

**CS673 Software Engineering**

**Team 6 - FitFusion**

**Software Test Document**

| Team Member | Role(s) | Signature | Date |
| --- | --- | --- | --- |
| Haoran Zheng | Team Leader | *Haoran Zheng* | 10/12/2024 |
| Jiankun Dong | Design and Implementation Leader | *Jiankun Dong* | 10/12/2024 |
| Chengqin Li | QA Leader | *Chengqin Li* | 10/12/2024 |
| Yuhan Pan | Requirement Leader | *Yuhan Pan* | 10/12/2024 |
| Yunrui Huang | Configuration Leader | *Yunrui Huang* | 10/12/2024 |
| Hangqi Wu | Security Leader | *Hangqi Wu* | 10/12/2024 |
| Team Member | Role(s) | Signature | Date |
| Haoran Zheng | Team Leader | *Haoran Zheng* | 10/12/2024 |

**Revision history**

| **Version** | **Author** | **Date** | **Change** |
| --- | --- | --- | --- |
| **V1** | **All members** | **11.06** | **Initial version of STD** |
| **V2** | **Jiankun Dong** | **12.4** | **Final version of STD** |

[Testing Summary](#_heading=h.gjdgxs)

[Manuel Tests Reports](#_heading=h.30j0zll)

[Automated Testing Reports](#_heading=h.1fob9te)

[Testing Metrics](#_heading=h.3znysh7)

[References](#_heading=h.2et92p0)

[Glossary](#_heading=h.tyjcwt)

# Testing Summary

The testing for **frontend** part is done by Haoran Zheng, Yuhan Pan, Chengqin Li and Hangqi Wu, to write operation scripts with Cypress and simulate user inputs for testing the submission status and page jumping feasibility. Through End-to-End testing, we can ensure the user’s experience when accessing our project, detect possible integration issues between different components or pages within our project, and validate the entire workflow of our application. Also, with the composing of tests to Docker, we can check the validity of the whole system through automatically running the tests and get result reports, which improves our confidence in deployment.

The **backend** testing process, by Jiankun Dong and Yunrui Huang, ensures the reliability and functionality of core system features, including server availability, user account management, records, exercises, and workout plans. Tests were conducted across various stages, beginning with unit tests for individual API endpoints, integration tests to validate inter-module interactions, and system tests for end-to-end functionality. Acceptance testing confirmed that the backend met project requirements, while regression testing continues to maintain stability as updates are made. This comprehensive approach has verified successful CRUD operations, field validations, access controls, and data integrity.

# Manual Testing Report

## Locally manual testing for frontend

For frontend, the manual tests for basic functions are mainly focused on testing the registration of new users, and the login for existing users. The tool used is Cypress.

1. Test for registration.

Test case ID, name: 1. register\_test.cy.js

New or old: New

Test items: Registration function

Test priority: High

Dependencies: Need to run both frontend and backend first.

Preconditions: Need to run both frontend and backend first, and setting correct port cypress’s configuration file.

input data: Example new user’s email, username and password.

Test steps: Visit registration page, check visibility of input box, input data, click button, and finally check if the link jumps back to login(which means registration finished).

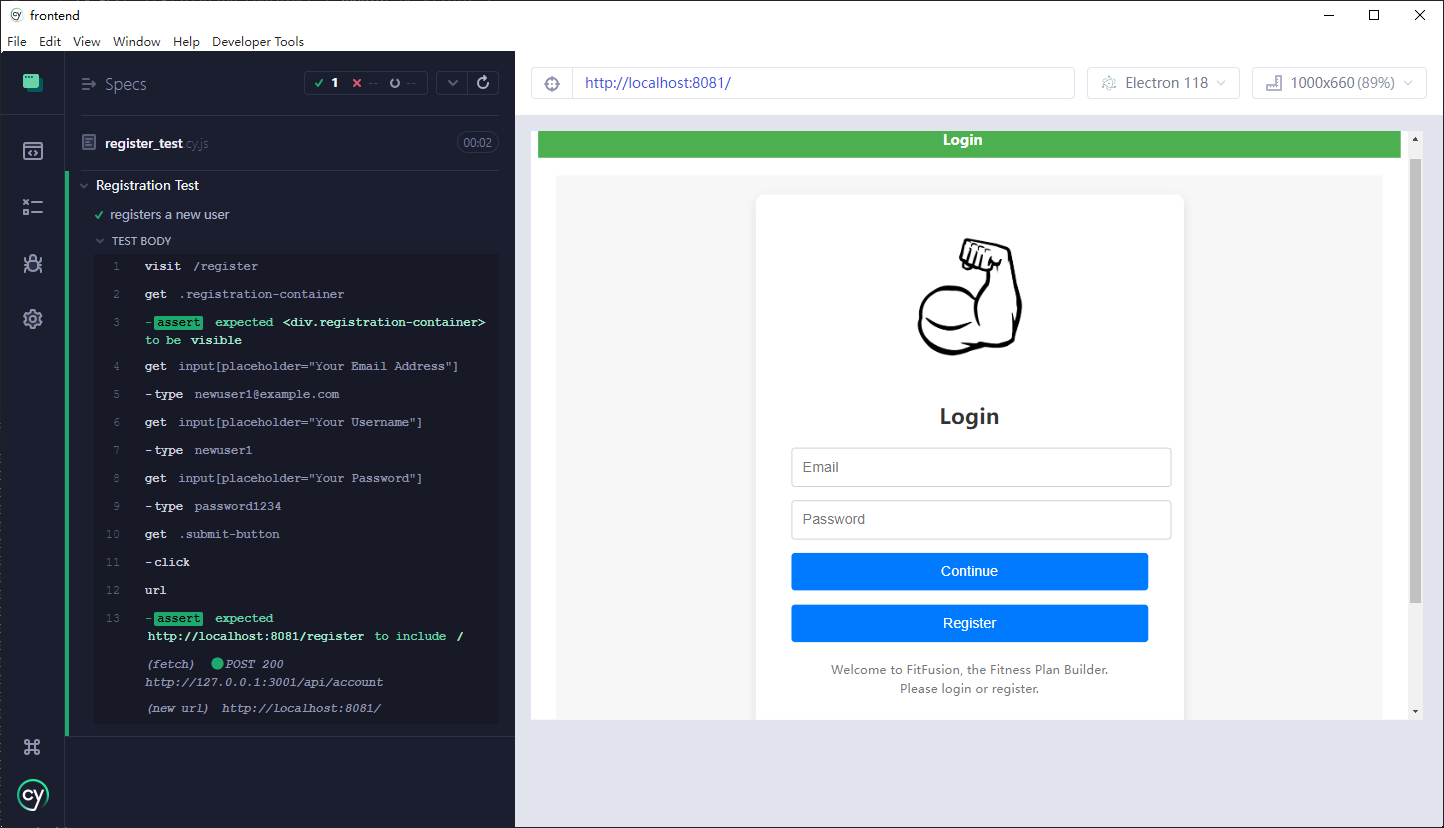
Postconditions: The example user’s information should not be the same, as deleting the test user has not been implemented yet.

Expected output: Register successfully and jump to the login page.

Actual output: Register successfully and jump to the login page. Test users can be found in the backend database.

Pass or Fail: Pass.

Additional notes: Screenshot



1. Test for login(fail).

Test case ID, name: 2. login\_fail.cy.js

New or old: New

Test items: Login function

Test priority: Medium

Dependencies: Need to run both frontend and backend first.

Preconditions: Need to run both frontend and backend first, and setting correct port cypress’s configuration file.

input data: An invalid wrong user’s email and password that does not exist in the user account database. .

Test steps: Visit login page, input email at row 1 and password at row 2, click login button, and check the console log to find the keyword “Invalid email or password”.

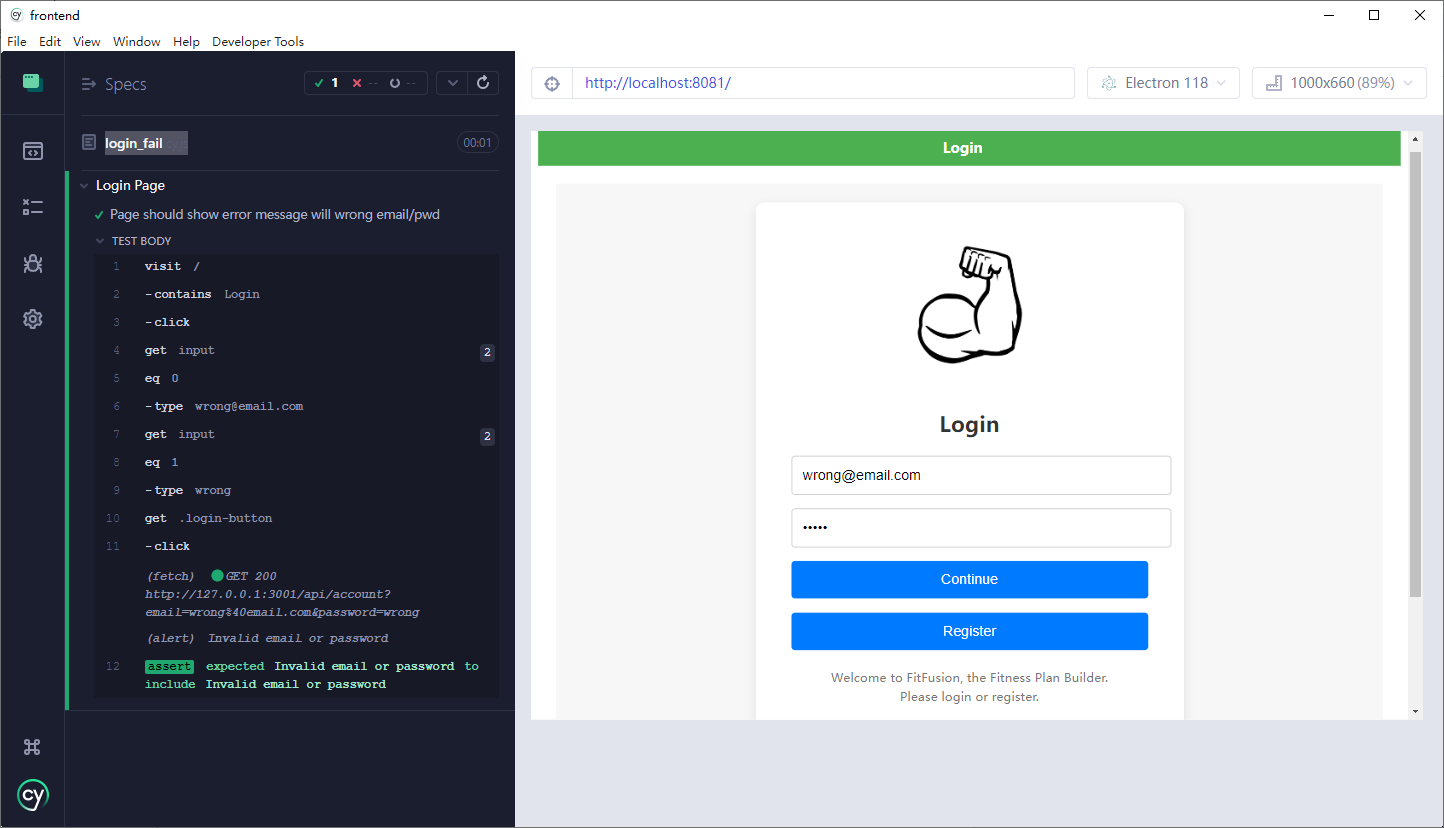
Postconditions: The user info should be invalid, and error setting is correct for testing to checkout.

Expected output: Console log shows error message, cannot push to next page(welcome page) as login failed.

Actual output: Console log shows error message, cannot push to next page(welcome page) as login failed. Test return success.

Pass or Fail: Pass.

Additional notes: Screenshot



1. Test for login(success).

Test case ID, name: 3. login\_test.cy.js

New or old: New

Test items: Login function

Test priority: High

Dependencies: Need to run both frontend and backend first.

Preconditions: Need to run both frontend and backend first, and setting correct port cypress’s configuration file.

input data: A valid user’s email and password that exist in the user account database.

Test steps: Visit login page, input email at row 1 and password at row 2, click login button, and check the console log to find the keyword “Successfully logged in”.

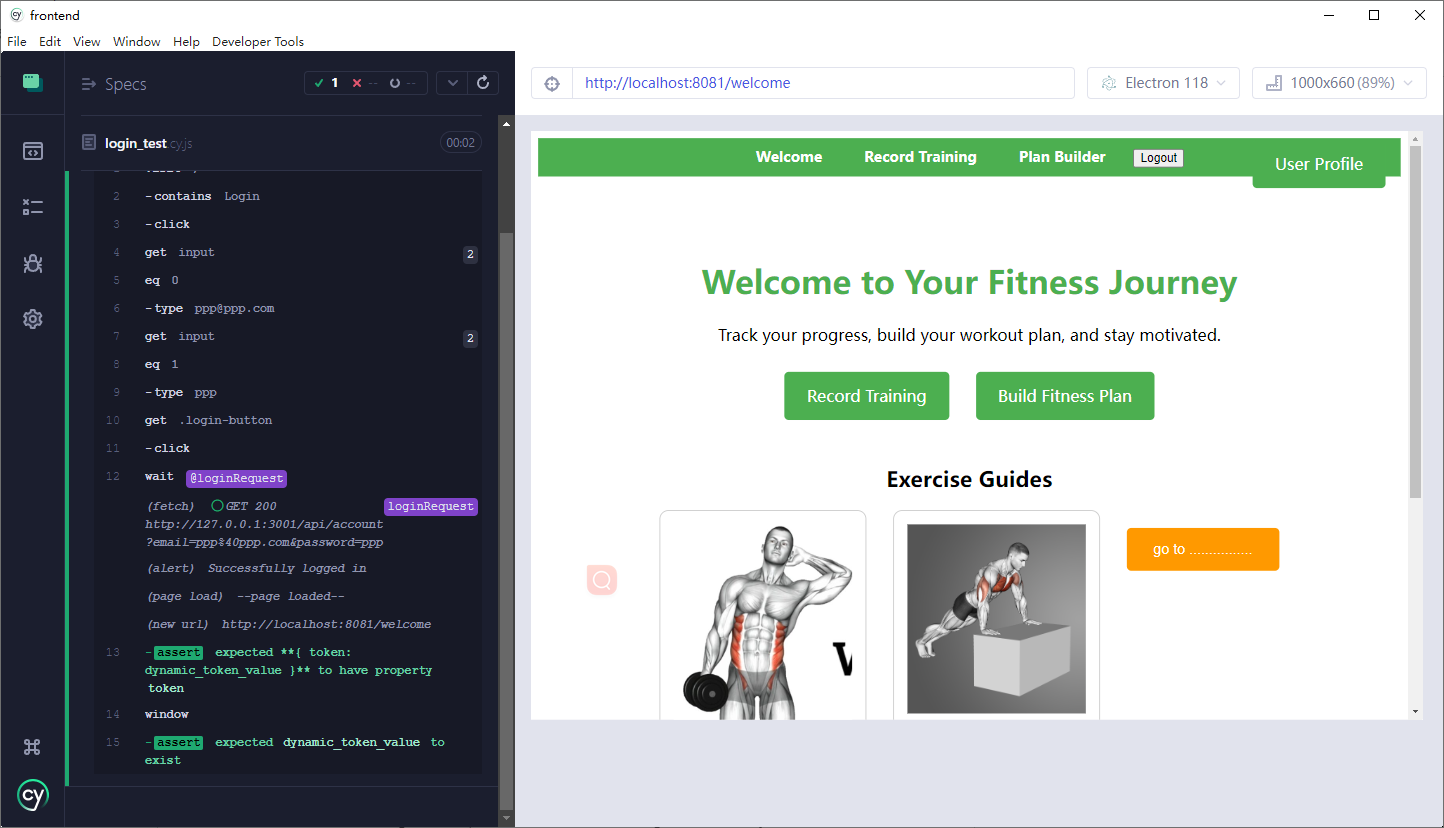
Postconditions: The user info should be valid, and the login success info setting is correct for testing to checkout.

Expected output: Console log shows success message, and push to next page(welcome page) as login successful.

Actual output: Console log shows success message, and push to next page(welcome page) as login successful. Return test success.

Pass or Fail: Pass.

Additional notes: Screenshot



1. Test for record.

Test case ID, name: record\_page

New or old: New

Test items: record function

Test priority: High

Dependencies: Need to run both frontend and backend first.

Preconditions: Need to run both frontend and backend first, and setting correct port cypress’s configuration file.

input data: The training that users did.

Test steps: Visit record page, check visibility of input box, input data, click button, and finally check if it stores successfully.

Postconditions: The example should store the input values successfully.

Expected output: The message of user input will be shown on the page.

Actual output: The message shows successfully.

Pass or Fail: Pass.

1. Test for user profile page.

Test case ID, name: user\_profile page

New or old: New

Test items: store function

Test priority: High

Dependencies: Need to run both frontend and backend first.

Preconditions: Need to run both frontend and backend first, and setting correct port cypress’s configuration file.

input data: The information of the user.

Test steps: Visit user\_profile page, check visibility of input box, input data, click button, and finally check if it stores successfully.

Postconditions: The example should store the input values successfully.

Expected output: Go to the user\_profile page, the information will be shown.

Actual output: The message shows successfully.

Pass or Fail: Pass.

## Locally manual testing for backend

For the backend, we use Postman for manual tests.

For modules that came after the user account, we opted to use automatic testing due to the complexity.

1. Test for register account

Test case ID, name: account registration test

New or old: New

Test items: Registration function

Test priority: Medium

Dependencies: N/A

Preconditions: N/A

input data:

{

Email: “testEM1@bu.edu”,

Name: “testUN1”,

Password: “”password

}

Test step: Send a POST operation to http://localhost:3001/api/account/

Postconditions: N/A

Expected output:

A response with body containing the field token and a value for the token.

200 as code.

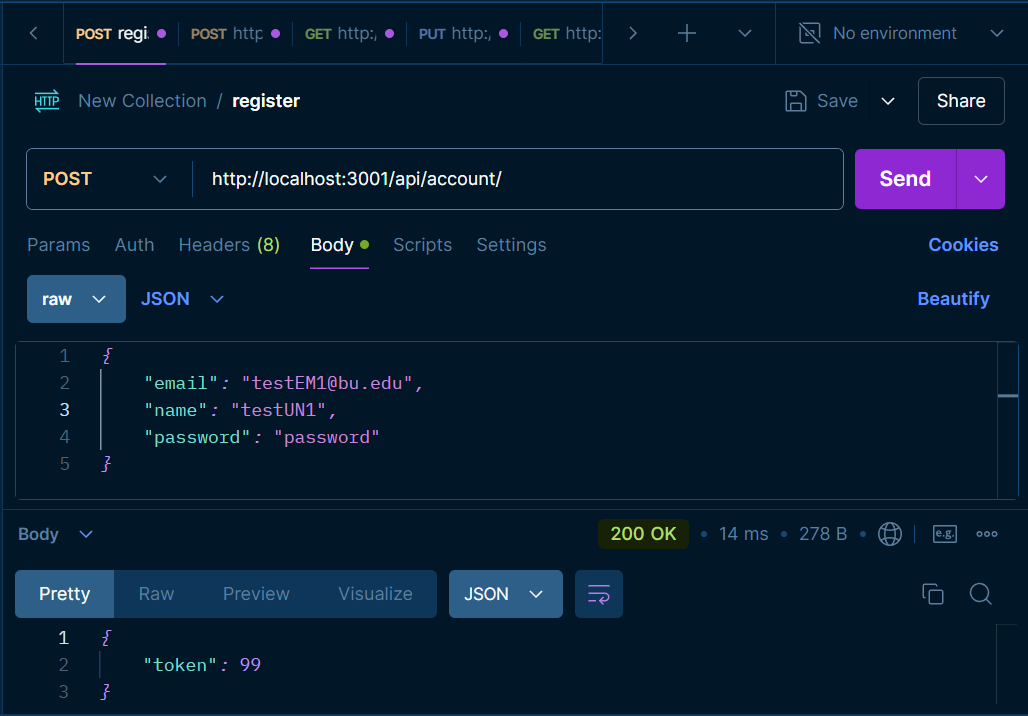
Actual output:

Code 200,

Body has token field and value

Pass or Fail: Pass.

Additional notes: Screenshot



1. Test for login to account

Test case ID, name: account login test

New or old: New

Test items: Login function

Test priority: Medium

Dependencies: User account login

Preconditions: We have registered the user account with password: password

input data:

Username or email address, password

Test step:

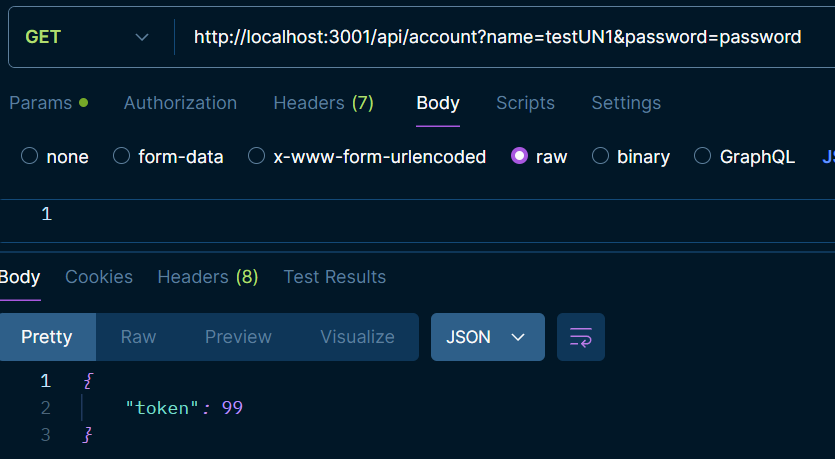
Send a GET operation to <http://localhost:3001/api/account/?name=testUN1&password>=password

Postconditions: N/A

Expected output: Code 200 and token value

Actual output: Code 200 and token value

Pass or Fail: Pass.

Additional notes: Screenshot  


1. Test for account deletion

Test case ID, name: account deletion test

New or old: New

Test items: Deletion function

Test priority: Medium

Dependencies: User account registration

Preconditions:

We have registered the user account with password: password, and we have recorded the token value as userID

Input data:

header{ x-user-id: userID }

Body { password: “password”}

Test step:

Send a DELETE operation to <http://localhost:3001/api/account>, with header and body value as shown in input datat

Postconditions: N/A

Expected output:

{

"msg": "Account deleted successfully"

}

Actual output:

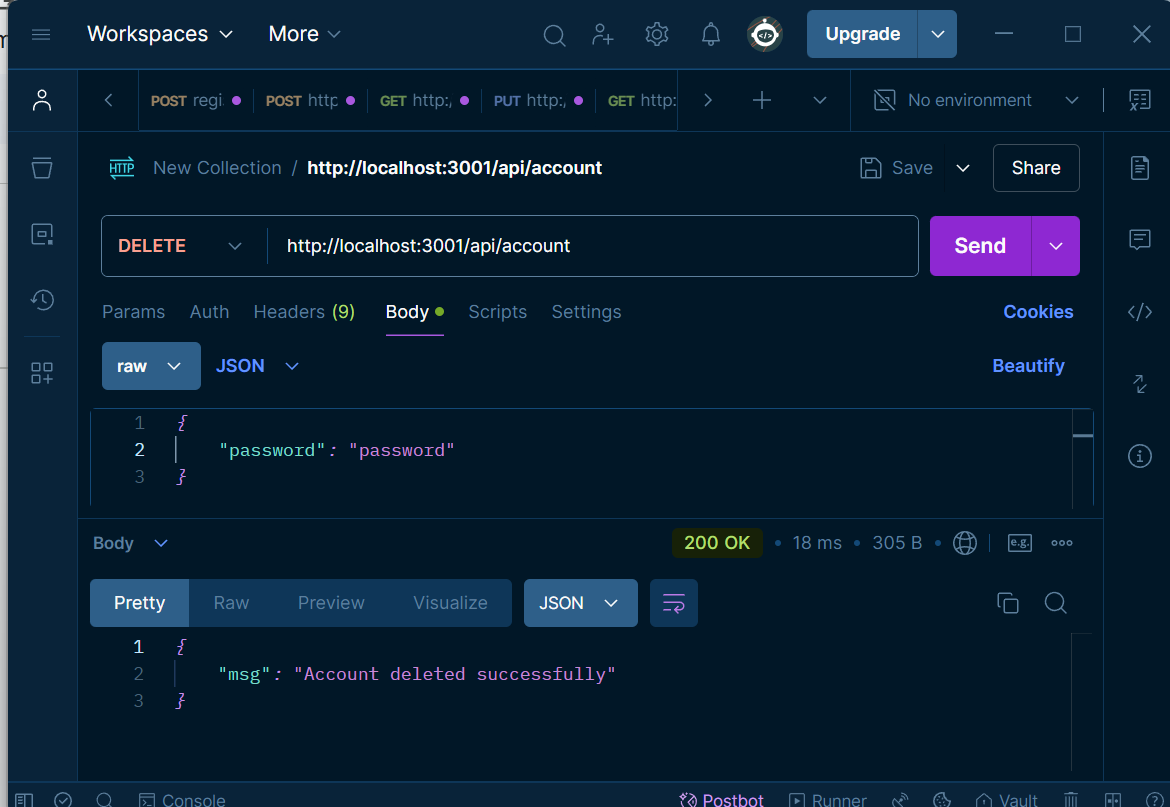
{

"msg": "Account deleted successfully"

}

Pass or Fail: Pass.

Additional notes: Screenshot



1. Test Searching Function in Video Illustration Part

**1. Define the Feature Behavior:**

• Given the user is on the exercises page,

• When they type “arms” in the search box and click search,

• Then they should see exercises related to “arms.”

2. **Write the Test (Cypress)**:

First check if the search box is present, then simulate typing “arms,” clicking the search button, and finally verifying that the results contain exercises related to “arms.

Source code:

describe('Search and Select Body Part', () => {

beforeEach(() => {

cy.intercept('GET', '\*\*/exercises/bodyPartList').as('getBodyPartList');

cy.intercept('GET', '\*\*/exercises?limit=1000&offset=0').as('getExercises');

cy.visit('http://localhost:3002');

cy.wait('@getBodyPartList');

cy.wait('@getExercises');

});

it('should search for exercises based on body part input', () => {

cy.get('[data-testid="search-input"]').should('exist').type('arms');

cy.get('[data-testid="search-button"]').should('exist').click();

cy.get('[data-testid="exercise-card"]', { timeout: 10000 }).should('exist');

cy.get('[data-testid="exercise-card"]')

.should('have.length.greaterThan', 0)

.each(($card) => {

cy.wrap($card)

.find('div, p, span')

.contains(/Arms/i)

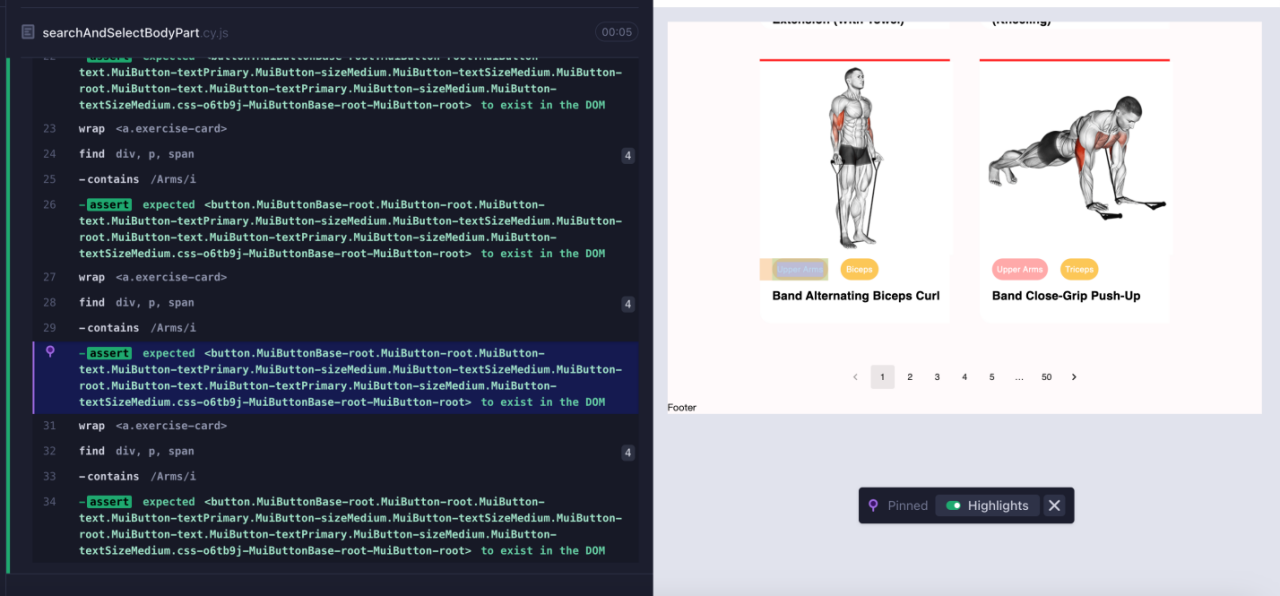
.should('exist');

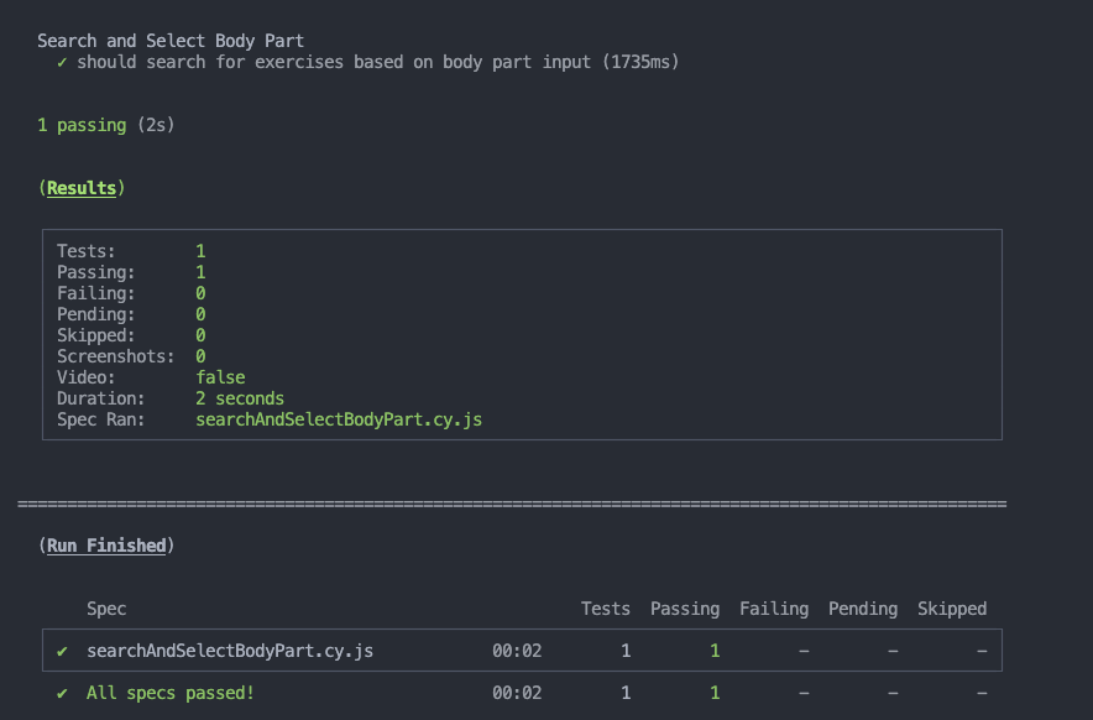
});

});

});

Success Screen Shot:





# Automated Testing Report

For the automated testing process, the Cypress testing framework was utilized to perform end-to-end (E2E) testing of the registration functionality in a Vue.js application. The test scripts are structured in a well-organized folder hierarchy within the code repository to maintain clarity and efficiency.

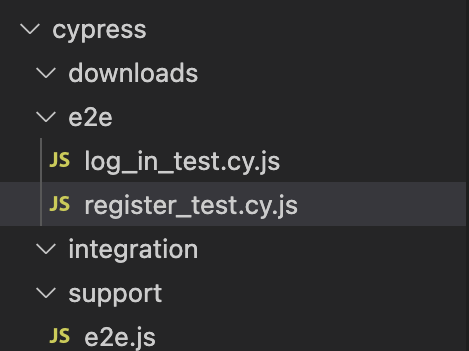
**Test code repository structure:**

* **cypress/e2e**: This folder contains the primary E2E test scripts, including log\_in\_test.cy.js and register\_test.cy.js.
* **cypress/support**: Contains e2e.js (currently empty but can be used for custom Cypress commands or pre-test setup).
* **cypress/integration**: Reserved for any additional integration test cases.
* **cypress/downloads**: For storing any files that Cypress downloads during testing.

**Test Framework and Reporting**

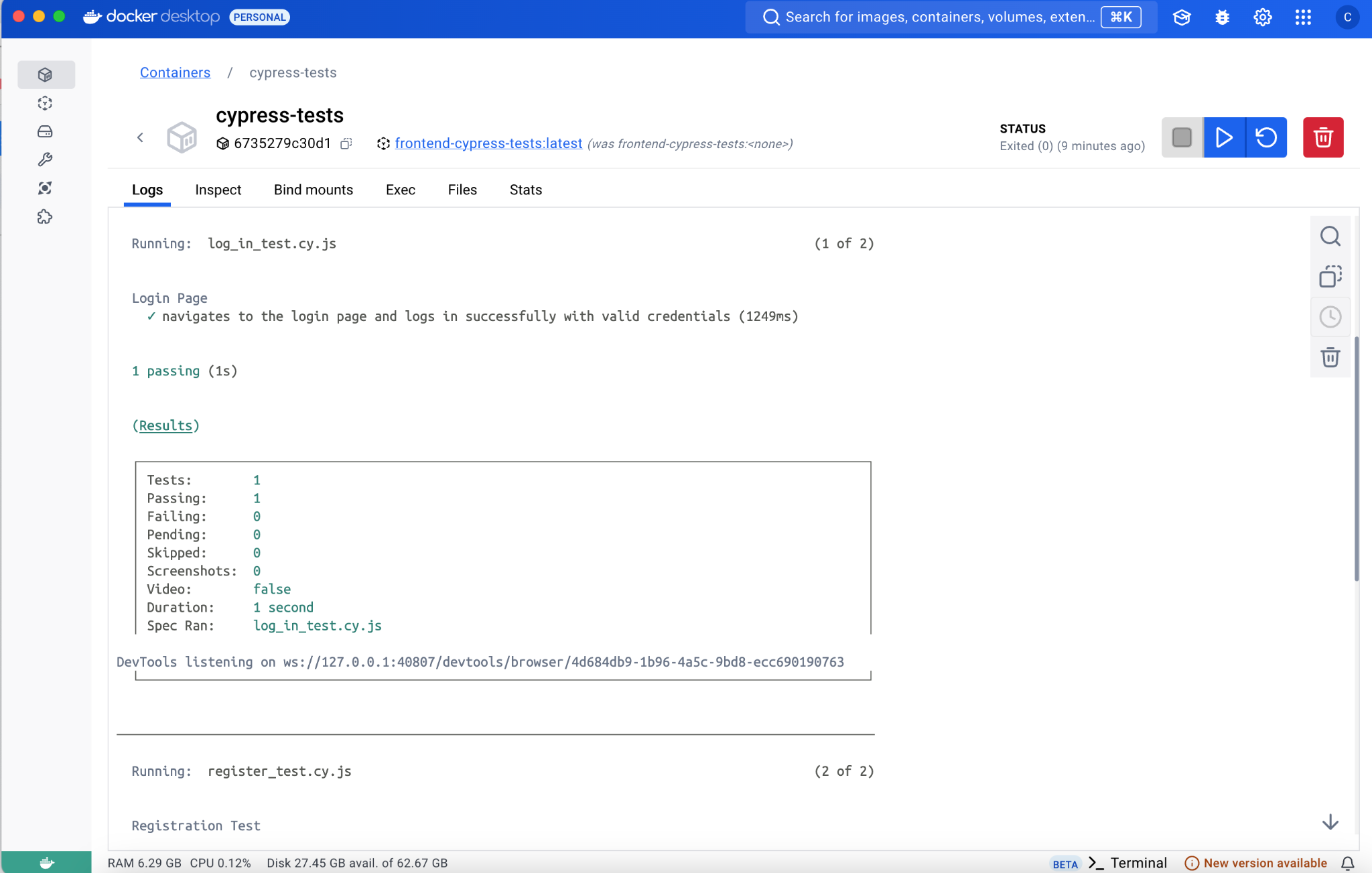
* **Framework Used**: Cypress version specified in cypress/included:13.15.1.
* **Execution Command**: The tests are run using npx cypress run within a Docker container, ensuring consistent execution across different environments. If we run on the VSC to compose up to the docker, and run Cypress test automatically, we just need to use the command: docker-compose up --build on the terminal.
* **Test Reports**: Cypress generates a summary report upon completion, which includes:
  + **Number of Tests Passed/Failed**.
  + **Screenshots and Videos**: Automatically captured for failed tests to aid debugging.
  + **Detailed Logs**: Console outputs provide information on test steps and any issues encountered.

### Frontend Test Details:

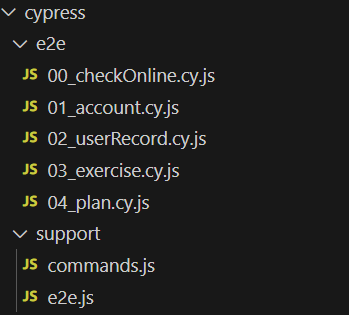


1. **Registration Test** :
   * This test suite navigates to the registration page, inputs user details, submits the form, and verifies that the user is redirected appropriately.
   * The test includes assertions to ensure the registration container is visible and checks the final URL for successful navigation.
2. **Login Test** :
   * The test suite verifies the login process by navigating to the login page, entering valid credentials, and simulating a login attempt.
   * An alert is captured to confirm the message indicating successful login.
   * A URL verification step is included to check for redirection after a successful login.

**Screen shots:**

****

**Backend testing details:**



### 00\_checkOnline.cy.js

* **Purpose:** Verifies that the backend server is online and responsive.
* **Test:** Sends a GET request to the root endpoint (/) and checks for a status code of 200 and a specific response property (test: 'success'), confirming server availability​(00\_checkOnline.cy).

### 2. 01\_account.cy.js

* **Purpose:** Tests the user account management endpoints.
* **Tests:**
  + **Account Creation:** Verifies that a new user account can be created with valid data.
  + **Duplicate Prevention:** Attempts to create an account with the same credentials and expects an error (status 400 with duplicate error).
  + **Login:** Checks successful login with correct credentials, ensuring a token is returned.
  + **Invalid Login:** Attempts to log in with incorrect credentials and expects a 401 error.
  + **Account Deletion:** Verifies that deleting an account requires a password and denies deletion with an incorrect password.
  + **Delete Non-existent Account:** Ensures an appropriate error (404) when trying to delete a non-existent account.
  + **Clean-up:** Deletes the test user after all tests to maintain data integrity​(01\_account.cy).

### 3. 02\_userRecord.cy.js

* **Purpose:** Tests operations on user records, including adding, updating, retrieving, and deleting records.
* **Tests:**
  + **Login and Token Generation:** Logs in the user and obtains a token for authentication in subsequent record tests.
  + **Add Record:** Creates a new record (e.g., exercise data), verifying that it returns the correct response with a record ID.
  + **Get Record:** Retrieves the user’s records and verifies that data is returned as expected.
  + **Update Record:** Modifies an existing record, verifying that the changes were successful.
  + **Delete Record:** Removes the record and ensures it is no longer accessible.
  + **Clean-up:** Deletes the test data if any test fails, and performs a final clean-up after all tests​(02\_userRecord.cy).

### 4. 03\_exercise.cy.js

* **Purpose:** Tests the exercise management endpoints, covering user-created and system exercises.
* **Tests:**
  + **Exercise Creation:** Validates required fields, such as name and type, to ensure a successful creation.
  + **Exercise Update:** Updates a specific exercise and verifies that the modified data is saved.
  + **Retrieve Exercises:** Fetches all exercises for the user, verifying that at least one exercise is present.
  + **System Exercise Constraints:** Tests access restrictions for system exercises by attempting to update and clone system exercises. Unauthorized modifications should fail, but cloning is allowed.
  + **Exercise Deletion:** Creates a temporary exercise, deletes it, and verifies it is no longer retrievable.
  + **Clean-up:** Deletes any test-created exercises and the test user account​(03\_exercise.cy).

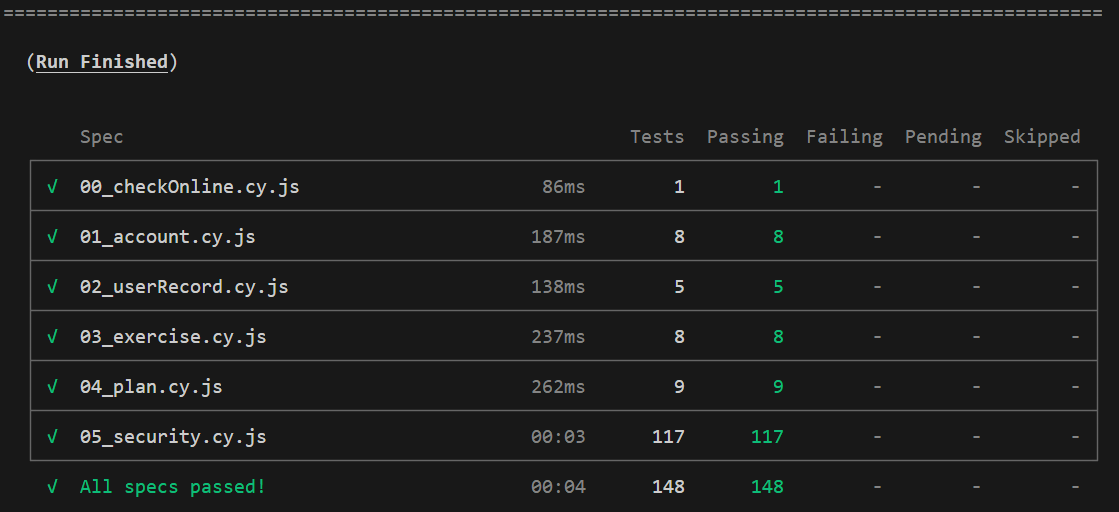
### 5. 04\_plan.cy.js

* **Purpose:** Tests the workout plan management endpoints, including creating, retrieving, updating, and controlling access to workout plans.
* **Tests:**
  + **Plan Creation Validation:** Ensures that required fields are validated during plan creation.
  + **Plan Creation:** Creates a workout plan with specific attributes (e.g., frequency, days of the week) and verifies successful creation.
  + **Status Update Validation:** Checks that the status update requires a status field, returning an error if missing.
  + **Retrieve Active Plans:** Verifies active plans for a specified date, ensuring data retrieval accuracy.
  + **Access Control:** Tests that a user cannot access or modify another user’s plans, enforcing ownership restrictions.
  + **Clean-up:** Deletes test plans, exercises, and the user account after testing is completed​(04\_plan.cy).

### 5. 05\_security.cy.js

* **Purpose:** Tests the security related issue of the website, mainly injection attacks on different endpoints
* **Tests:**
  + **Account Endpoints Protection:** test injection attack against login
  + **Exercise Endpoints Protection:** test injection attack against exercises
  + **Workout Plan Endpoints Protection:** test injection attack against workout plan
  + **Workout Record Endpoints Protection:** test injection attack against workout record

Test passing screenshot:



# Testing Metrics

* **Test Case Pass Rate**
  + **Definition**: The pass rate of executed test cases, indicating the stability of the application.
  + **Formula**: Pass Rate = (Number of Passed Test Cases / Total Test Cases) × 100%
  + **Example Data**: In this iteration, a total of 2 test cases were executed, all of which passed, resulting in a test case pass rate of:Pass Rate= 2/2 × 100% = 100%Pass Rate= 2/2 ​× 100% = 100%
* **Feature Coverage**
  + **Definition**: The proportion of key functional modules that are covered by test cases, ensuring that critical features are thoroughly tested.
  + **Assessment**: Currently, we have test cases covering registration and login functionalities. These two modules represent approximately 50% of the primary functions (assuming the main modules include registration, login, training records, and plan builder).
  + **Example Data**: Feature Coverage = 50%
* **Test Coverage**
  + **Definition**: The percentage of code lines covered by test cases.
  + **Tool**: While a specific test coverage tool has not been utilized, we estimate test coverage based on the covered feature modules.
  + **Example Data**: Test Coverage is approximately 50%.
* **Average Defect Resolution Time**
  + **Definition**: The average time taken to resolve defects, measuring the efficiency of the team in addressing issues.
  + **Data**: Currently, defect resolution time is not being tracked specifically, but we plan to use GitHub Issues or another defect management tool in future iterations to monitor this metric.
* **Person-Hours**
  + **Definition**: The total number of person-hours spent in testing, used to assess workload and resource allocation in testing efforts.
  + **Example Data**: In this iteration, an estimated 8 person-hours were spent on testing (with each team member contributing approximately 2 hours to writing and executing test cases).
  + **Application**: Person-hours data will help optimize resource allocation in future iterations, ensuring efficient testing processes.
* **Regression Test Success Rate**
  + **Definition**: The pass rate of test cases re-executed after code changes, ensuring that new features or fixes do not negatively impact existing functionalities.
  + **Formula**: Regression Test Success Rate = (Number of Passed Regression Tests / Total Regression Tests) × 100%
  + **Example Data**: In this iteration, 2 regression test cases were executed, all of which passed, resulting in a regression test success rate of 100%.
* **Defect Reproduction Rate**
  + **Definition**: The proportion of reported defects that can be successfully reproduced, indicating the accuracy of defect reporting and effectiveness of the testing process.
  + **Formula**: Defect Reproduction Rate = (Number of Reproduced Defects / Total Reported Defects) × 100%
  + **Example Data**: In this iteration, no defects were reported, so the defect reproduction rate is not applicable. Future iterations will track this metric as defects are reported.

# References

# Glossary