SOEN390 - Software Engineering Team Design Project
Team 6 - Deliverable 1

Testing Plan and Report

Winter 2024

Done by:
Hoang Minh Khoi Pham 40162551
Michaël Gugliandolo 40213419
Jessey Thach 40210440
Mahanaim Rubin Yo 40178119
Vanessa DiPietrantonio 40189938
Ahmad Elmahallawy 40193418
Clara Gagnon 40208598
Khanh Huy Nguyen 40125396
Jean-Nicolas Sabatini-Ouellet 40207926
Mohamad Mounir Yassin 40198854

Professor Junqiu Yang
Department of Computer Science and Software Engineering
Gina Cody School of Engineering and Computer Science

Concordia University

Table of Content

This test approach document describes the appropriate strategies, process, workflows and methodologies used to plan, organize, execute and manage testing of Condo Management System.

1. TESTING TOOL

As a part of the developing cycle, testing plays a key role in maintainability and reliability for existing software, including the Condo Management System. In 2 parts, backend and frontend, tests will be conducted under types of unit tests, integration test and system test.

1.1 Platform

2 platforms that we chose to implement for this project: Github platform and Sonarqube

1.1.1 Github

We implemented 2 workflows as GitHub actions for CI in our repository, one for dev branch and one for main branch. The CI GitHub action is responsible for building and deploying the frontend and backend, and for monitoring the status of the deployment on the server, not to mention the compatibility of the node_modules implemented throughout the project.

1.1.2 Sonarqube

We implemented Sonarqube to validate the code quality and maintainability. Not only it's a powerful static code analysis tool that automatically reviews codes for bugs, vulnerabilities, code smells and quality issues, but also it can indicate the latent issues early on, resulting in faster resolution and reducing risks in development.

1.2 Testing Module

Jest is a testing framework for unit and integration tests with the benefits of being simple, speedy and built-in features (mocks and snapshots). In the frontend, we implemented Jest with a configuration for Typescript, on the other hand, the backend is implemented with JavaScript. Jest, with its functions, can provide a quick test report with insightful data, telling the % coverage of statements, branches, functions and lines.

2. APPROACH

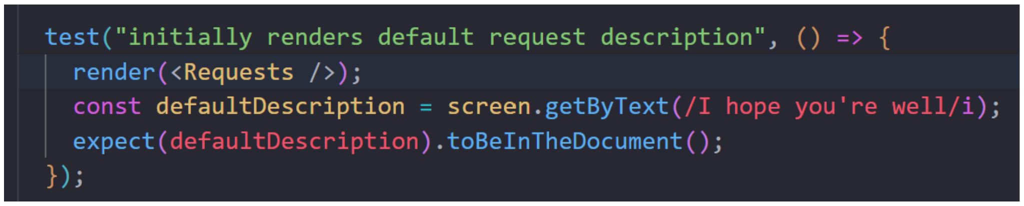
During the development process, the systematic approach is adopted to use Jest for testing. The process includes several important steps to ensure our tests' efficiency and the imminent resolution of issues may encounter:

2.1 Test Planning and Design

Supposedly, before any implementation, test cases and scenarios of each functional requirement, user story or acceptance criterion will be conveyed. All possible cases of functionality (positive, negative, edge, error) will have to be covered by test cases.

2.2 Unit Test

Unit testing evaluates individual components or "units" of software independently. It is a part of white box testing, an approach that examines the internal modules of the code. These units are the smallest testable pieces in the software. Unit testing is an essential component of software development since it ensures that each unit is executed correctly and detects any flaws early on. Following the TDD approach, each component, function or module will be put as top priority to write unit tests. After each implementation or code change, we execute the unit test to ensure the maintainability and quality of the updated codes. Notwithstanding, it results in high chances of catching regression earlier in the process and ensuring the new code still meets the expected behavior. A sample of a JEST unit testing on the Frontend side is shown below.



```
test("initially renders default request description", () => {
  render(<Requests />);
  const defaultDescription = screen.getByText(/I hope you're well/i);
  expect(defaultDescription).toBeInTheDocument();
});
```

Figure 1: Unit Testing Sample with JEST

2.3 Integration Test

Integration testing aims to evaluate the collective performance of various system components by examining their interactions rather than scrutinizing the entire system as a single entity. The focus is on verifying if these components communicate effectively. For instance, when component A transmits message X to component B, integration testing assesses whether their interactions result in the expected behaviors and functionalities. For instance, the landing page for the user profile is composed of several components. Testing these components separately would be unit testing, but testing them together and how they interact is integration testing. Verification between frontend and backend components, API endpoints and data flow will be ensured by integration tests. We will be writing integration tests to ensure the quality of our code.

2.4 End-to-End (E2E)

Condo Management website's workflow from frontend UI and backend services will be tested by E2E with real user interaction, stimulated by Cypress UI. The reason we decided to go with Cypress is because it provides an intuitive UI which makes it easier to learn and use. Furthermore, Cypress provides a powerful debugging tool, developers can easily inspect the DOM element if a test assertion fails and find out what went wrong. Supposedly, E2E is less frequently run compared to unit and integration tests, only if there are some major deployments such as pushing from dev to main. Broken user flows and critical bugs are more likely to be seen in this test as it provides confidence in overall functionality.

3. METRICS

With SonarQube it is easy to keep track of our quality metrics. The metrics are separated in two distinct sections. The first section is overview metrics, which combine other metrics to give an overall perspective of the code quality. As such, we have selected reliability, maintainability and issue metrics. The second section is singular metrics, which are security, test coverage, code duplication, code size, code complexity and issues.

3.1 Reliability

SonarQube assesses the reliability of code by identifying potential bugs, code smells, and other issues that could lead to unexpected behavior or system failures. By analyzing reliability metrics, such as the number of bugs detected and their severity, SonarQube helps ensure that the codebase is robust and dependable. Furthermore, the use of tests ensures that the functionalities work consistently according to the requirements. Finally, as part of our core development structure, each function also goes through manual testing in our dev environment. This acts as a final safety net to ensure that the code does what it is supposed to do consistently.

3.2 Maintainability

SonarQube assesses code maintainability based on readability, complexity, and adherence to coding standards and best practices. Improving maintainability measures such as code duplication, cyclomatic complexity, and code coverage simplifies future maintenance, refactoring, and code review procedures.

3.3 Security

SonarQube contains security-focused rules and tests that help uncover vulnerabilities, security hotspots, and possible security hazards in code. In addition, SonarQube reduces security risks and strengthens the application's overall security posture by assessing security metrics such as the number and severity of security vulnerabilities.

3.4 Coverage

SonarQube assesses code coverage by calculating the proportion of code covered by automated tests, such as unit tests, integration tests, and end-to-end tests.

boosting code coverage measures assures that the codebase has been properly tested, lowering the possibility of undetected bugs and boosting overall product quality.

3.5 Duplication

Duplication is a metric that needs to be monitored correctly since multiple people work on the same project. For example, two developers work on two different features that have similarities. If there are no checks in place, the code can get bloated, and hard to read and maintain. Thus, SonarQube is used to help us reduce duplication by finding repeating code patterns or similar code parts throughout the source. Addressing code duplication improves maintainability, readability, and consistency, resulting in more efficient development and simpler issue fixes.

3.6 Size

SonarQube gives information on the size of the codebase, including the number of files, lines of code, and code churn over time. Monitoring code size metrics allows us to track project growth, identify areas of code bloat or excessive complexity, and improve resource allocation and project planning.

3.7 Complexity

SonarQube evaluates code complexity by measures such as cyclomatic complexity, nesting depth, and method length. Analyzing complexity metrics aids in identifying unnecessarily complicated code segments, potential failure spots, and places for rewriting or optimization to enhance code quality and maintenance.

3.8 Issues

SonarQube keeps track of issues found in the code, including but not limited to: code smells, potential bugs, bad practices, unnecessary hard coding etc…
This process is called linting and is a good tool to ensure consistent good practices across developers. As a consequence, issue tracking can increase the maintainability, readability, complexity and other software qualities. The number of issues should be kept as low as possible

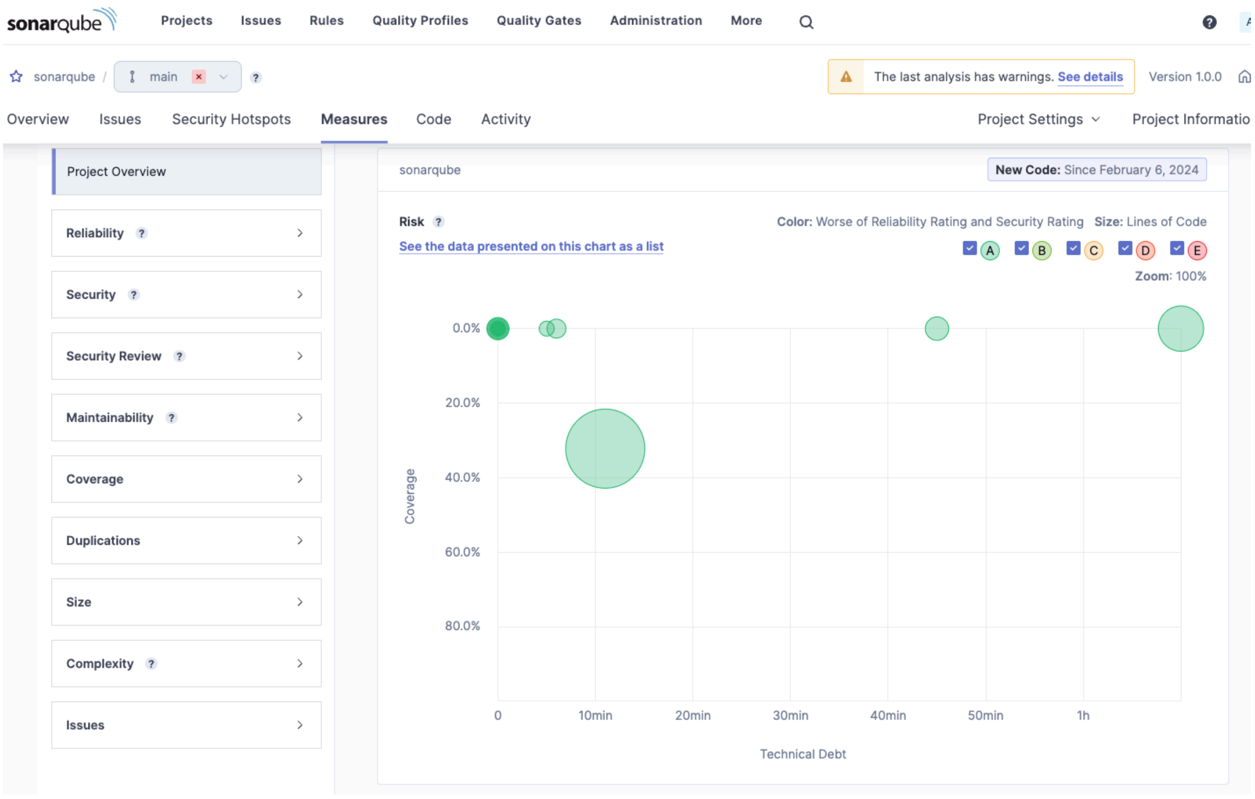


Figure 2: Measures of our code inside Sonarqube localhost server

4. COVERAGE

Software testing involves various metrics to assess the effectiveness of test cases in exercising the code. Four commonly used metrics are statement coverage, branch coverage, function coverage, and line coverage.

Statement coverage

measures the percentage of executable statements that have been executed at least once by the test cases. This metric provides insight into how well individual statements in the code have been exercised during testing.

Branch coverage

focuses on decision points in the code, such as if statements or loops. It assesses the percentage of decision branches that have been executed at least once, giving a more granular view of how well the code paths have been tested.

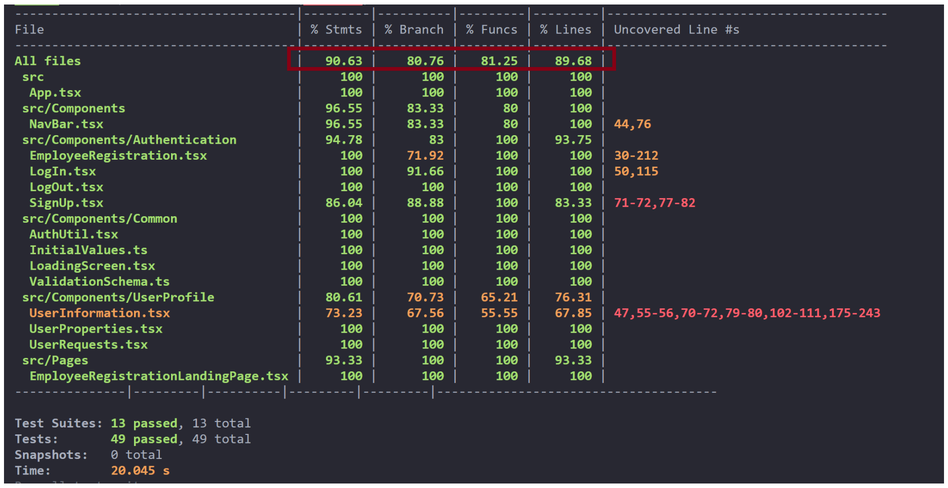
Function coverage

evaluates the percentage of functions or methods that have been executed by the test cases. This metric helps ensure that the different functional components of the software are adequately tested.

Line coverage

measures the percentage of lines of code that have been executed at least once. It provides a comprehensive overview of how well the entire codebase has been tested in terms of individual lines of code.

As asked in the requirements. We were able to pass 80% code coverage. On the frontend side, the statement coverage overall was 90.63%, the branch coverage was 80.76%, function coverage was 81.25% and lines were at 89.68% (as shown in the figure below. Therefore we succeeded at achieving over 80% for everything. We tested 13 different components and we had 49 different scenarios which all passed without any failure



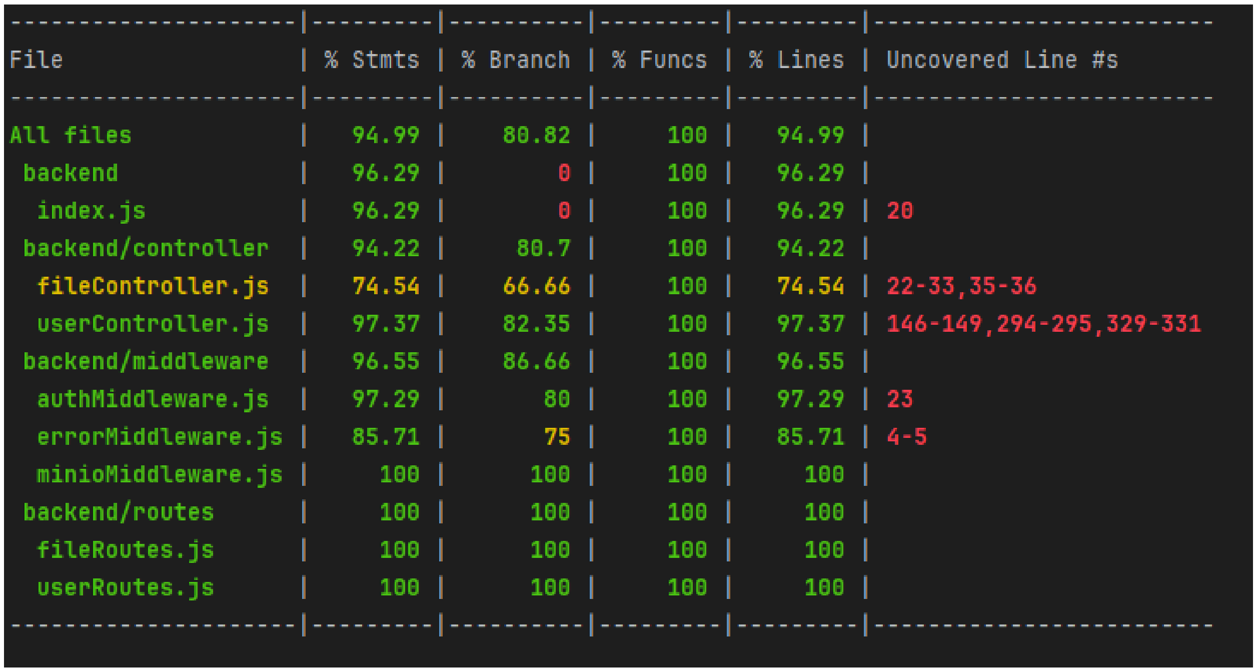
```
--------------------------------------|---------|----------|---------|---------|-------------------------------------
File                                  | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
--------------------------------------|---------|----------|---------|---------|-------------------------------------
All files                             |   90.63 |    80.76 |   81.25 |   89.68 |
 src                                  |     100 |      100 |     100 |     100 |
  App.tsx                             |     100 |      100 |     100 |     100 |
 src/Components                       |   96.55 |    83.33 |      80 |     100 |
  NavBar.tsx                          |   96.55 |    83.33 |      80 |     100 | 44,76
 src/Components/Authentication        |   94.78 |       83 |     100 |   93.75 |
  EmployeeRegistration.tsx            |     100 |    71.92 |     100 |     100 | 30-212
  LogIn.tsx                           |     100 |    91.66 |     100 |     100 | 50,115
  LogOut.tsx                          |     100 |      100 |     100 |     100 |
  SignUp.tsx                          |   86.04 |    88.88 |     100 |   83.33 | 71-72,77-82
 src/Components/Common                |     100 |      100 |     100 |     100 |
  AuthUtil.tsx                        |     100 |      100 |     100 |     100 |
  InitialValues.ts                    |     100 |      100 |     100 |     100 |
  LoadingScreen.tsx                   |     100 |      100 |     100 |     100 |
  ValidationSchema.ts                 |     100 |      100 |     100 |     100 |
 src/Components/UserProfile           |   80.61 |    70.73 |   65.21 |   76.31 |
  UserInformation.tsx                 |   73.23 |    67.56 |   55.55 |   67.85 | 47,55-56,70-72,79-80,102-111,175-243
  UserProperties.tsx                  |     100 |      100 |     100 |     100 |
  UserRequests.tsx                    |     100 |      100 |     100 |     100 |
 src/Pages                            |   93.33 |      100 |     100 |   93.33 |
  EmployeeRegistrationLandingPage.tsx |     100 |      100 |     100 |     100 |
--------------|---------|----------|---------|---------|-------------------------------------

Test Suites: 13 passed, 13 total
Tests:       49 passed, 49 total
Snapshots:   0 total
Time:        20.045 s
```

Figure 3: Sample Frontend Percentage Coverage on the Terminal

One of JEST's advantages is that it can provide a report in .html format which creates a nice view for the user to see the coverage. A few snapshots of the report were taken below. This provides more details and insights to the person interested.
For the future, we plan to maintain the overall code coverage of at least 80%. This will reduce the number of undetected bugs and will give us more confidence in our application. Code coverage enforcement can be done by configuring JEST to fail every time coverage drops below 80% which will prevent the PR from being merged.

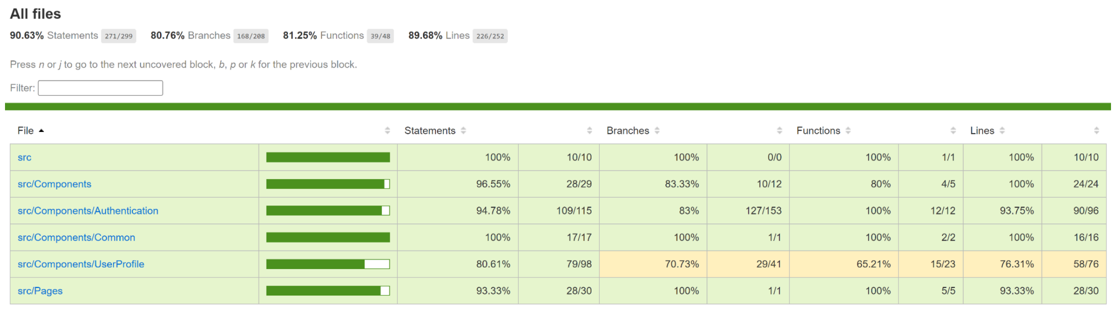
Our goal for the next sprint is to keep improving the different types of coverage to cover more test and edge cases. We are aiming to be at around 90% and ensure that all of them lie within the green color.

On the backend side, we were able to have 94.99% statement coverage, 80.82% branch coverage,94.99% line coverage, with 100% coverage on the functions percentage. For the next sprint, on top of improving our file controller testing, we will try to maintain coverage above 80% as we add more features to our application.



```
---------------------|---------|----------|---------|---------|-------------------------
File                 | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
---------------------|---------|----------|---------|---------|-------------------------
All files            |   94.99 |    80.82 |     100 |   94.99 |
 backend             |   96.29 |        0 |     100 |   96.29 |
  index.js           |   96.29 |        0 |     100 |   96.29 | 20
 backend/controller  |   94.22 |     80.7 |     100 |   94.22 |
  fileController.js  |   74.54 |    66.66 |     100 |   74.54 | 22-33,35-36
  userController.js  |   97.37 |    82.35 |     100 |   97.37 | 146-149,294-295,329-331
 backend/middleware  |   96.55 |    86.66 |     100 |   96.55 |
  authMiddleware.js  |   97.29 |       80 |     100 |   97.29 | 23
  errorMiddleware.js |   85.71 |       75 |     100 |   85.71 | 4-5
  minioMiddleware.js |     100 |      100 |     100 |     100 |
 backend/routes      |     100 |      100 |     100 |     100 |
  fileRoutes.js      |     100 |      100 |     100 |     100 |
  userRoutes.js      |     100 |      100 |     100 |     100 |
---------------------|---------|----------|---------|---------|-------------------------
```

Figure 4: Sample backend Percentage Coverage on the Terminal



All files

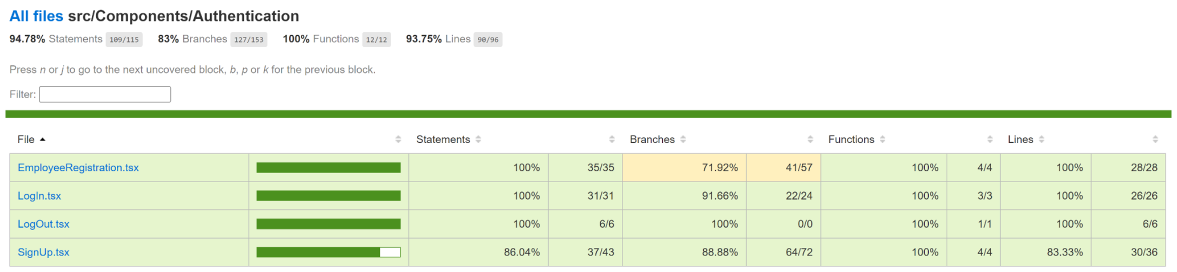
90.63% Statements 271/299 **80.76%** Branches 168/208 **81.25%** Functions 39/48 **89.68%** Lines 226/252

Press *n* or *j* to go to the next uncovered block, *b*, *p* or *k* for the previous block.

Filter:

File ▲		Statements		Branches		Functions		Lines	
src		100%	10/10	100%	0/0	100%	1/1	100%	10/10
src/Components		96.55%	28/29	83.33%	10/12	80%	4/5	100%	24/24
src/Components/Authentication		94.78%	109/115	83%	127/153	100%	12/12	93.75%	90/96
src/Components/Common		100%	17/17	100%	1/1	100%	2/2	100%	16/16
src/Components/UserProfile		80.61%	79/98	70.73%	29/41	65.21%	15/23	76.31%	58/76
src/Pages		93.33%	28/30	100%	1/1	100%	5/5	93.33%	28/30

Figure 5: Main Page Report with JEST



All files **src/Components/Authentication**

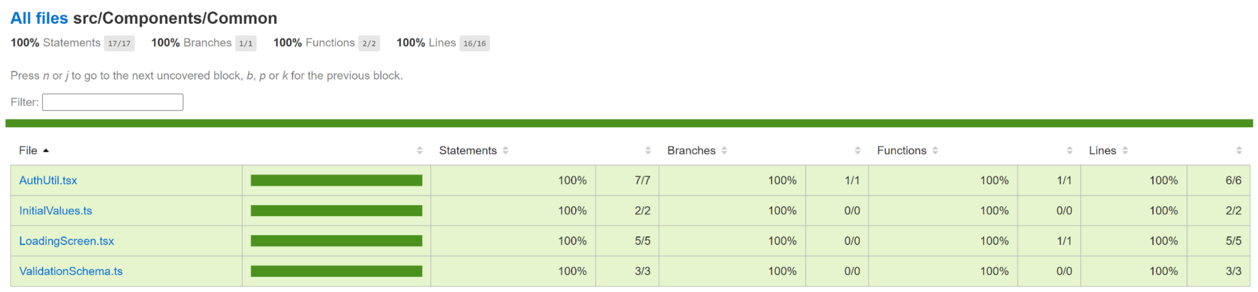
94.78% Statements 109/115 **83%** Branches 127/153 **100%** Functions 12/12 **93.75%** Lines 90/96

Press *n* or *j* to go to the next uncovered block, *b*, *p* or *k* for the previous block.

Filter:

File ▲		Statements		Branches		Functions		Lines	
EmployeeRegistration.tsx		100%	35/35	71.92%	41/57	100%	4/4	100%	28/28
LogIn.tsx		100%	31/31	91.66%	22/24	100%	3/3	100%	26/26
LogOut.tsx		100%	6/6	100%	0/0	100%	1/1	100%	6/6
SignUp.tsx		86.04%	37/43	88.88%	64/72	100%	4/4	83.33%	30/36

Figure 6: Detailed Statistics of the Authentication Folder and its Components



All files src/Components/Common

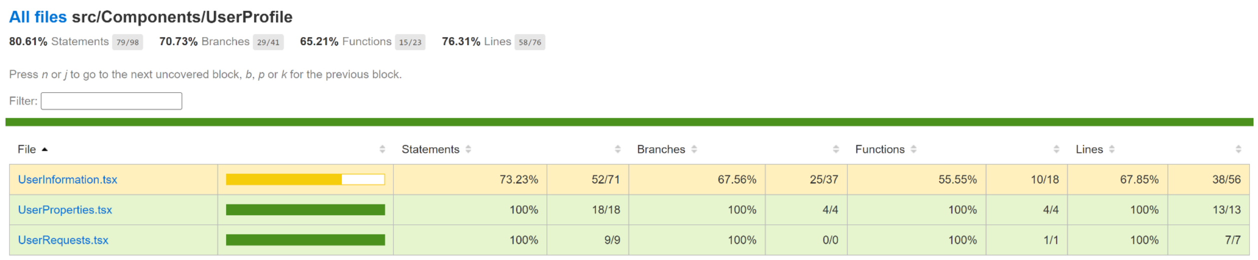
100% Statements 17/17 **100%** Branches 1/1 **100%** Functions 2/2 **100%** Lines 16/16

Press *n* or *j* to go to the next uncovered block, *b*, *p* or *k* for the previous block.

Filter:

File ▲		Statements		Branches		Functions		Lines	
AuthUtil.tsx	🟩	100%	7/7	100%	1/1	100%	1/1	100%	6/6
InitialValues.ts	🟩	100%	2/2	100%	0/0	100%	0/0	100%	2/2
LoadingScreen.tsx	🟩	100%	5/5	100%	0/0	100%	1/1	100%	5/5
ValidationSchema.ts	🟩	100%	3/3	100%	0/0	100%	0/0	100%	3/3

Figure 7: Detailed Statistics of the Common Folder with Reusable files



All files src/Components/UserProfile

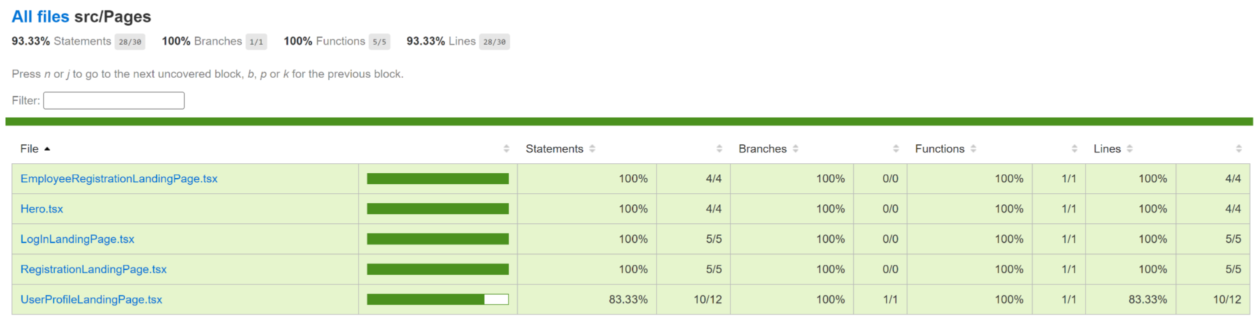
80.61% Statements 79/98 **70.73%** Branches 29/41 **65.21%** Functions 15/23 **76.31%** Lines 58/76

Press *n* or *j* to go to the next uncovered block, *b*, *p* or *k* for the previous block.

Filter:

File ▲		Statements		Branches		Functions		Lines	
UserInformation.tsx	🟨	73.23%	52/71	67.56%	25/37	55.55%	10/18	67.85%	38/56
UserProperties.tsx	🟩	100%	18/18	100%	4/4	100%	4/4	100%	13/13
UserRequests.tsx	🟩	100%	9/9	100%	0/0	100%	1/1	100%	7/7

Figure 8: Detailed Statistics of the Common Folder with Reusable files



All files src/Pages

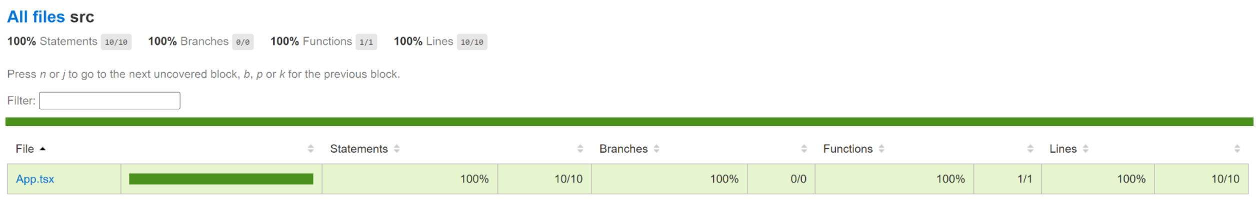
93.33% Statements 28/30 **100%** Branches 1/1 **100%** Functions 5/5 **93.33%** Lines 28/30

Press *n* or *j* to go to the next uncovered block, *b*, *p* or *k* for the previous block.

Filter:

File ▲		Statements		Branches		Functions		Lines	
EmployeeRegistrationLandingPage.tsx	🟩	100%	4/4	100%	0/0	100%	1/1	100%	4/4
Hero.tsx	🟩	100%	4/4	100%	0/0	100%	1/1	100%	4/4
LogInLandingPage.tsx	🟩	100%	5/5	100%	0/0	100%	1/1	100%	5/5
RegistrationLandingPage.tsx	🟩	100%	5/5	100%	0/0	100%	1/1	100%	5/5
UserProfileLandingPage.tsx	🟩	83.33%	10/12	100%	1/1	100%	1/1	83.33%	10/12

Figure 9: Detailed Statistics of the Pages Folder with Landing Pages



All files src

100% Statements 10/10 **100%** Branches 0/0 **100%** Functions 1/1 **100%** Lines 10/10

Press *n* or *j* to go to the next uncovered block, *b*, *p* or *k* for the previous block.

Filter:

File ▲		Statements		Branches		Functions		Lines	
App.tsx	🟩	100%	10/10	100%	0/0	100%	1/1	100%	10/10

Figure 10: Detailed Statistics of the Src folder (we tested the app itself)

5. ACCEPTANCE TEST

Sprint 1 Acceptance Tests:

Test ID	AT-1
Requirement	As a public user, I want to be able to create my profile with a profile picture, user name, contact email and phone number.
Acceptance Criteria	The system provides a user interface for public users to create their profiles.
Test Case	Expected Results
TC-1 Users sign up with unique username, email, phone number	The system will save the user record and direct users to the home page
TC-2 Users sign up with an existing username, email or phone number	The system will return a pop-up to indicate that users can't create a new account since the data are already in use
TC-3 Users sign up as an admin for a new company with a unique username, email, phone number	The system will save the user record and company record, establish the relationship and direct users to the home page
TC-3 Users sign up as an admin for a new company with existing username, email, phone number	The system will return a pop-up to indicate that users can't create a new account since the data are already in use

Test ID	AT-2
Requirement Tested	As a public user, I want to be able to log in and log out of my account.
Acceptance Criteria	The user has an existing account. The system provides a user interface for public users to log in and log out.
Test Case:	Expected Results
TC-1 User logs in with correct credentials	The system successfully authenticates the user's credentials and redirects the user to the designated landing page.

TC-2 User logs in with incorrect credentials	The system displays an error message indicating that the login credentials are incorrect.
TC-3 User logs out from account	The system securely logs the user out of their account and redirects the user to a designated logout confirmation page (or the login page).

Test ID	AT-3
Requirement Tested	As a public user, I want to be able to view and modify my account.
Acceptance Criteria	The user has an existing account. The system provides a user interface to display the user’s profile.
Test Case:	Expected Results
TC-1 User edits profile and clicks “Save Changes”	The system successfully saves and displays the changes made by the user to their profile.
TC-2 User edits profile and clicks “Cancel”	The system discards the changes made by the user without saving and displays the existing profile information.

Sprint 2 Acceptance Tests:

Test ID	AT-4
Requirement Tested	As a public user, I want to be able to change my password.
Acceptance Criteria	The user has an existing account.
Test Case:	Expected Results
TC-1 User enters a new password	The system updates the user’s password with the new one provided and invalidates the old password.
TC-2 User enters the same password	The system displays an error message indicating that the new password cannot be the same as the current password.

TC-3 User enters a password less than 8 characters long	The system displays an error message indicating the password requirement for minimum length was not met.

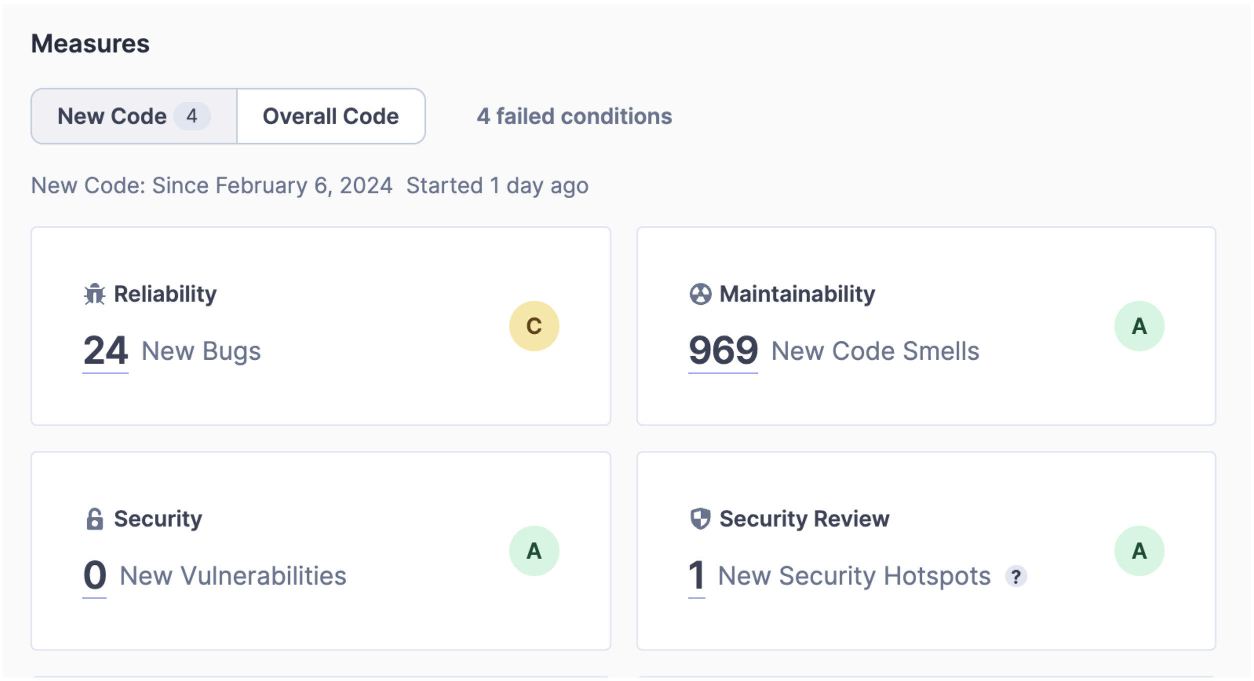
Test ID	AT-5
Requirement Tested	The system should be able to generate and send registration keys to unit owners and tenants.
Acceptance Criteria	Registration keys are unique.
Test Case:	Expected Results
TC-1 Generate Registration Key for Unit Owner	A unique registration key is generated for a unit owner.
TC-2 Generate Registration Key for Tenant	A unique registration key is generated for a tenant.
TC-3 Send Key to Registered User	The user receives the registration key via a communication channel in the system.
TC-4 Link Profile to Unit using Registration Key	The user gains access to relevant information and features to edit their unit.

Test ID	AT-6
Requirement Tested	As a condo management company, I want to be able to enter detailed information for each condo unit.
Acceptance Criteria	Given the company admin is logged in, they should be able to enter detailed information for each condo unit of their properties.
Test Case:	Expected Results
TC-1 Enter detailed information for Condo Unit	The system saves the detailed information of the new condo unit and displays a “Confirmation” message.

Test ID	AT-7

Requirement Tested	The system should allow the company admin to upload condo files to a property.
Acceptance Criteria	The condo file is in .pdf format.
Test Case:	Expected Results
TC-1 The user drags and drops the files in the dropbox.	The system displays the condo files in the dropbox.
TC-2 The user clicks the upload button in the dropbox and selects the files.	The system displays the condo files in the dropbox.

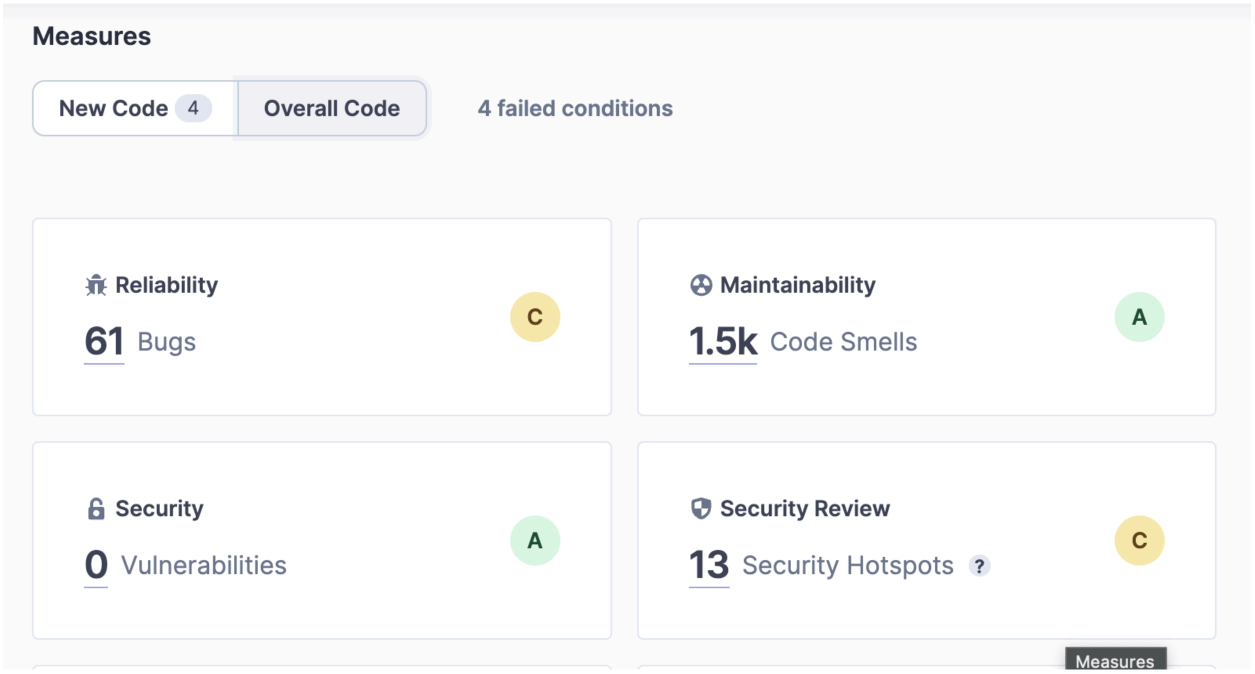
6. TEST RESULT



New Code: Since February 6, 2024 Started 1 day ago

Measure	Value	Rating
Reliability	24 New Bugs	C
Maintainability	969 New Code Smells	A
Security	0 New Vulnerabilities	A
Security Review	1 New Security Hotspots	A

Figure 10: SonarQube Analytics Result for new Code from yesterday



Measures

4 failed conditions

Reliability
61 Bugs
C

Maintainability
1.5k Code Smells
A

Security
0 Vulnerabilities
A

Security Review
13 Security Hotspots ?
C

Figure 11: SonarQube Analytics Result for overall Code