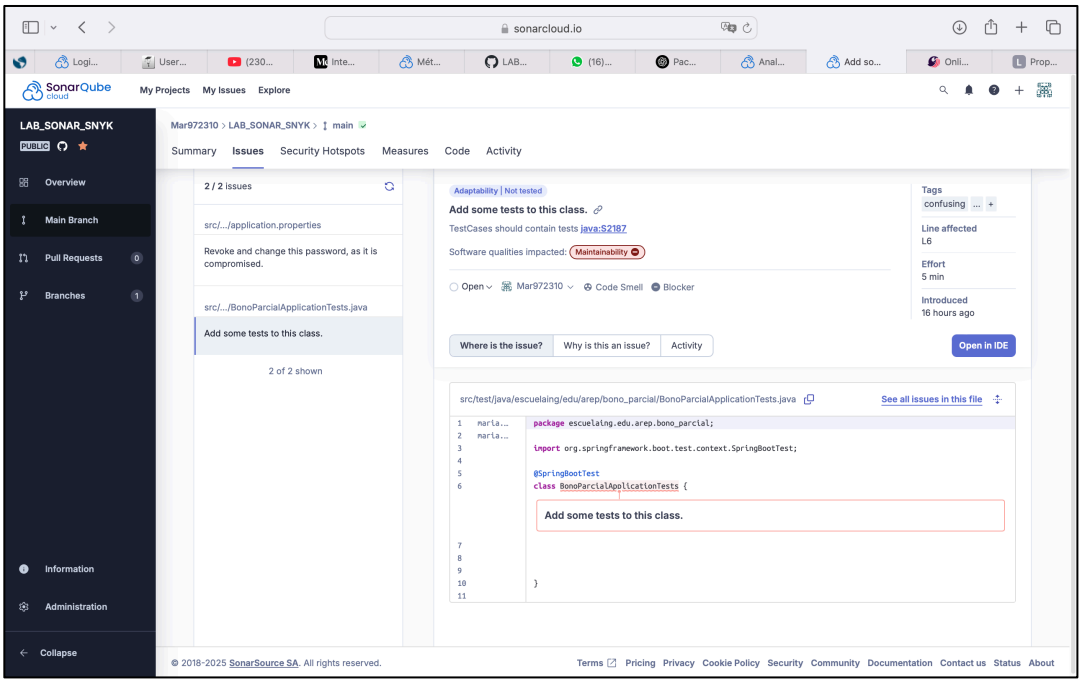


- **Correction of naming in classes according to Java conventions**

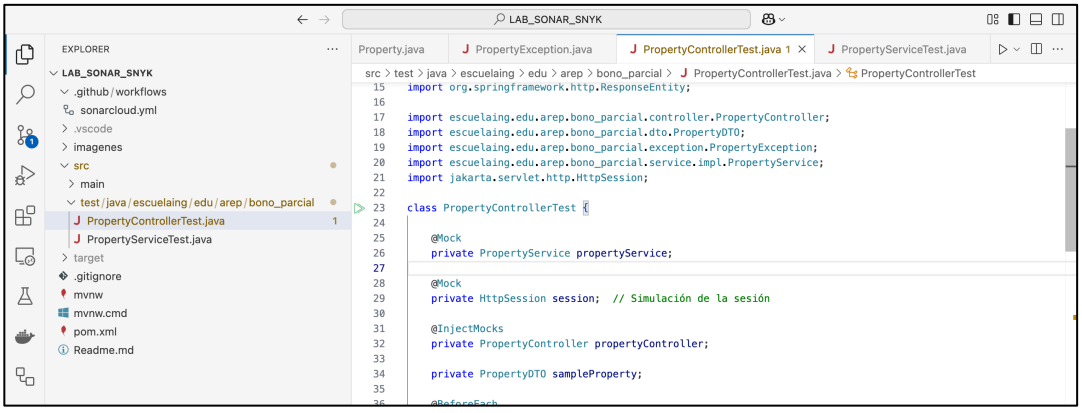
Another problem related to the convention established by the regular expression ^[a-z_]+(\.[a-z_][a-z0-9_]*)*$ is that the viewController.java class does not comply with Java best practices, since its name does not start with a capital letter. According to naming conventions in Java, class names should start with an uppercase letter, so the proper correction would be *ViewController.java*.

- **Deleting classes without code or functionality**

In the test package of the application we have a class in which we are not implemented and there is only the empty class.



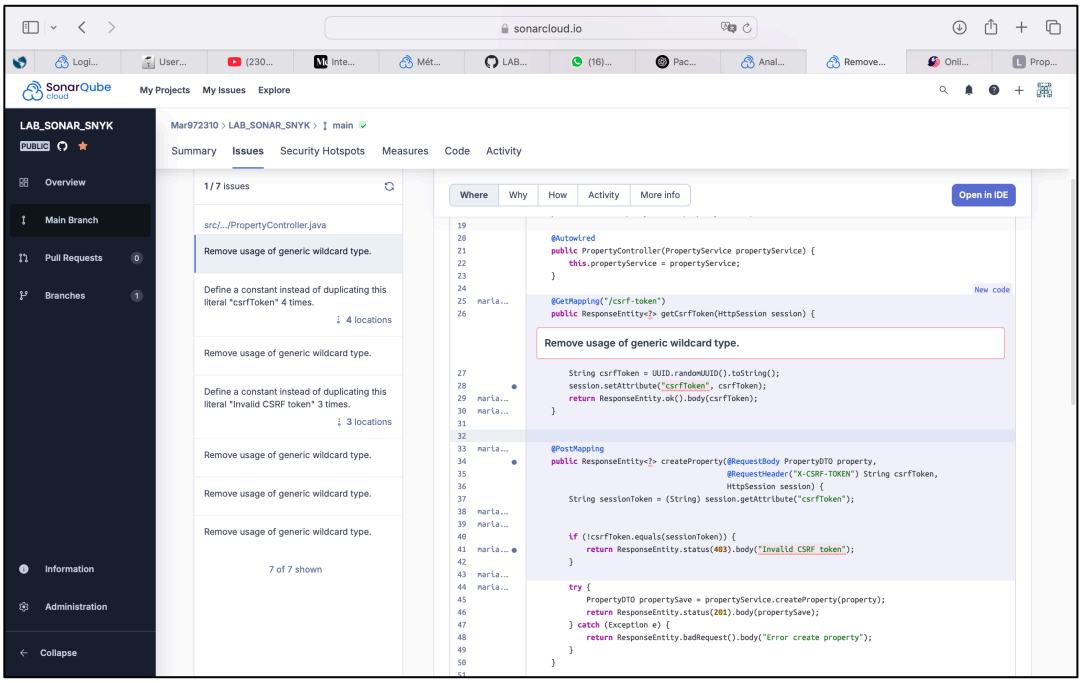
As the tests were implemented in other classes, we can eliminate this one and avoid this issue and a more orderly project structure with what is necessary.



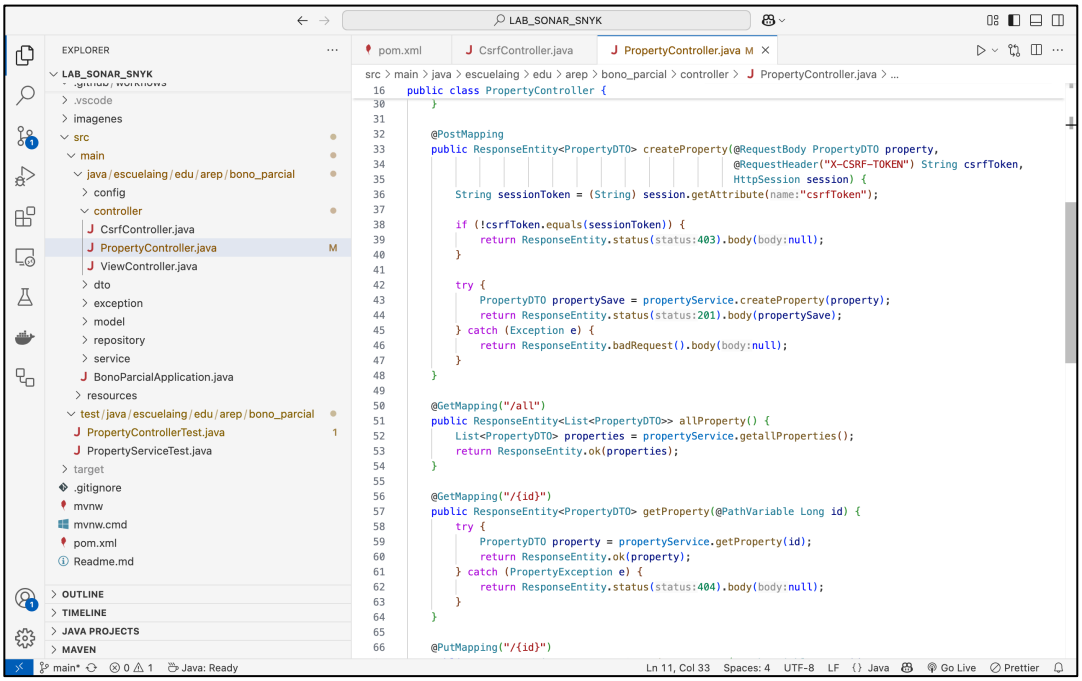
- **Remove usage of generic wildcard type.**

The problem lies in the use of wildcards (?) in the return type of a Java method, which prevents the type from being properly constrained in any context. This is due to the invariance of generic parameters in Java, which means that they cannot behave either covariately or countervariately when used in methods with input and output. As a

result, a wildcard return does not allow safe manipulation of the returned type, indicating that the developer's intent was likely different and should be corrected by using a more specific type.



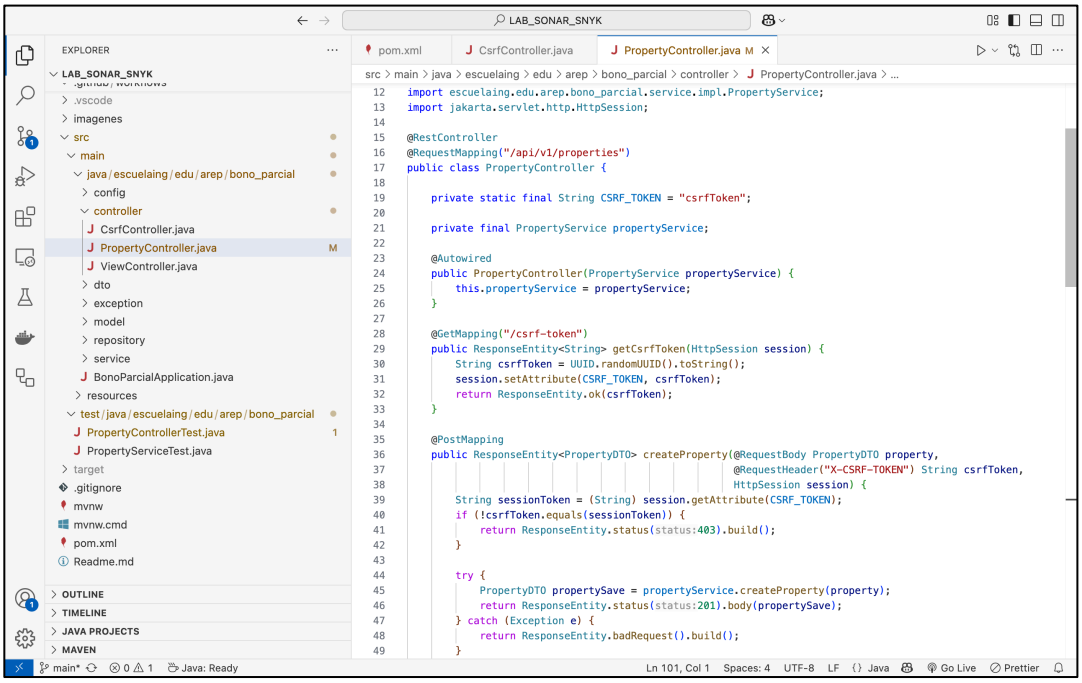
In class ***PropertyController.java***, we initially used wildcards in return types due to uncertainty about the specific values we needed to return on each endpoint. However, now that development is complete and we have clarity on the types of data that should be returned by API endpoints, we can replace wildcards with accurate and definitive return types. This improves readability, makes code easier to maintain, and avoids issues related to type variance in Java.



- **Define a constant instead of duplicating this literal "csrfToken" 4 times.**

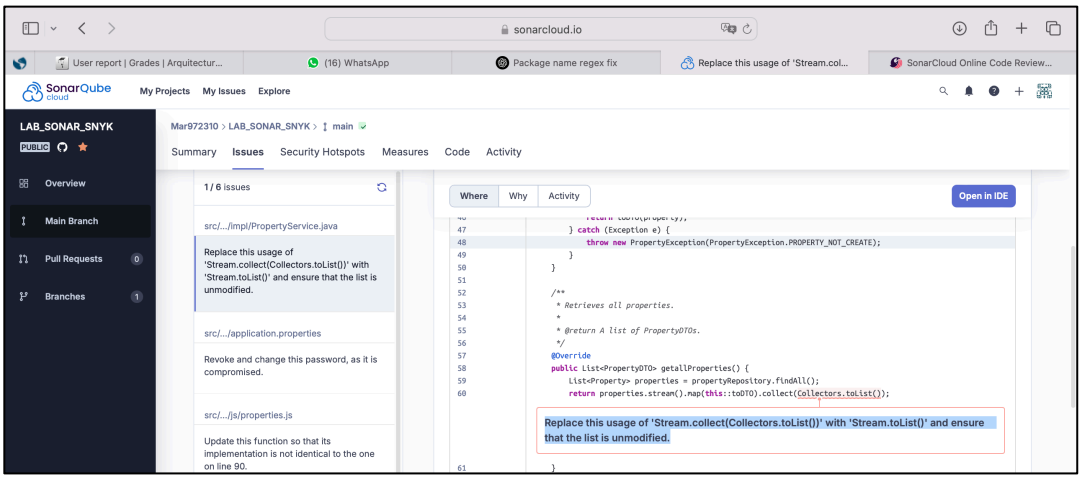
This issue in class ***PropertyController.java*** is the duplication of the "csrfToken" literal in four different places within the code. Repeating the same value in multiple parts of the code can lead to maintenance issues and increase the risk of errors in future modifications. The best practice to avoid this repetition is to define a constant with a descriptive name and use it instead of the literal value.

To work around this issue, we've declared a CSRF_TOKEN constant of type static final String, assigning it the value "csrfToken". Then, we replace all occurrences of the repeated string with the constant. In this way, if at any time it is necessary to modify the name of the key used to store the CSRF token in the session, it will only be necessary to change it in a single place, improving the readability and maintainability of the code.

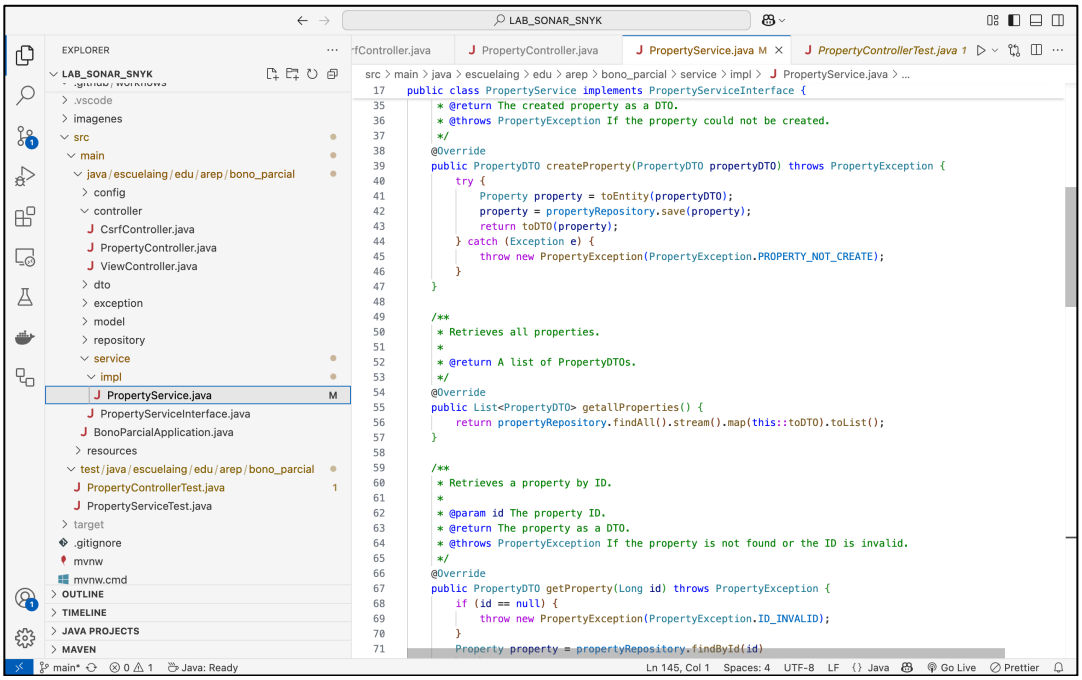


- **Replace this usage of 'Stream.collect(Collectors.toList())' with 'Stream.toList()' and ensure that the list is unmodified.**

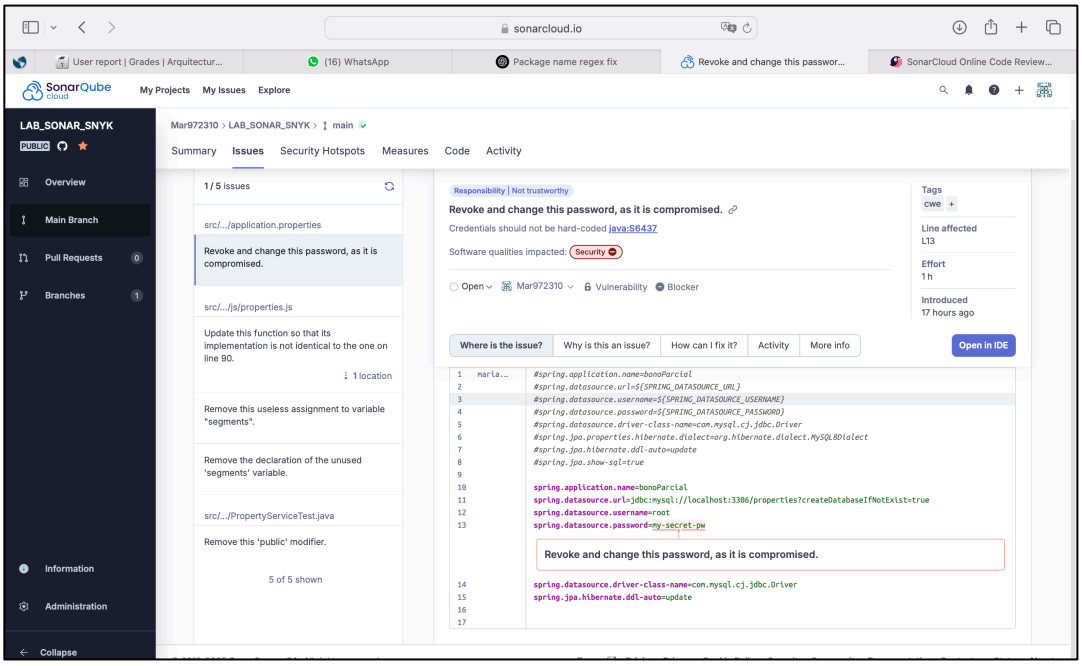
The issue identified is related to the use of Stream.collect(Collectors.toList()), which is an older, more verbose way to convert a Stream to a list. Since Java version 16, the Stream.toList() method has been introduced, which offers a more concise and efficient alternative. This method returns an immutable list, ensuring that it cannot be modified after it is created.



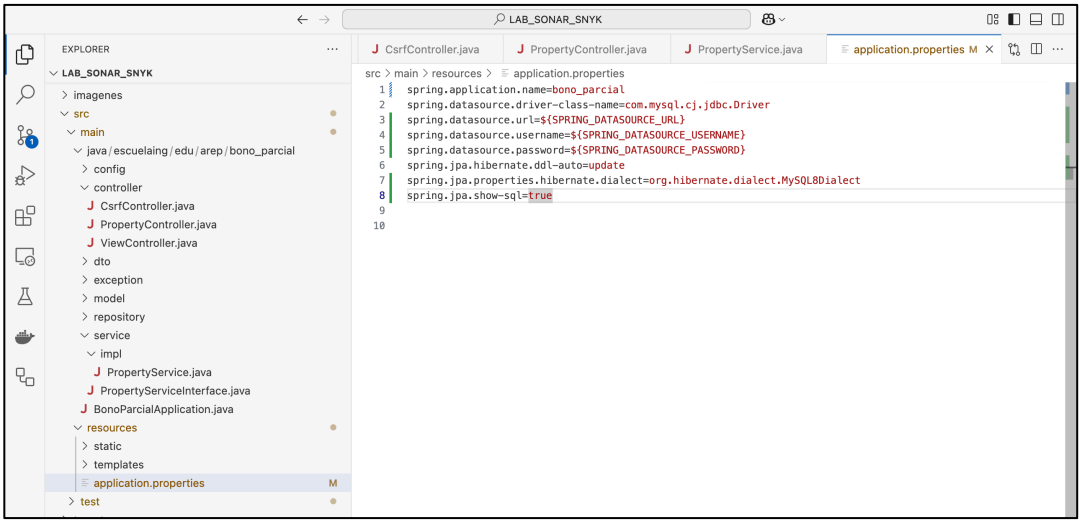
To fix this issue, we'll replace all instances of Stream.collect(Collectors.toList()) with Stream.toList(). This in addition to reducing the complexity of the code. It will also improve security by preventing unexpected modifications to the resulting list.



- Currently, we are performing the database configuration in the application includes credentials and connection details in the application.properties file. This poses a security risk, as credentials can be exposed if the code is shared or stored in a public repository. In addition, this rigid configuration makes it difficult to porte the application between different environments (development, test, and production), as any changes to the database credentials or URL require directly modifying the configuration file.

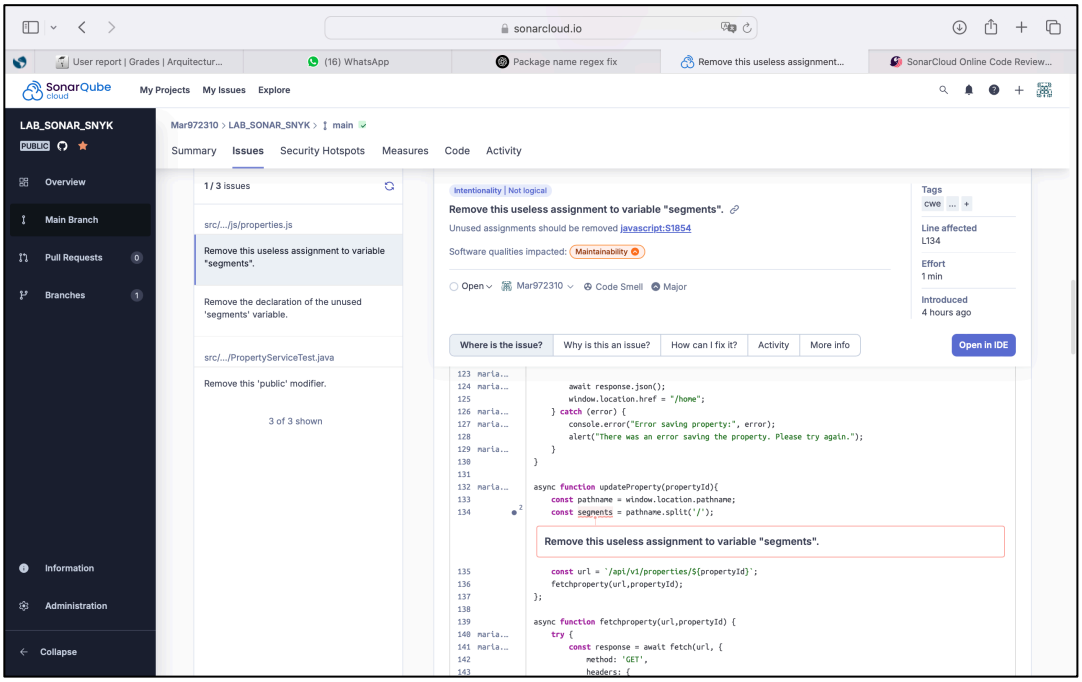


To work around this problem, we're going to replace the plaintext values with environment variables ${SPRING_DATASOURCE_URL}, ${SPRING_DATASOURCE_USERNAME},${SPRING_DATASOURCE_PASSWORD}) , so that your app dynamically gets the settings based on the environment it's running in. This will allow us to improve security by not exposing credentials in the code and will make the deployment more flexible and scalable.

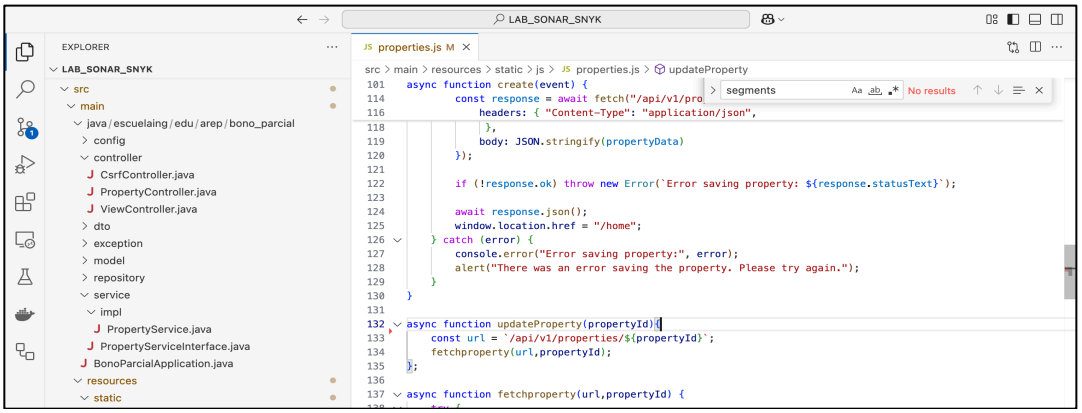


- **Remove this useless assignment to variable "segments"**

SonarCloud has identified an unnecessary mapping to the segments variable, indicating that its value is not used anywhere in the code. These types of unnecessary mappings affect the clarity and efficiency of your code, as they introduce elements that don't provide functionality and can lead to confusion in future revisions. In addition, maintaining redundant code can negatively impact the maintainability and scalability of the project.

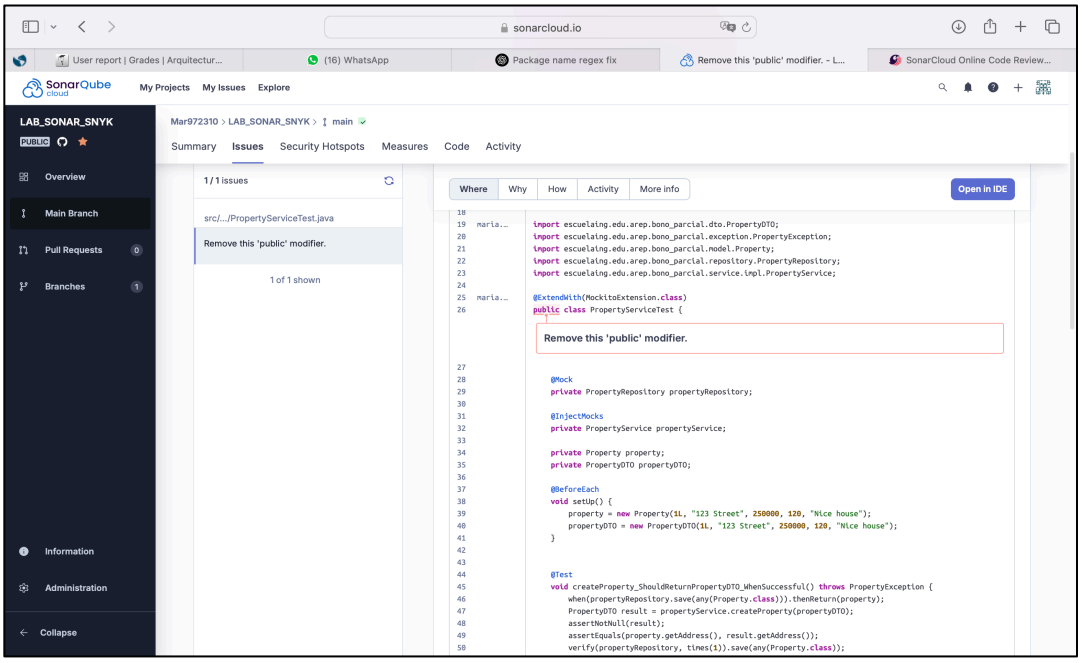


To resolve this issue, we'll remove the mapping of the segments variable if it's not used elsewhere in the code. This fix allows us to maintain cleaner code, reduce unnecessary complexity, and improve the overall quality of our codebase, aligning with the best practices recommended by SonarCloud.

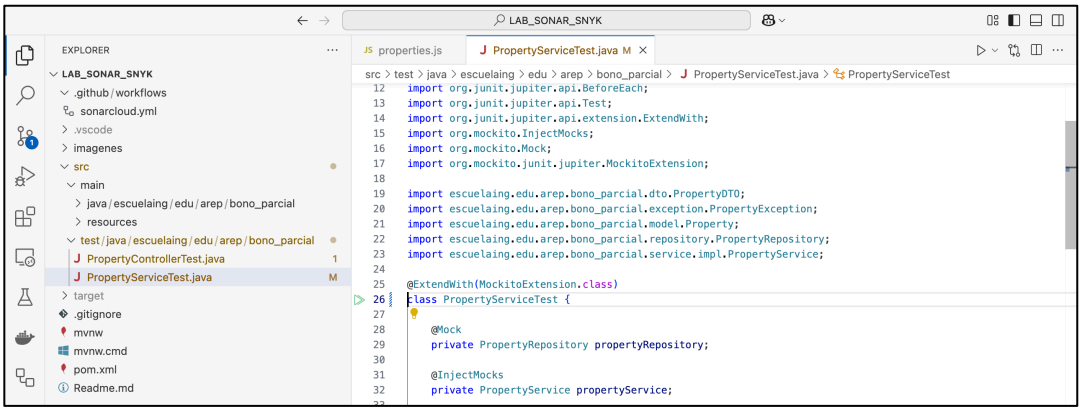


- **Remove this 'public' modifier.**

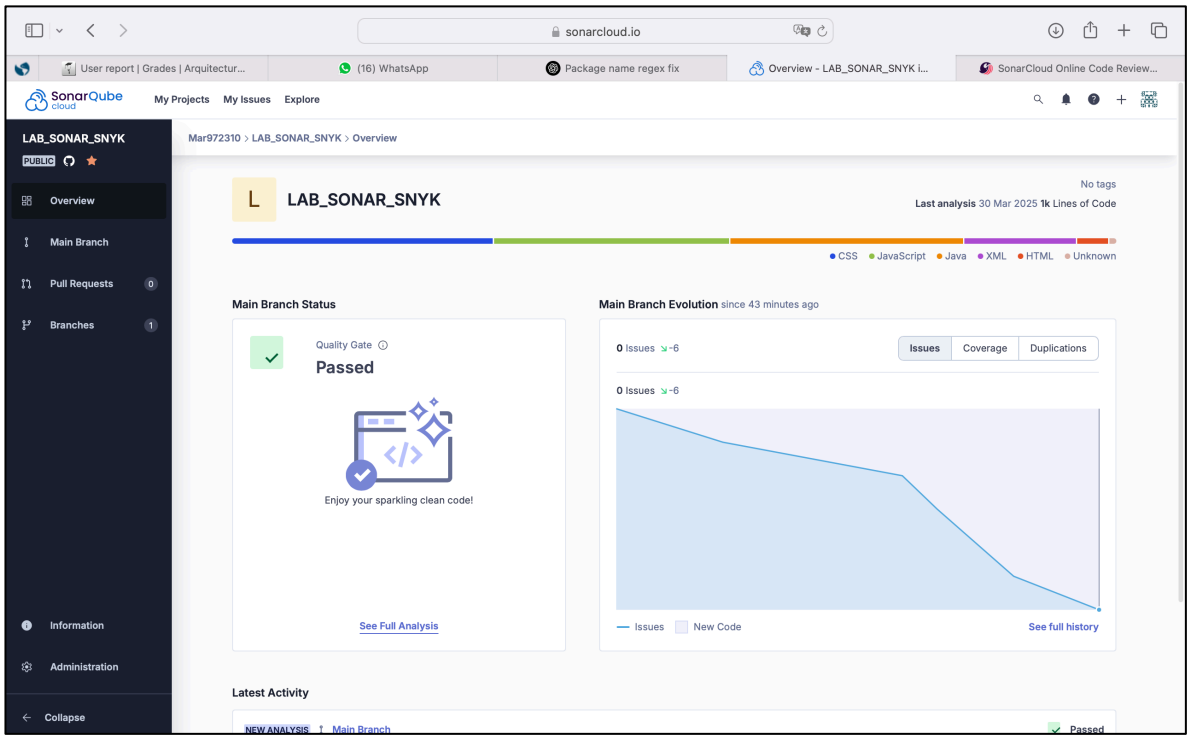
In JUnit 5, the classes and test methods do not need to be public, unlike in JUnit 4. Keeping them public can affect the readability of the code and does not follow recommended conventions. In addition, in the case of private methods, JUnit 5 ignores them without warning, which can lead to reliability issues in the tests.



To improve readability and follow best practices, we must remove the public modifier from our classes and test methods, unless there is a technical reason to keep it. Using the default visibility (package-private) is enough for JUnit 5 to recognize and execute tests correctly.

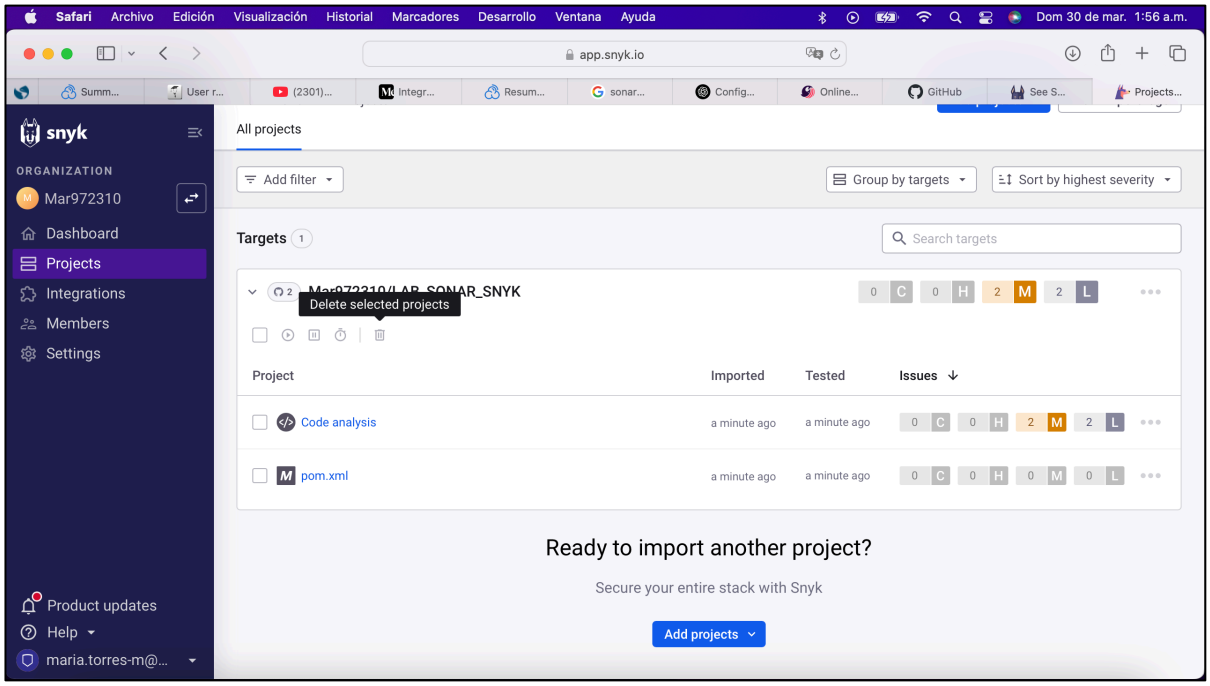


After having corrected all these issues, we will be able to see that the code complies with all good code practices.



Snyk Analysis

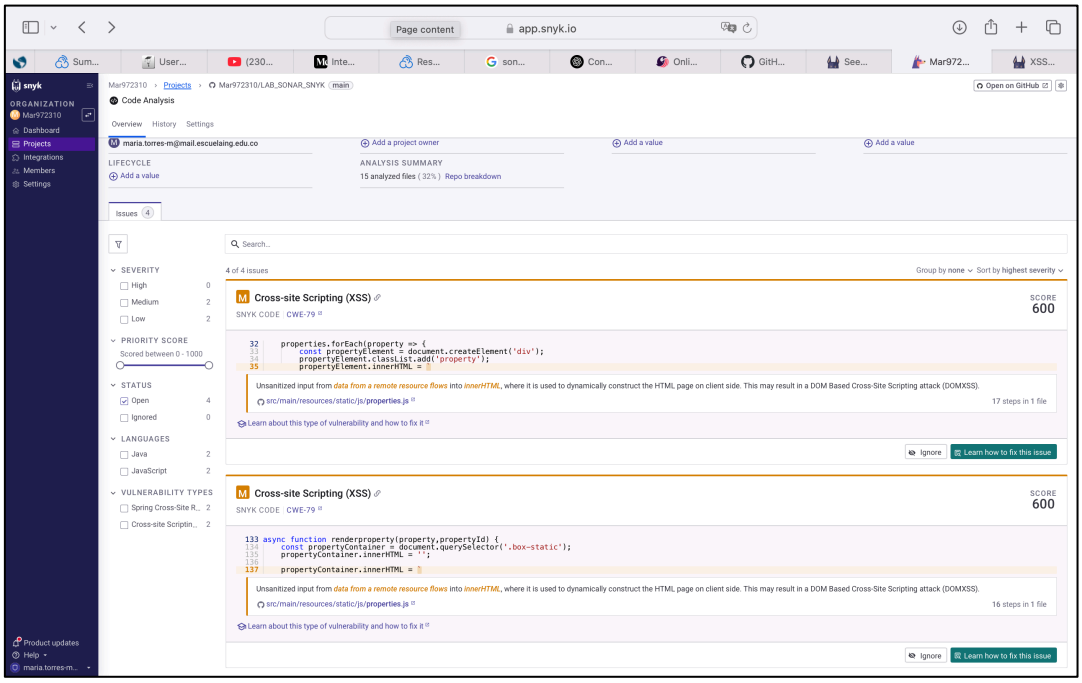
The analysis carried out with Snyk to the Mar972310/LAB_SONAR_SNYK repository, gave us as an answer after having analyzed the application code that we have two Medium risk vulnerabilities and two low risk vulnerabilities.



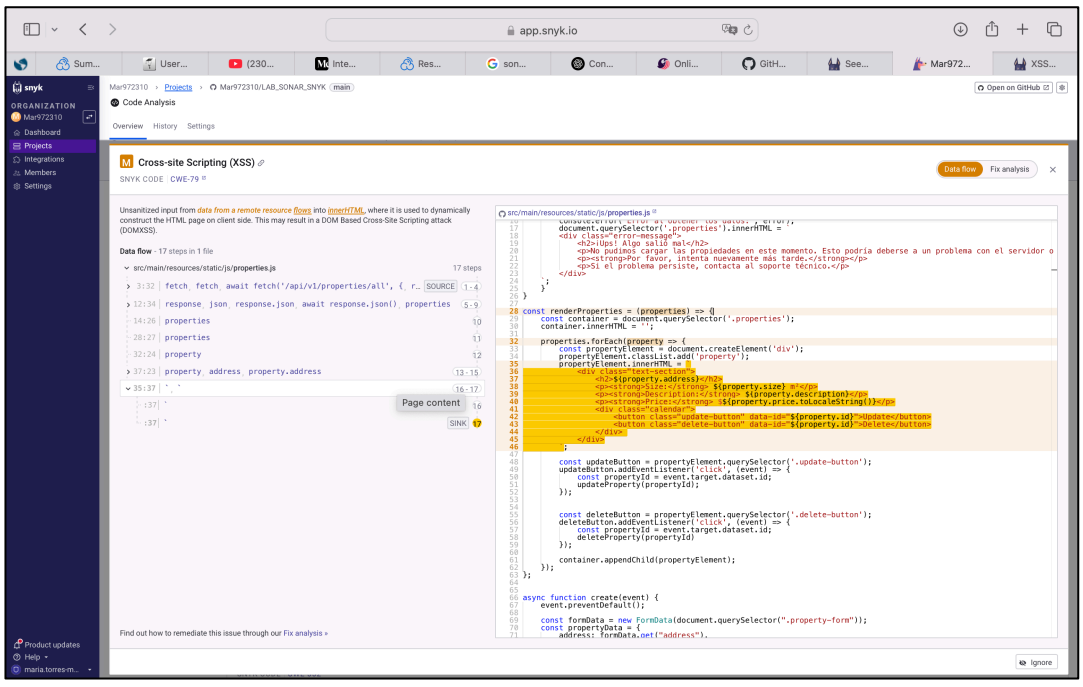
Below we will be analyzing each of the issues and making the corresponding corrections.

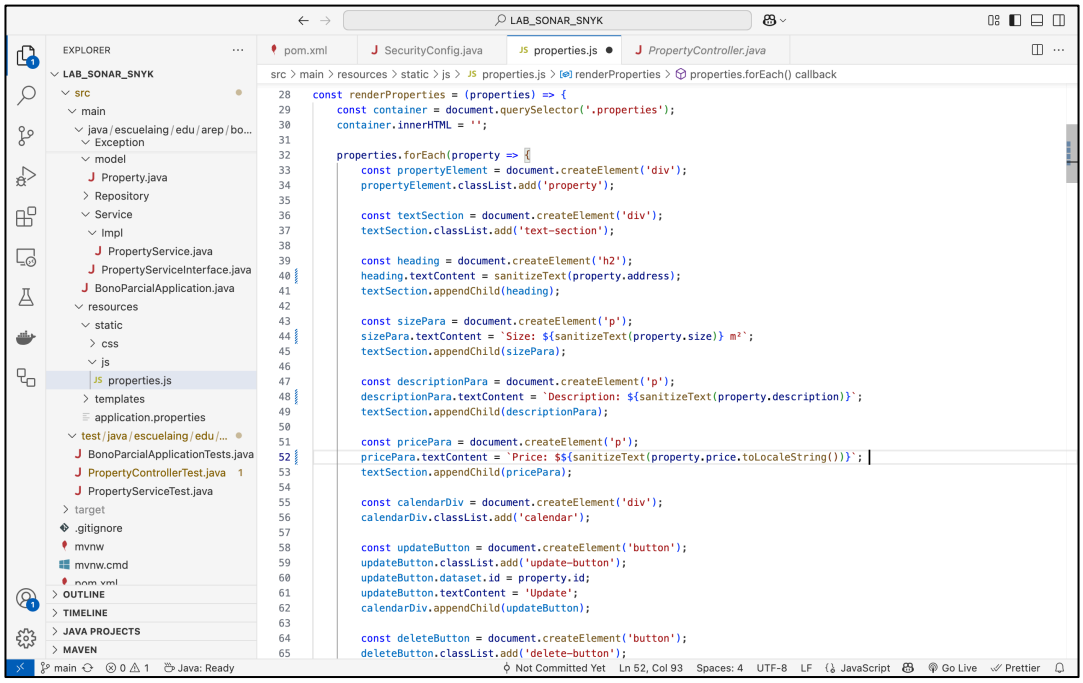
- **DOM-BASED CROSS-Site Scripting (DOM XSS)**

  Medium-risk issues are caused by receiving data from a remote resource flow, such as responses from an external server and APIs, and we are injecting them directly into innerHTML without performing any validation or sanitization of all the data that was received, opening up the possibility of a ***DOM-based Cross-Site Scripting (DOM XSS)*** attack. This type of attack is orchestrated entirely on the client side, i.e., in the user's browser, without being detected by the server. To work around this vulnerability, untrusted data should not be inserted directly into innerHTML. Instead, it's best to use textContent, which inserts content as text without interpreting HTML.

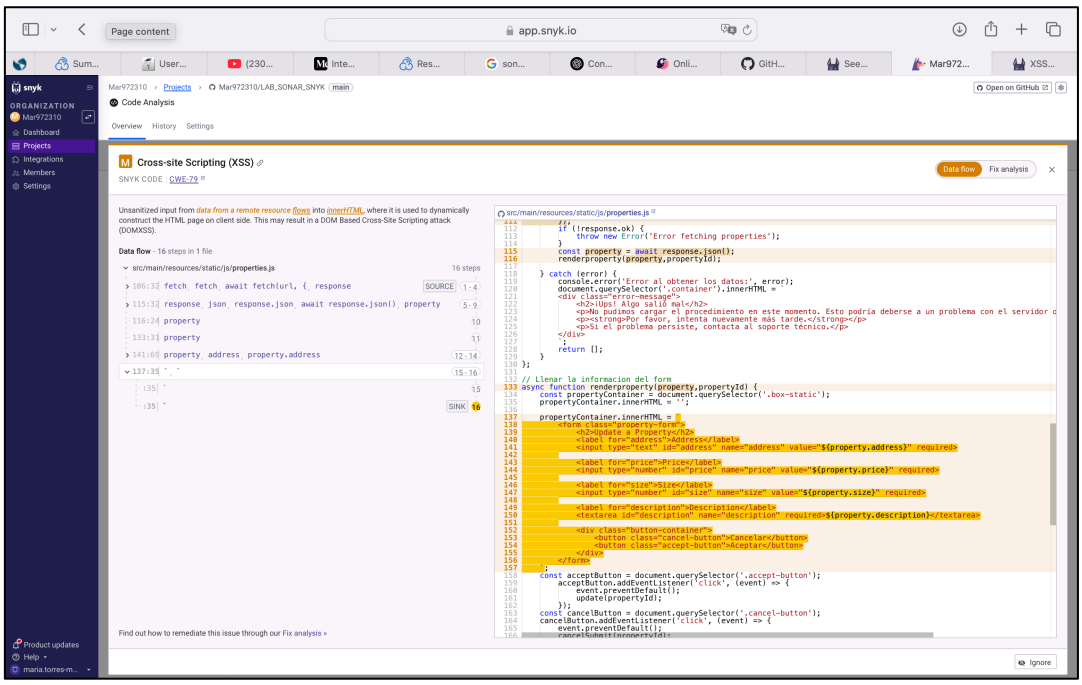


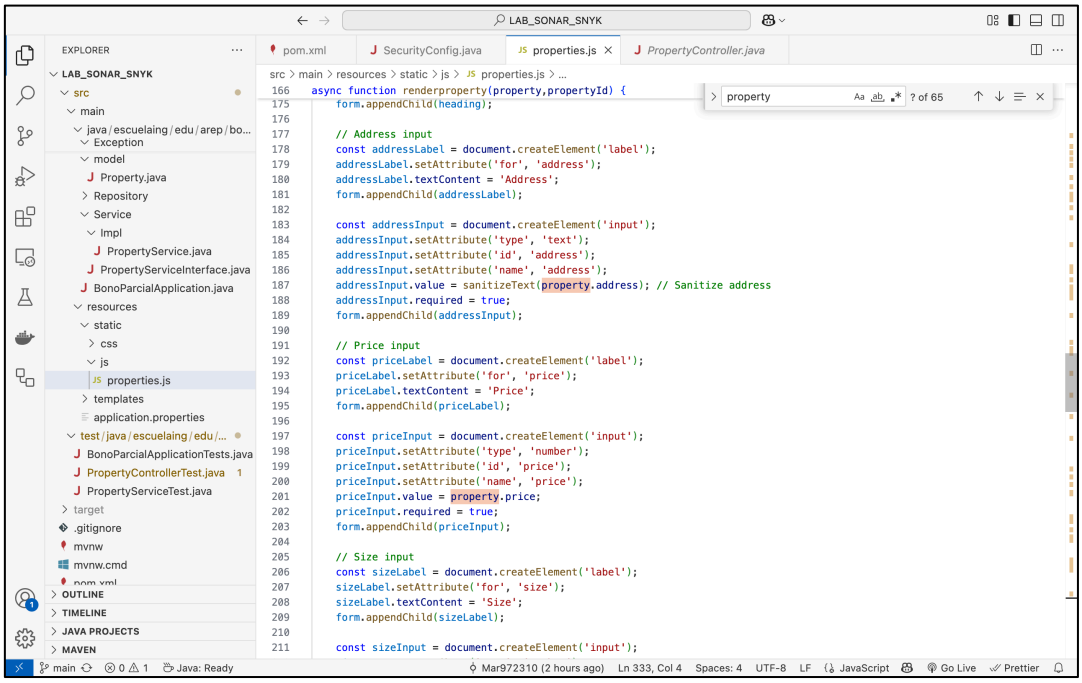
1. RenderProperties function, this is one of the functions that is being used to render the list of properties that are registered in the application's database, the information is being obtained from an endpoint of the properties API.





2. RenderProperty function, this is a function that is being used to render a specific property that is registered in the application's database, like the previous one, the information is being obtained from an endpoint of the properties API.

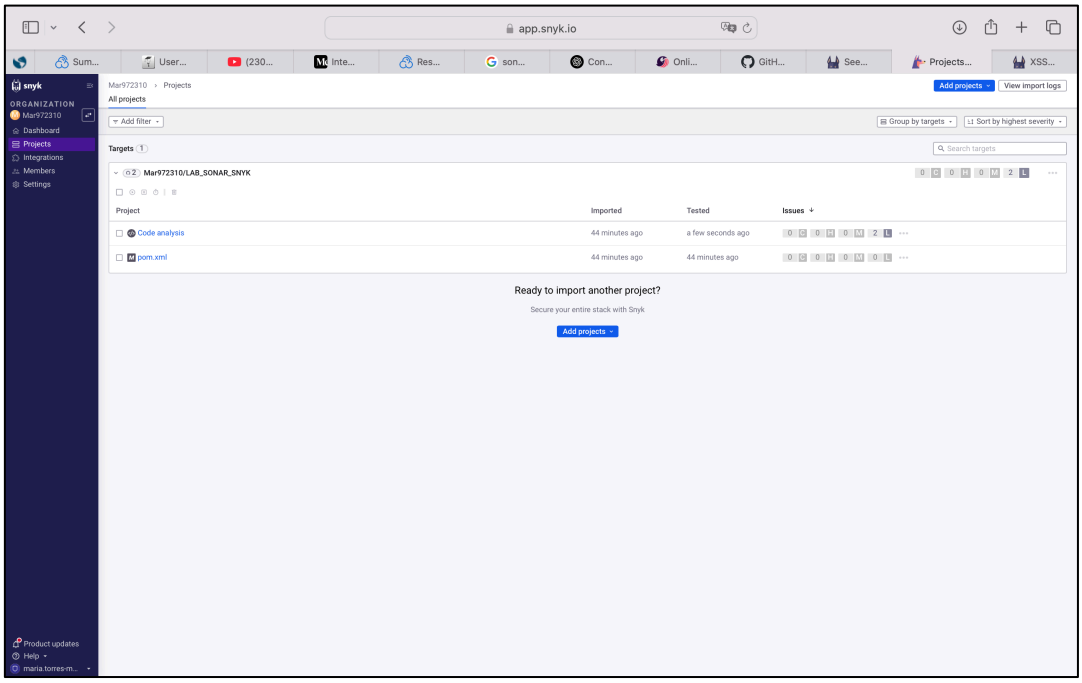




With these fixes we already mitigate the vulnerability of DOM-based Cross-Site Scripting, after uploading the changes to the repository we can see how SNYK performs the analysis again and we no longer have those issues.

- ## Cross-Site Request Forgery (CSRF)

The next vulnerabilities that the analysis shows us are two ***Cross-Site Request Forgery (CSRF)*** vulnerabilities that can generate a type of attack in which an attacker tricks an authenticated user into performing unwanted actions in a web application without their consent. This occurs because the user's browser automatically sends cookies and authentication headers with each request to a website on which they are authenticated, which an attacker can exploit to execute malicious actions on the user's behalf.



In a Spring application, if CSRF protection is not implemented, an attacker could, for example, send a POST request from a malicious website to change a user's password or make a bank transfer without their authorization. Since the user is already authenticated, the application would process the request as legitimate.

In our application, before making the changes, since we do not have an authentication system, anyone could send data to the server and make malicious insertions to the databases. For this we will have to add the security dependency provided by the Spring-boot framework.

1. In the pom.xml we will add this dependency.

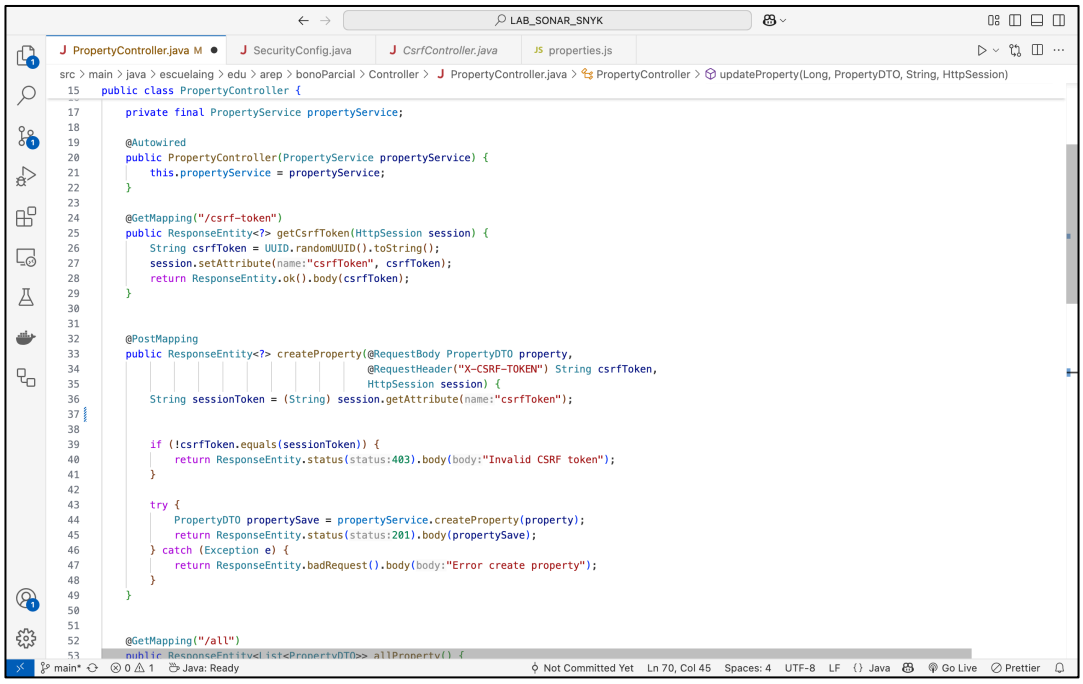
```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

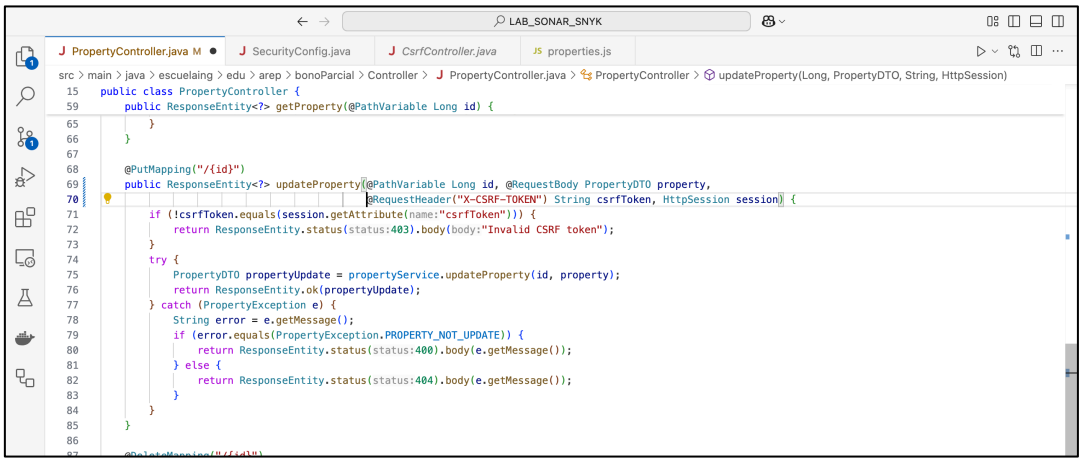
2. We created a class to configure everything related to security restrictions and access permissions, since adding this dependency in the pom will enable us to have a username and password to perform the authentications by default.

3. To the Property Controller to the POST, PUT and DELETE requests we will have to add parameters to receive the CSRF token and the session, with this we can control if an attacker tries to send malicious requests on behalf of the user without

having the correct token, the checks will fail and the request will be rejected.

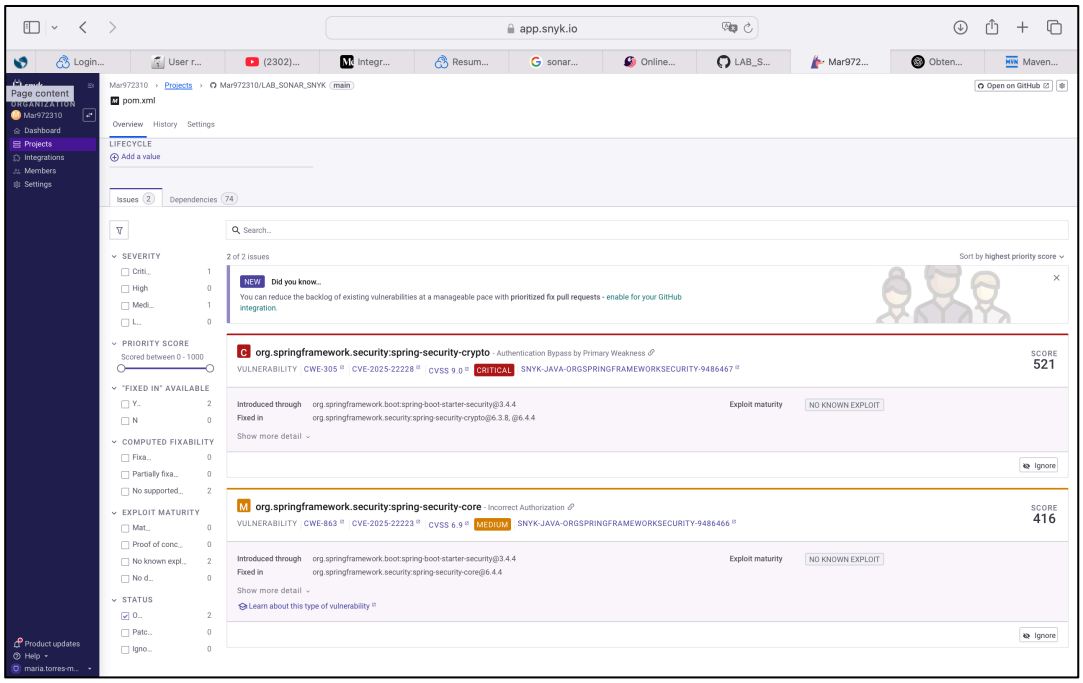
This means that any user who logs into the browser will receive a unique CSRF token and will need to submit it on each protected request. Even if there is no authentication, this verification prevents attacks where an external site tries to send malicious requests on behalf of the user who has an active session in our application.





Going back to analyze the platform, it throws us two new issues in the pom.xml, the first vulnerability in org.springframework.security:spring-security-crypto since it is affecting the BCryptPasswordEncoder.matches() function, which only considers the first 72 characters of a password when comparing it with its hash. This allows

different passwords with the same first 72 characters to be accepted as valid, facilitating brute force attacks and possible authentication bypasses.



Additionally, it threw another vulnerability in org.springframework.security:spring-security-core can affect authorization in our application if we use @EnableMethodSecurity in parameterized methods or types. The issue arises because the findClosestMethodAnnotations() function does not correctly search for security annotations in legacy methods, interfaces, or parameterized superclasses, which could allow certain methods to bypass security restrictions. However, our application would not be vulnerable if we do not use @EnableMethodSecurity, if we do not protect parameterized methods or if we always place the annotations directly in the methods we want to secure.

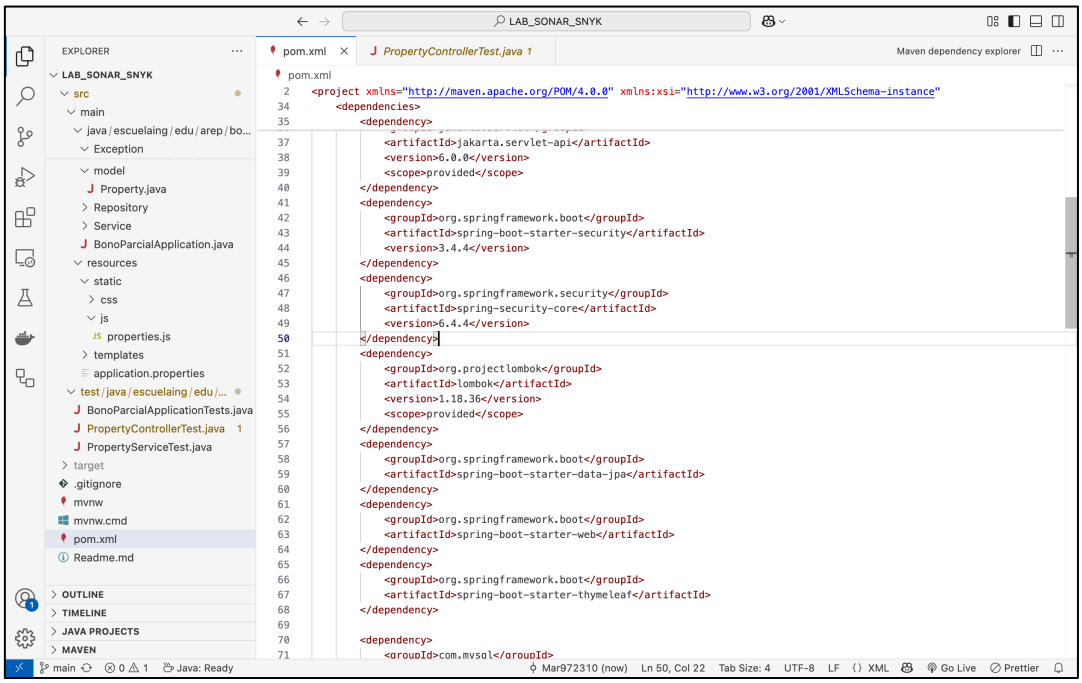
To mitigate both vulnerabilities found in ***spring-security-crypto*** and ***spring-security-core***, we have added the updated dependencies that fix these issues, ensuring that our application is protected even if we do not implement user authentication and access is free. Even if we do not handle credentials, it is crucial to prevent possible authorization bypasses and failures in password validation that could compromise the security of the system. With these updates, we strengthen the integrity of our application, avoiding risks associated with vulnerable versions of Spring Security and ensuring a more secure environment for its execution.

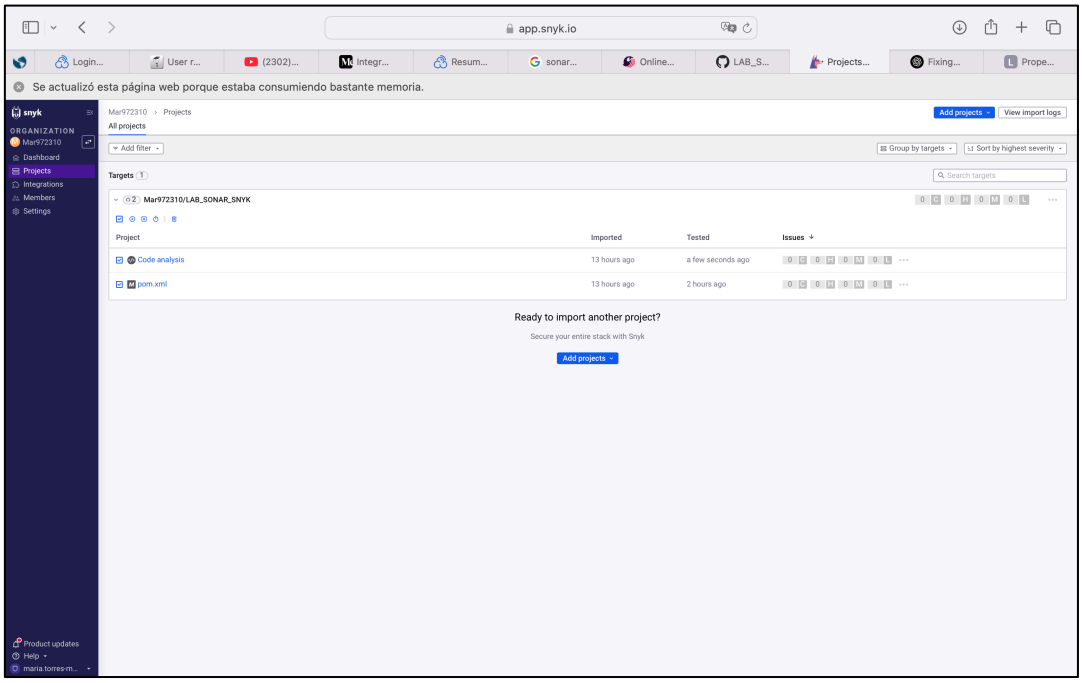
- spring-security-crypto:

```
<dependency>
        <groupId>org.springframework.security</groupId>
        <artifactId>spring-security-crypto</artifactId>
        <version>6.4.4</version>
</dependency>
```

- spring-security-core:

```
<dependency>
        <groupId>org.springframework.security</groupId>
        <artifactId>spring-security-core</artifactId>
        <version>6.4.4</version>
</dependency>
```





Conclusions

This lab made it possible to significantly improve the security and maintainability of the real estate property management system code. By using SonarCloud and Snyk, issues related to nomenclature, redundancy, security, and good development practices were identified and fixed. These improvements strengthen project security and ensure a more efficient and maintainable code.