

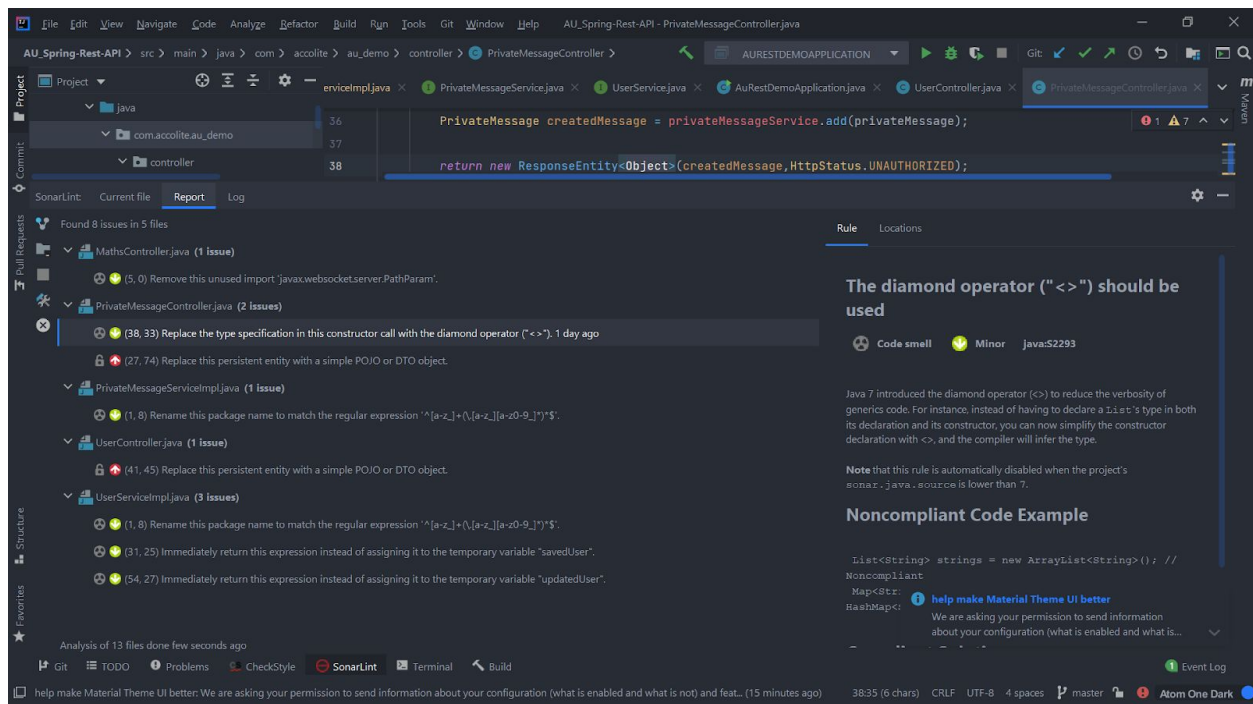
Java Code Quality Assignment

Prabhakar

Code Analyzers:

SonarLint:

- Install in IntelliJ File->settings->plugin->type sonarlint and install it.
- Once the installation is over, need to restart the IDE.
- Scanned project with 8 class files and encountered few issues



Package names should comply with a naming convention:

- Shared coding conventions allow teams to collaborate efficiently. This rule checks that all package names match a provided regular expression.

Code : `package com.accolite.au_demo.service.impl;`

Local variables should not be declared and then immediately returned or thrown:

- Declaring a variable only to immediately return or throw it is a bad practice.

Code : `User updatedUser = userRepository.save(user);`

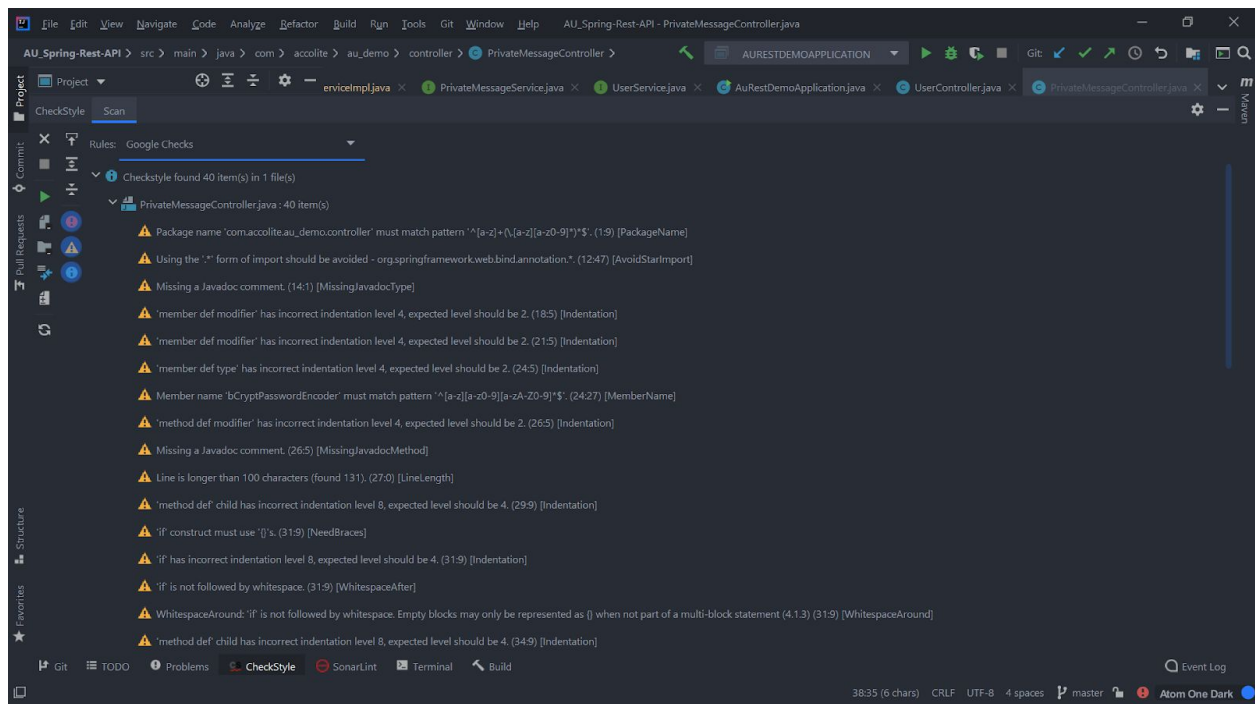
The diamond operator ("<>") should be used:

- Java 7 introduced the diamond operator (<>) to reduce the verbosity of generics code. For instance, instead of having to declare a List's type in both its declaration and its constructor, you can now simplify the constructor declaration with <>, and the compiler will infer the type.

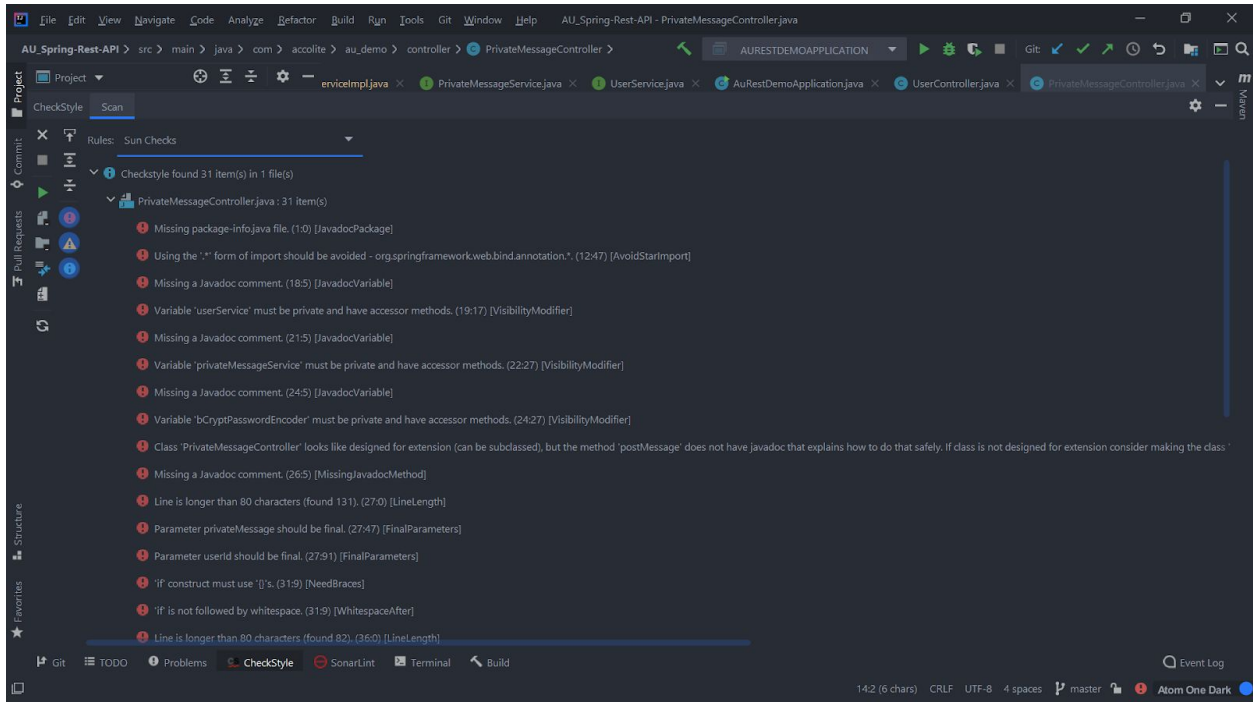
Code: `return new ResponseEntity<Object>(createdMessage, HttpStatus.UNAUTHORIZED);`

CheckStyle:

- Install in IntelliJ File->settings->plugin->type sonarlint and install it.
- Once the installation is over, need to restart the IDE.
- Scanned google check a class file and got 40 warnings



- Scanned with sun check and got 31 errors

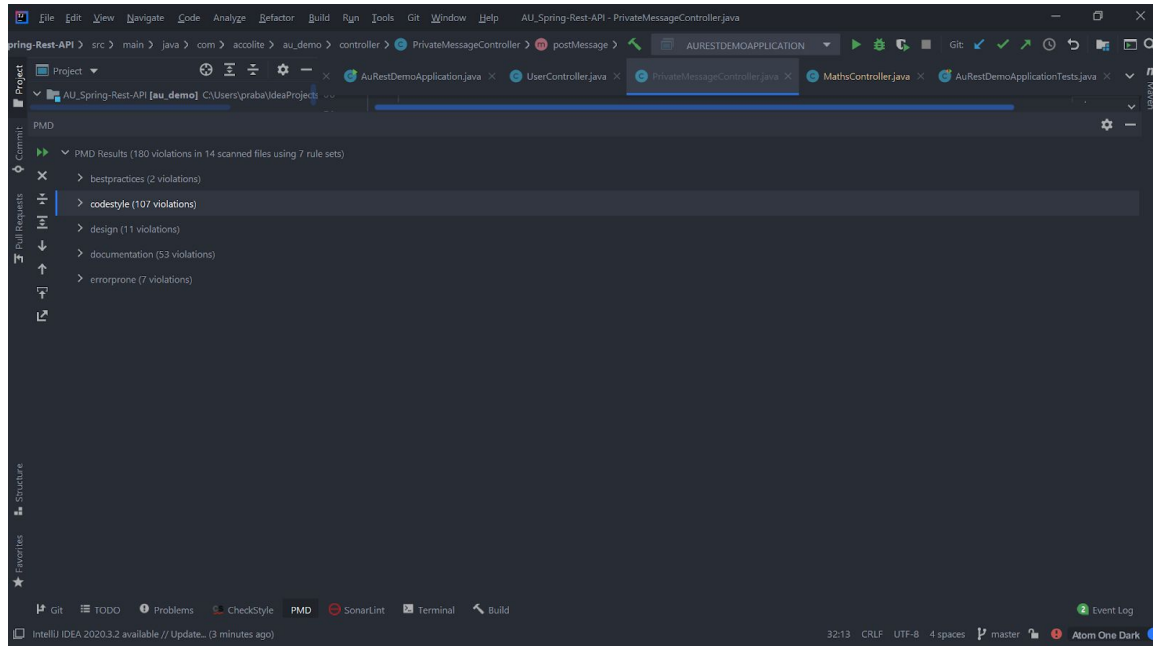


PMD:

- A plugin to run static analysis using PMD in IntelliJ.
- Install in IntelliJ File->settings->plugin->type sonarlint and install it.
- Once the installation is over, need to restart the IDE.
- Scanned the project with 14 files and got 180 violations.

Violations:

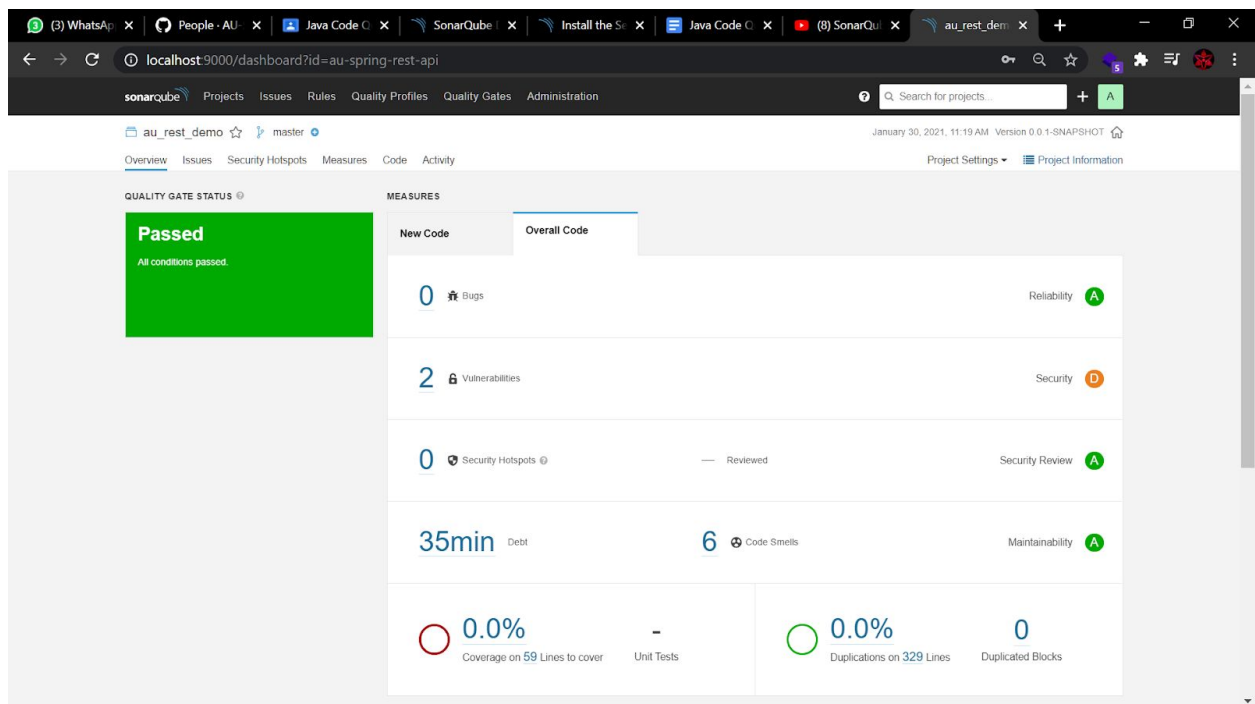
- Best practices
- Code style
- Design
- Documentation
- error prone



SonarQube:

- sonarQube is installed and configured in /conf/wrapper.conf
- Scanned the project with 7 java class files.

SonarQube home:



Vulnerabilities:

The screenshot shows the SonarQube web interface for the project 'au_rest_demo'. The left sidebar contains filters for 'Type' (VULNERABILITY, 2), 'Severity' (Critical, 2), and 'Scope'. The main area displays two vulnerability issues:

- Issue 1:** 'Replace this persistent entity with a simple POJO or DTO object. Why is this an issue?' (Vulnerability, Critical, 6 days ago, L27, 10min effort). Location: `src/_/accolite/au_demo/controller/PrivateMessageController.java`.
- Issue 2:** 'Replace this persistent entity with a simple POJO or DTO object. Why is this an issue?' (Vulnerability, Critical, 6 days ago, L41, 10min effort). Location: `src/_/accolite/au_demo/controller/UserController.java`.

Below the issues, a yellow banner states: 'Embedded database should be used for evaluation purposes only. The embedded database will not scale, it will not support upgrading to newer versions of SonarQube, and there is no support for migrating your data out of it into a different database engine.'

Replace this persistent entity with a simple POJO or DTO object:

The screenshot shows the SonarQube web interface with the code context for the vulnerability issue. The left sidebar shows the issue details. The main area displays the Java code for `UserController.java` with the vulnerability highlighted:

```
19 UserService userService;
20
21 @Autowired
22 PrivateMessageService privateMessageService;
23
24 BCryptPasswordEncoder bCryptPasswordEncoder = new BCryptPasswordEncoder();
25
26 @PostMapping("/add/{userId}")
27 public ResponseEntity<Object> postMessage(@RequestBody PrivateMessage privateMessage, @PathVariable("userId") Integer userId) {
28
29     User user = userService.getUserById(userId);
30
31     if(user == null)
32         return ResponseEntity.notFound().build();
33
34     privateMessage.setUserId(user.getUserId());
35
36     PrivateMessage createdMessage = privateMessageService.add(privateMessage);
37
38     return new ResponseEntity<Object>(createdMessage, HttpStatus.UNAUTHORIZED);
39 }
40
41 @GetMapping("/get/{userId}")
42 public ResponseEntity<Object> getMessage(@PathVariable("userId") Integer userId, @RequestHeader("password") String password){
43
44     User user = userService.getUserById(userId);
45
46     if(user == null)
```


Code Smell:

The screenshot shows the SonarQube web interface for the 'au_rest_demo' project. The left sidebar contains filters for 'Type' (CODE SMELL), 'Severity' (Blocker, Critical, Major, Minor, Info), 'Scope', 'Resolution', 'Status', and 'Security Category'. The main panel displays a list of 6 code smells, each with a description, severity, and effort. The first smell is 'Replace the type specification in this constructor call with the diamond operator ("<>")' with a severity of 'Minor' and an effort of '1min'.

Issue	Severity	Effort
Replace the type specification in this constructor call with the diamond operator ("<>"). Why is this an issue?	Minor	1min
Rename this package name to match the regular expression "[a-z]*([a-z0-9])\$". Why is this an issue?	Minor	10min
Rename this package name to match the regular expression "[a-z]*([a-z0-9])\$". Why is this an issue?	Minor	10min
Immediately return this expression instead of assigning it to the temporary variable "savedUser". Why is this an issue?	Minor	2min
Immediately return this expression instead of assigning it to the temporary variable "updatedUser". Why is this an issue?	Minor	2min
Add at least one assertion to this test case. Why is this an issue?	Blocker	10min

Replace the type specification in this constructor call with the diamond operator ("<>"):

The screenshot shows the SonarQube web interface for the 'au_rest_demo' project, displaying the code for the 'PrivateMessageController' class. The code is shown in a tabbed view, and the 'Issues' tab is active. The code is as follows:

```
31 if(user == null)
32     return ResponseEntity.notFound().build();
33
34 privateMessage.setUserId(user.getUserId());
35
36 PrivateMessage createdMessage = privateMessageService.add(privateMessage);
37
38 return new ResponseEntity<Object>(createdMessage, HttpStatus.UNAUTHORIZED);
39
40 }
41
42 @GetMapping("/{get}/{userId}")
43 public ResponseEntity<Object> getMessage(@PathVariable("userId") Integer userId, @RequestHeader("password") String password){
44
45     User user = userService.getUserById(userId);
46
47     if(user == null)
48         return ResponseEntity.notFound().build();
49
50     if(bCryptPasswordEncoder.matches(password, user.getPassword())) {
51         return ResponseEntity.ok(privateMessageService.getMessages(userId));
52     }
53     return ResponseEntity.ok("Invalid login");
54 }
55
56 @DeleteMapping("/{delete}/{messageId}")
57 public String deleteMessage(@PathVariable("messageId") Integer messageId){
58     return privateMessageService.deleteMessage(messageId);
59 }
```

The code smell 'Replace the type specification in this constructor call with the diamond operator ("<>")' is highlighted in the code, pointing to the line: `return new ResponseEntity<Object>(createdMessage, HttpStatus.UNAUTHORIZED);`

CWE - Common Weakness Enumeration:

- CWE™ is a community-developed list of software and hardware weakness types.
- It serves as a common language, a measuring stick for security tools, and as a baseline for weakness identification, mitigation, and prevention efforts.

Benefits:

- Reducing time and effort to triage, validate, and prioritize reports;
- Improving the alignment of expectations for report resolution and bounty payouts;
- Improving interoperability with external systems;
- Improving interoperability with custom internal taxonomies;
- Unlocking advanced analytics.

OWASP Top 10:

The OWASP Top 10 is a standard awareness document for developers and web application security. It represents a broad consensus about the most critical security risks to web applications.

Security Risks and Prevention

1. Injection

Prevention - Application security testing

2. Broken Authentication and Session Management

Prevention- Multi-factor authentication

3. Sensitive Data Exposure

Prevention - Encryption of data at rest and in transit

4. XML External Entity

Prevention - Static application security testing (SAST)

5. Broken Access Control

Prevention - Penetration testing

6. Security Misconfiguration

Prevention - Dynamic application security testing (DAST)

7. Cross-Site Scripting

Prevention - Developer training complements security testing

8. Insecure deserialization

Prevention - Application security tools can detect deserialization flaws

9. Using Components With Known Vulnerabilities

Prevention - Software composition analysis

10. Insufficient Logging and Monitoring

Prevention - Think like an attacker

CERT

- The CERT Division is a leader in cybersecurity. They partner with government, industry, law enforcement, and academia to improve the security and resilience of computer systems and networks.
- They study problems that have widespread cybersecurity implications and develop advanced methods and tools to counter large-scale, sophisticated cyber threats.

Functionalities of area CERT serve:

- Collection, analysis and dissemination of information on cyber incidents.
- Forecast and alerts of cyber security incidents
- Emergency measures for handling cyber security incidents
- Coordination of cyber incident response activities.
- Issue guidelines, advisories, vulnerability notes and whitepapers relating to information security practices, procedures, prevention, response and reporting of cyber incidents.
- Such other functions relating to cyber security as may be prescribed

SANS 25

The SANS Top 25 Most Dangerous Software Errors is a list of the most widespread and critical errors that can lead to serious vulnerabilities in software (please note: not all vulnerability types apply to all programming languages). The vulnerabilities include insecure interaction between components, risky resource management, and porous defenses.

Errors:

- Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
- Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
- Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')
- Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
- Missing Authentication for Critical Function

- Missing Authorization
- Use of Hard-coded Credentials
- Missing Encryption of Sensitive Data
- Unrestricted Upload of File with Dangerous Type
- Reliance on Untrusted Inputs in a Security Decision
- Execution with Unnecessary Privileges
- And so on.