**CS673 Software Engineering** 

**Team 2 - CareerForge**

**Software Test Document**

| Team Member | Role(s) | Signature | Date |
| --- | --- | --- | --- |
| Stacey Burns | QA Leader | *SB* | 09/22/2025 |
| Qi Chen | Requirement Leader | *QC* | 09/22/2025 |
| Gopi Rayini | Configuration Leader | *GR* | 09/22/25 |
| James Rose | Team Leader | *JR* | 10/4/2025 |
| Yongxiang Chen | Security Leader | *YC* | 10/05/2025 |
| Pedro Ramirez | Design and implementation | *PR* | 10/06/2025 |
|  |  |  |  |
|  |  |  |  |

**Revision history**

| **Version** | **Author** | **Date** | **Change** |
| --- | --- | --- | --- |
| **1.0.0** | **Team** | **09/22/2025** | **Initial Draft** |
| **1.1.0** | **James Rose & Yongxiang Chen** | **10/5/2025** | **Frontend Info** |
|  | **Stacey Burns** | **10/06/2025** | **Updated number of backend tests and provided snapshot** |
| **1.2.0** | **Stacey Burns** | **10/07/2025** | **Added backend metrics of code coverage** |
| **1.3.0** | **Stacey Burns** | **10/12/2025** | **Added snapshots of testing flow diagrams based on feedback** |
| **1.4.0** | **Yongxiang Chen** | **10/12/2025** | **Added Frontend Tests** |

[Testing Summary](#_sm5odwyvuk3j)

[Manuel Tests Reports](#_pqso2mbjyzx4)

[Automated Testing Reports](#_mtfbusfb0eq3)

[Testing Metrics](#_rijyjeu2ojqa)

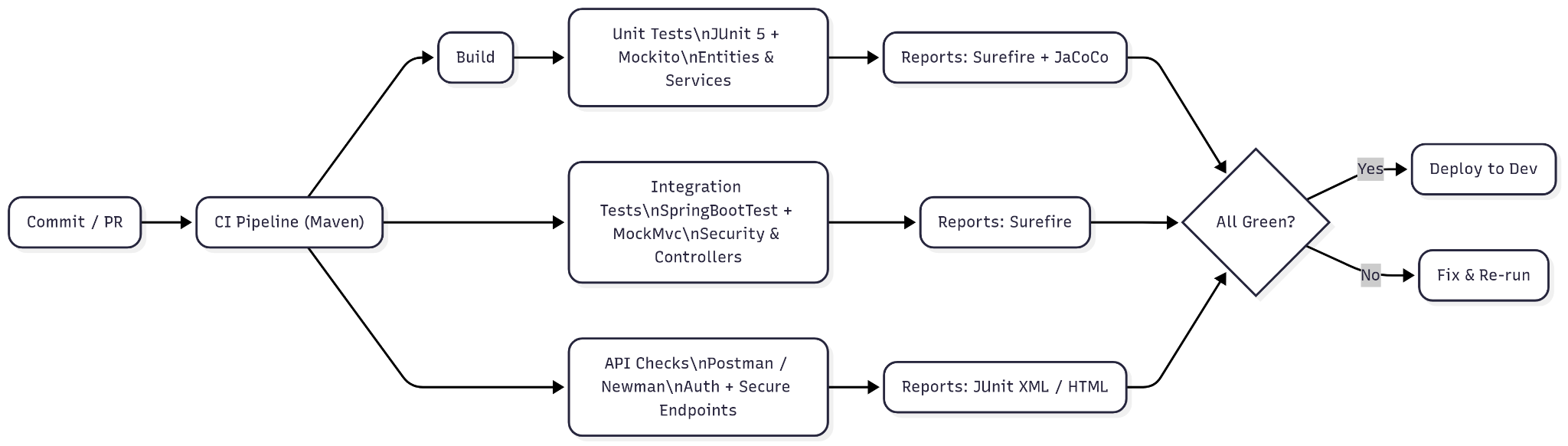
[References](#_15tmymhipvdv)

[Glossary](#_8n34lvocupub)

# Testing Summary

In this section, you will summarize what was tested, who is involved in testing, when to test, testing techniques used, and testing result. You may have the following tests

**Backend:**

****

* + Unit Testing - Our unit tests are automatically executed as part of every Maven build and deployment, ensuring continuous validation of the system. The tests cover all core entities and security components, including:
    - Entity Layer: Validation of User, Job, ApplicationTracking, Benefits, Values, and LocationCoordinate entities, focusing on field integrity, default/null handling, relationships, lifecycle methods (timestamps, updates), and equality/hashCode correctness.
    - Security Layer: Validation of JwtUtil for token generation, validation, and expiration handling, as well as integration tests for secured endpoints (/secure, /admin/dashboard) to confirm proper authentication and role-based authorization.
    - Roles Involved: Developers are responsible for writing, reviewing, and executing the unit tests as part of the CI/CD pipeline.

Our unit tests are executed automatically during each Maven build and before every deployment. Developers also run these tests locally before pushing changes.

The techniques used are JUnit5 for test execution and assertions, SpringBootTest with MockMvc for security and endpoint validation, Dynamic Property Injection to generate secrets and configuration values at runtime, ensuring isolation and preventing flaky tests, and test Doubles/Mocks where applicable to isolate functionality.

* + Specific unit tests:
    - Unit tests for ApplicationTracking - Verifies that the entity behaves correctly in terms of construction, field values, and business logic between the user, job, and application tracking.
    - Unit tests for Benefits - Verifies that the entity correctly handles creation, field updates, nulls/empties, equality, hashing, and string representation.
    - Unit tests for Jobs- Verifies Job entity. It checks that job postings are created, validated, and updated correctly, while also verifying related behaviors like deadlines, salaries, applications, and relationships.
    - Unit tests for LocationCoordinates - Verifies that latitude and longitude values are handled correctly, along with equality, hashing, and string output.
    - Unit tests for Users - Validates that a user can be created, updated, and used correctly both as an employee and as an employer, while also testing related business logic, timestamps, embedded objects, and Spring Security integration.
    - Unit tests for Values - Verifies that the Values entity can be created, updated, compared, and represented reliably.
    - Unit tests for JWT - Unit test for security utility. It verifies that JSON Web Tokens (JWTs) are generated, validated, and expired correctly within the application.
    - Unit tests for UserService - Unit test for UserServiceImpl which manages users in the application. It uses JUnit5 and Mockito to mock dependencies and validate business logic.
  + Integration testing

Our integration testing validates how different components of the system interact, with a particular focus on authentication, authorization, and API workflows. Integration tests are created and maintained by developers. These tests have been run manually during the development of these functionalities via Postman. The collection can be found [here](https://metcs673-team-2.postman.co/workspace/METCS673-Team-2-Workspace~2e290ce4-6167-408d-8f8b-60e4fc88d7dd/collection/15824451-823da234-e01b-481d-8e5d-5c2efd03ebce?action=share&creator=15824451) and can be run manually at any time. These tests provide confidence that authentication, registration, and secure endpoint access behave as expected across the system, preventing regressions and ensuring reliability before deployment.

* + - Security Integration Tests (JUnit and MockMvc):

We maintain an automated test suite for our Spring Security and JWT setup. These tests verify that user authentication and role-based authorization function correctly when interacting with real endpoints, ensuring secure access control between users and admins.

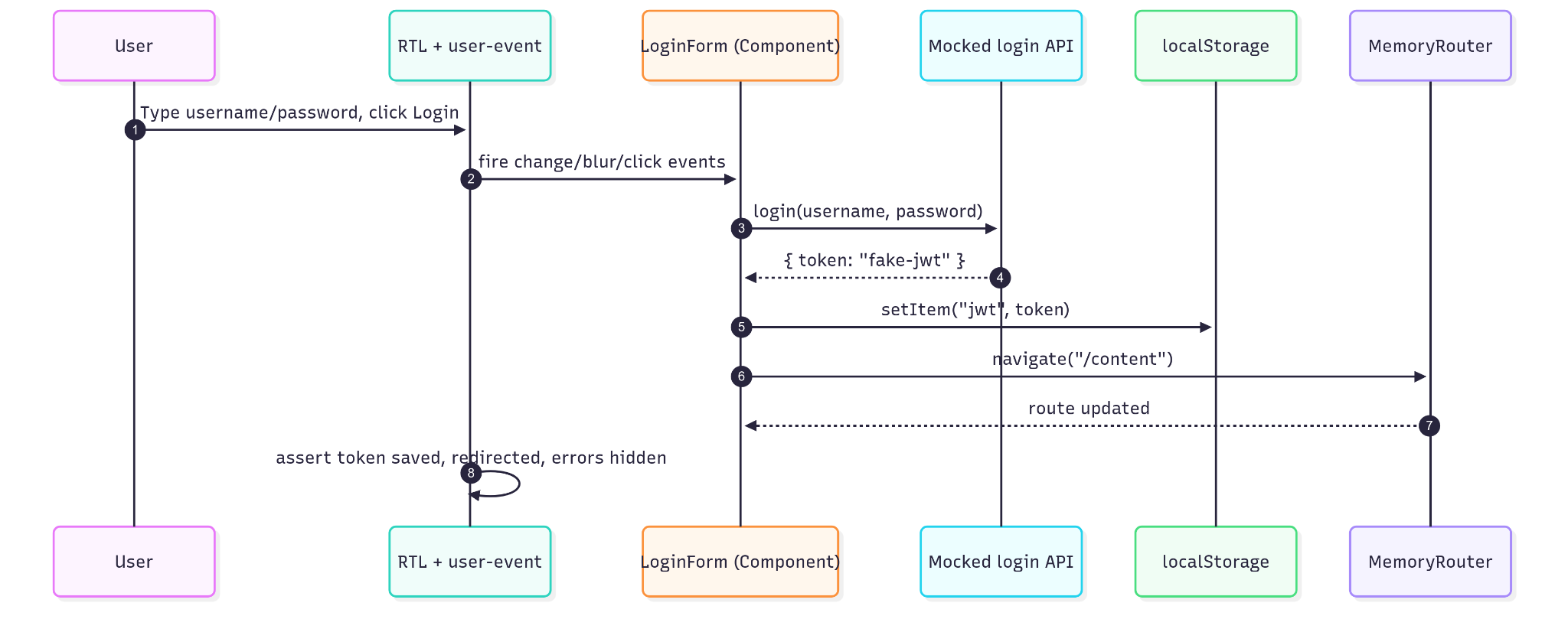
* + - Postman API Test Collection:

We supplement automated integration tests with a Postman collection that validates key API endpoints:

* + - * Login Tests: Confirm that user credentials return a 200 OK response on successful login.
      * Registration Tests: Verify that new users can be created with all required fields (username, email, password, firstName, lastName, userType, and active status).
      * JWT Tests: Confirm that the authentication API returns a valid JWT token upon login.
      * Secure Endpoint Tests: Validate that authorized endpoints accept valid tokens and reject invalid or missing tokens.
      * Job Tests: To be completed
  + Acceptance Testing - At present, we have implemented a basic login test. The primary purpose of this test was to establish the foundation for using Selenium to drive our acceptance, system, and regression testing efforts moving forward. These tests are built by our engineering team and will be reviewed collaboratively by peers to ensure coding standards and test coverage align. We are using Selenium WebDriver with Java for UI automation, focused on simulating real user actions and verifying UI-level outcomes.
  + Regression Testing - Our unit tests also function as a form of regression testing, since they are executed automatically with every code push to ensure that existing functionality continues to work as expected. This provides an early safeguard against unintended side effects from new changes. At present, we have a small subset of tests that require maintenance, and these are being addressed to restore full regression coverage.

**Frontend:**

Our React unit tests are executed with Jest and React Testing Library (RTL) as part of the frontend CI pipeline, and developers also run them locally before pushing changes. The tests validate component rendering, route-aware behavior, form validation, user interactions (typing, clicking, blur), and shared utilities. This provides early feedback and prevents regressions in UI workflows. Below is a snapshot of our frontend testing sequence diagram.



The techniques used are Jest for execution and assertions, React Testing Library and @testing-library/user-event for accessible queries and realistic user interactions, MemoryRouter to simulate routing without a browser, ThemeProvider harnessing where needed, and jest.mock to stub API calls and isolate side effects (e.g., localStorage writes) without real HTTP.

Specific unit tests:

* Unit tests for Aside – Verifies route-aware labeling (“My Jobs” vs “Home”) and navigation between /content and /myJobs.
* Unit tests for Field – Verifies dropdown open/close, live filtering by text input, selection behavior, and onChange propagation.
* Unit tests for MyJobsButton – Verifies rendering and expected styling class.
* Unit tests for LoginForm – Verifies required-field validation, prevention of invalid submit, successful submit saving JWT to localStorage, and redirect to /content.
* Unit tests for RegisterForm – Verifies required fields, email format validation, and onSubmit payload when valid.
* Unit tests for validation helpers – Verifies isRequired and isEmail.
* Unit tests for API Client– Test isolated functions like fetch requests, error handling, or response parsing.
* Unit tests for ApplyJobButton – Verifies initial rendering of the Apply button, successful apply flow, and failed apply flow. Mocks applyJob API, localStorageHelpers, and window.open to isolate behavior.
* Unit tests for SavedJobApplyButton – Verifies initial rendering and disabled state, successful apply flow, failed apply flow, and link-opening behavior when a job URL exists. Mocks applyJob, deleteSavedJob, and window.open for isolated testing.
* Unit tests for SaveJobButton – Verifies initial rendering for saved and unsaved states, disabled button when detailed=false, successful save flow, failed save flow, and unsave behavior when toggled again. Mocks saveJob, isJobSaved, and toggleSavedJobId to isolate UI and logic interactions.
* Unit tests for ThemeToggler – Verifies correct icon rendering based on initial theme and ensures theme toggles properly on click, updating both UI icons and localStorage. Wraps component in ThemeProvider to test context-based state updates.
* Unit tests for Type component – Verifies default rendering, dropdown open/close behavior, dynamic population from jobs:types event, and selection logic. Ensures choosing a type triggers onChange, dispatches a jobs:typeSelect event, and closes the panel; selecting “Any type” calls onChange. Also confirms “No types detected yet…” displays when no types exist.

*Integration testing:*  
Our UI integration tests validate component interaction with routing, context, and API layers. Components are mounted with MemoryRouter for route state, wrapped in ThemeProvider where required, and network is isolated with jest.mock (e.g., login API) to assert navigation and localStorage effects without real backend calls.

*Security Integration Tests (frontend):*  
Not a separate layer in the UI. Authentication/authorization behavior is exercised via LoginForm tests that mock API responses and assert token handling and redirects.

*Postman API Test Collection (frontend):*  
Not applicable to the UI test suite. Frontend relies on Jest/RTL for unit and integration tests; end-to-end flows will be covered by acceptance/system testing.

*System Testing:*  
System testing for the UI is planned but not yet fully implemented; it will drive full user workflows against a running backend.

*Acceptance Testing:*  
Acceptance testing is being bootstrapped. We will expand from a basic login scenario to broader flows using Selenium WebDriver to drive a real browser and verify UI-level outcomes, aligned with the backend approach.

*Regression Testing:*  
All frontend unit tests are executed on each commit/PR, serving as a regression net to detect unintended UI changes early.

# Manual Testing Report

In this section, you will give a detailed description of each manual test case performed and the result. If this is a previous You shall list what are existing tests developed in the previous semester and what are new tests developed currently.

Here is a sample template that can be used for each test case. For system tests or acceptance tests, you may also include some screenshots.

* Test case ID, name
* New or old:
* Test items: (what do you test )
* Test priority (high/medium/low)
* Dependencies (to other test case/requirement if any):
* Preconditions: (if any)
* input data:
* Test steps:
* Postconditions:
* Expected output:
* Actual output:
* Pass or Fail:
* Bug id/link: (this should link to your github issue id)
* Additional notes:

(You can use an additional spreadsheet for this section as well)

**Backend:**

BACK-1 – **Auth: Login happy path returns JWT**

New/Old: New | Priority: High

Endpoint: POST /auth/login

Preconditions: User pedrotest/pedro123 exists

Input: {"username":"pedrotest","password":"pedro123"}

Expected: 200 + JSON {token:<non-empty>}

BACK-2 – **Auth: Empty username/password → 400**

New | High

Endpoint: POST /auth/login

Input: {"username":"","password":""}

Expected: 400 with field errors for username/password

BACK-3 – **Register: invalid email → 400**

New | Medium

Endpoint: POST /auth/register

Input: {"name":"James","username":"james1","email":"hello","password":"secret123"}

Expected: 400 with “Enter a valid email”

BACK-4 – **AuthZ: protected route requires JWT**

New | High

Endpoint: e.g., GET /applications/me (or your protected endpoint)

Steps: Without token → 401; with Authorization: Bearer <jwt> → 200

Expected: Unauthorized vs OK with current user payload

**Frontend:**

*Below are representative manual UI tests following the same template:*

Test case ID, name: **FRONT-1 – Login: happy path saves JWT and redirects** New or old: New  
 Test items: LoginForm → API call, localStorage write, redirect  
 Test priority (high/medium/low): High  
 Dependencies: Auth API (mocked or dev backend)  
 Preconditions: Valid credentials exist  
 input data: username=pedrotest, password=pedro123  
 Test steps: Open /login → enter username/password → click **Login** Postconditions: JWT present in localStorage  
 Expected output: Redirect to /content and content UI visible  
 Actual output: Redirect to /content and content UI visible Pass or Fail: *Pass*

Test case ID, name: **FRONT-2 – Login: validation prevents empty submit** New or old: New  
 Test items: Required-field validation  
 Test priority: High  
 Dependencies: None  
 Preconditions: None  
 input data: empty username/password  
 Test steps: Blur both fields → click **Login** Postconditions: No navigation and no API call  
 Expected output: “Username is required”, “Password is required”  
 Actual output:“Username is required”, “Password is required” Pass or Fail: *Pass*

Test case ID, name: **FRONT-3 – Register: invalid email shows error** New or old: New  
 Test items: Email format validation  
 Test priority: Medium  
 Dependencies: None  
 Preconditions: None  
 input data: name=James, email=hello, password=secret123  
 Test steps: Fill fields → blur → click **Register** Postconditions: Submit blocked  
 Expected output: “Enter a valid email”  
 Actual output: “Enter a valid email” Pass or Fail: *Pass*

Test case ID, name: **FRONT-4 – Aside: toggles Home/My Jobs and navigates** New or old: New  
 Test items: Route-aware label; navigation target  
 Test priority: Medium  
 Dependencies: Router  
 Preconditions: Start once at /content and once at /myJobs  
 input data: Click the red button  
 Test steps: From /content expect “My Jobs” → click → /myJobs; from /myJobs expect “Home” → click → /content  
 Postconditions: URL updates accordingly  
 Expected output: Navigation to expected routes  
 Actual output: Navigation to expected routes Pass or Fail: *Pass* Additional notes: —

Test case ID, name: **FRONT-5 – Field filter: opens, filters, selects** New or old: New  
 Test items: Dropdown open/close, filter, selection callback  
 Test priority: Medium  
 Dependencies: None  
 Preconditions: None  
 input data: Filter text=Eng, select “Engineering”  
 Test steps: Open dropdown → type → select  
 Postconditions: Button label updates; onChange('Engineering') fired  
 Expected output: Selection reflected in UI and callback  
 Actual output: Selection reflected in UI and callback Pass or Fail: *Pass*

Test case ID, name: **FRONT-6 – ApplyJobButton: successful and failed apply flow**

New or old: New  
Test items: ApplyJobButton → UI rendering, API call, success/error message, localStorage mock  
Test priority: High  
Dependencies: applyJob API (mocked)  
Preconditions: Job object available  
Input data: job = { title: "Frontend Developer", url: "<https://example.com/apply>" }  
Test steps:Render component ->Click “Apply” button->Wait for API response (success and failure cases)  
Postconditions: Button text and message update based on response  
Expected output:

* Success: “Applied” + success message
* Failure: error message shown (“Network fail”)

Actual output: Matches expected behavior  
Pass or Fail: Pass

### Test case ID, name: FRONT-7 – SavedJobApplyButton: apply and delete flow

New or old: New  
Test items: SavedJobApplyButton → Apply flow, deleteSavedJob API, URL open, error handling  
Test priority: High  
Dependencies: applyJob, deleteSavedJob APIs (mocked)  
Preconditions: Saved job object exists  
Input data: savedJob = { id: 1, title: "Frontend Developer" }  
Test steps: Render component with detailed=true->Click “Apply Now” button->Observe API calls and UI updates  
Postconditions: Job deleted from saved list, success message shown  
Expected output:

* Success: Both APIs called, success message displayed
* Failure: “Try Again” + error message shown

Actual output: Matches expected  
Pass or Fail: Pass

### Test case ID, name:FRONT-8 – SaveJobButton: save/unsave behavior

### New or old: New Test items: SaveJobButton → localStorage logic, saveJob API, toggle behavior Test priority: Medium Dependencies: saveJob API, localStorage helpers (mocked) Preconditions: Job not saved initially Input data: job = { \_id: "abc123", title: "Frontend Engineer" } Test steps: Render component->Click “Save Job”->Simulate API success/failure and toggle saved state Postconditions: Button updates and localStorage toggled Expected output:

### Success: “Saved” button visible

### Failure: “Network fail” message shown

### Unsave: Button returns to “Save Job”

### Actual output: Matches expected behavior Pass or Fail: Pass

### Test case ID, name: FRONT-9 – ThemeToggler: light/dark switch

### New or old: New Test items: ThemeToggler → icon rendering, click toggle, localStorage sync Test priority: Medium Dependencies: ThemeContext, localStorage Preconditions: localStorage.appTheme = “light” or “dark” Input data: theme = “light” → expect moon; theme = “dark” → expect sun Test steps: Render ThemeToggler inside ThemeProvider->Click toggle button twice Postconditions: Theme cycles between light and dark Expected output:

### Moon icon → Sun icon → Moon icon

### localStorage.appTheme updated accordingly

### Actual output: Matches expected behavior Pass or Fail: Pass

### Test case ID, name: FRONT-10 – Type component: open/close, selection, and event handling

### New or old: New Test items: Type component → dropdown behavior, event listening, onChange trigger Test priority: High Dependencies: Browser window events (jobs:types, jobs:typeSelect) Preconditions: Component rendered Input data: types = [“Full-time”, “Internship”], selectedType = null Test steps: Click “Type” button to open panel->Simulate jobs:types event->Select “Contract” or “Any type” option Postconditions: onChange triggered, event dispatched, panel closes Expected output:

### Selecting type: fires onChange('Contract') and dispatches jobs:typeSelect

### Selecting “Any type”: fires onChange(null)

### “No types detected yet…” shown when list empty

### Actual output: Matches expected Pass or Fail: Pass

# Automated Testing Report

Describe briefly the automated testing you have done, including where the test code resides in your code repository, what test frameworks are used, and the screen shots or generated testing report.

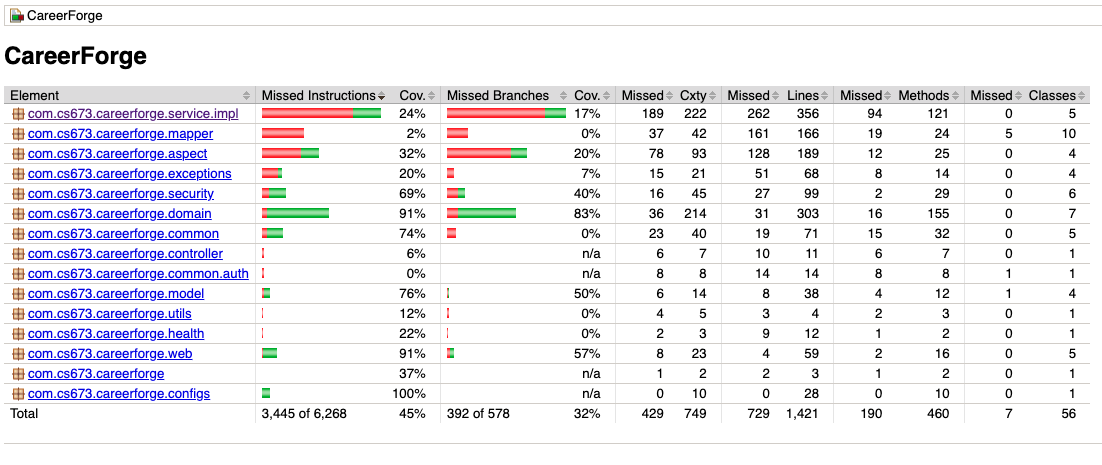
**Backend:**

All of our automated unit tests can be found under the code/backend/src/test directory of our repository, organized by package to mirror the structure of the main application. Our Selenium tests can be found in src/test, with our feature files to be built in the resources folder and our pages/steps to exist in the java folder. The frameworks we are using are JUnit5, Mockito, Spring Boot Test, Spring Test, and Selenium WebDriver.

* + JUnit5 - Core framework used in our unit tests
  + Mockito - Used for mocking dependencies that allows us to stub method calls and verify interactions
  + Spring Boot Test Framework - Used to load the full Spring context and specific beans for integration testing
  + Spring Test - Used MockMvc to simulate HTTP requests to controllers without needing to start a real server
  + Selenium WebDriver - Framework to drive browser testing and simulate end to end user behavior

Our unit test results can be found in the code/backend/target/surefire-reports folder.

Also, we have a jacoco maven plugin that allows us to analyze code coverage. If you execute maven verify we get the following test reports which can be found in code/backend/target/site:



We’re able to drill down into each element if we open the index.html in the browser. We have about 45% of lines covered (3,445 missed out of 6,268), 32% of branches covered, (392 missed out of 578), cyclomatic complexity has 429 missed paths, and 41% of method coverage. Overall, we can see our backend has ~51% line coverage. Entities and security logic are well tested, but services and controllers are under-tested and need more unit and integration tests.

**Frontend:**

All automated frontend unit tests are located under code/frontend/src/, organized by feature to mirror the source structure. We use Jest and React Testing Library (with @testing-library/user-event) in a jsdom environment. Tests focus on rendering, behavior, and accessibility-friendly queries, and they mock APIs where needed for isolation.

*JUnit5 / Mockito / Spring Test do not apply to the frontend; instead we use:*

* Jest – Core test runner and assertions
* @testing-library/react and @testing-library/jest-dom – DOM queries & matchers
* @testing-library/user-event – Realistic user input (typing/clicking)
* jsdom – Browser-like DOM environment for unit tests

*Representative test locations:*

* src/components/asideAndToggler/\_\_tests\_\_/Aside.test.tsx
* src/components/asideAndToggler/\_\_tests\_\_/Field.test.tsx
* src/components/buttons/\_\_tests\_\_/MyJobsButton.test.tsx
* src/components/loginAndRegistration/\_\_tests\_\_/LoginForm.tdd.test.tsx
* src/components/loginAndRegistration/\_\_tests\_\_/RegisterForm.test.tsx
* src/components/loginAndRegistration/\_\_tests\_\_/validation.test.tsx
* src/components/loginAndRegistration/\_\_tests\_\_/ApplyJobButton.tdd.test
* src/components/loginAndRegistration/\_\_tests\_\_/SavedJobApplyButton.tdd.test.tsx
* src/components/loginAndRegistration/\_\_tests\_\_/SaveJobButton.test.tsx
* src/components/loginAndRegistration/\_\_tests\_\_/themeToggler.tdd.test.tsx
* src/components/loginAndRegistration/\_\_tests\_\_/Type.tdd.test.tsx

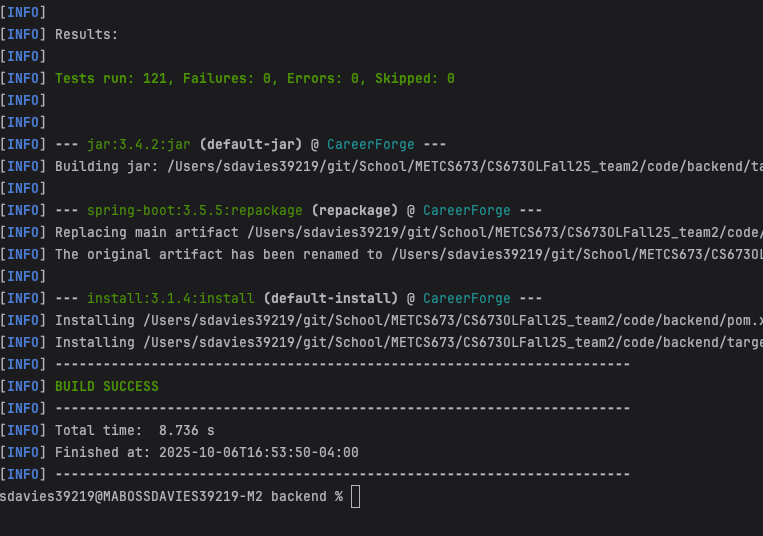
Our global setup is src/setupTests.ts, which imports @testing-library/jest-dom and polyfills TextEncoder/TextDecoder, window.matchMedia, ResizeObserver, and window.scrollTo. Frontend unit test results appear in CI logs; optional coverage can be generated with jest --coverage to code/frontend/coverage/.

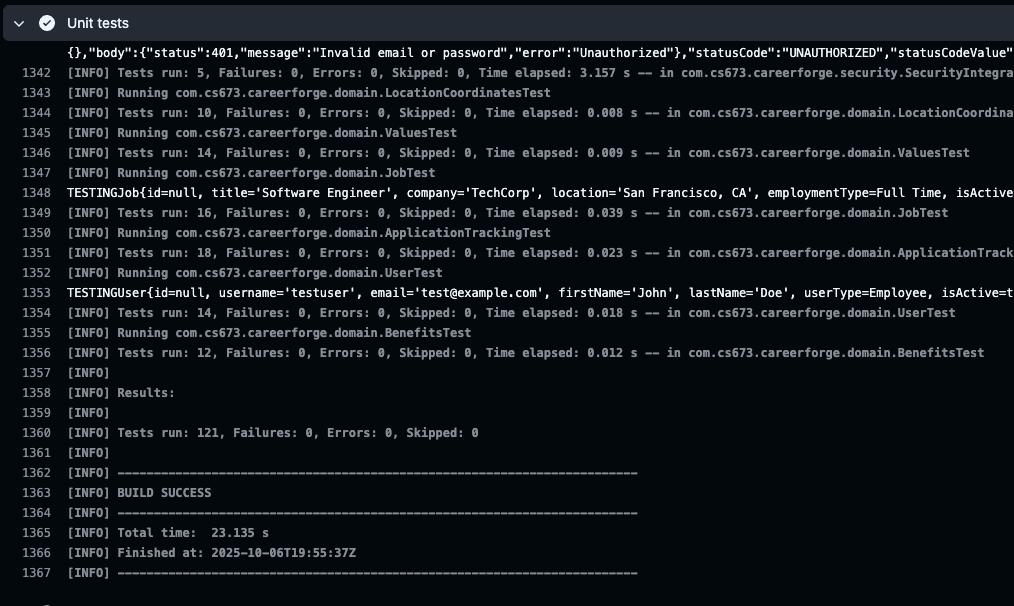
# Testing Metrics

In this section, you shall report any metrics used for the evaluation, e.g. # of test cases, test coverage, defects rate, etc.

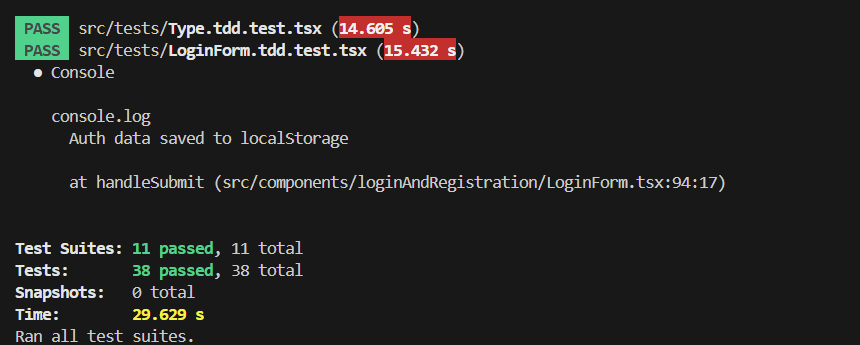
**Backend:**

* **Test coverage:** At the moment we have a total of 121 unit tests
* **Test pass rate:** All tests are now passing 100% locally and on GitHub





**Frontend:**

* **Test coverage:** 11 unit test files; 38 total test cases (current snapshot).
* **Test pass rate:** All frontend tests passing locally; CI reflects the same on recent PRs.
* 

# References

**Backend:**

* JUnit 5 (Jupiter) User Guide
* Mockito Documentation
* Spring Boot Reference — Testing
* Spring Framework Testing (MockMvc)
* Selenium WebDriver Documentation (for acceptance/system tests)
* Postman Documentation (collections; Newman if used in CI)

Project files: code/backend/src/test/java/\*\*, test resources in code/backend/src/test/resources/\*\*, Selenium tests under code/backend/src/test/java/\*\* (or your e2e module if separate), Postman collection (e.g., docs/postman/CareerForge.postman\_collection.json), and unit test reports in code/backend/target/surefire-reports/.

**Frontend:**

Jest, React Testing Library, and @testing-library/user-event official documentation

Project files: src/setupTests.ts, component files under src/components/\*\*, and tests under src/components/\*\*/\_\_tests\_\_

# Glossary