**Comprehensive End-to-End Testing with Cypress**

Demo x-cart

**Introduction**

This document details a **Cypress test suite** designed to assess the performance and stability of the **X-Cart demo application** Using Cypress and simulate user interactions and verify both the functionality and user interface of the application. The suite is structured to test various aspects of the system, from basic page load validation to complex user workflows such as login and checkout.

The suite is divided into four main categories of tests:

1. **Basic Tests**: Verify core functionality such as page load and essential content visibility.
2. **Functional Tests**: Simulate complete user journeys, including logging in, searching for products, adding items to the cart, and completing the checkout process.
3. **Edge Case Tests**: Evaluate how the system responds to unexpected inputs or scenarios, ensuring robust error handling.
4. **UI and Visual Tests**: Ensure that key visual components are correctly displayed and consistent across different pages.

Each category contains multiple test cases, all of which are documented below with detailed steps, expected results, and outputs obtained.

**Test Suites and cases**

1. **Basic Tests**

Basic tests are designed to check whether the core elements of the application are functioning correctly. These tests serve as the foundation for more complex testing, ensuring that pages load successfully, that URLs and page titles are correct, and that key elements are visible on the screen.

**Home Page Load and Title Verification**

**Objective**: Verify that the home page loads correctly with the appropriate title, URL, and content visibility.

**Steps**:

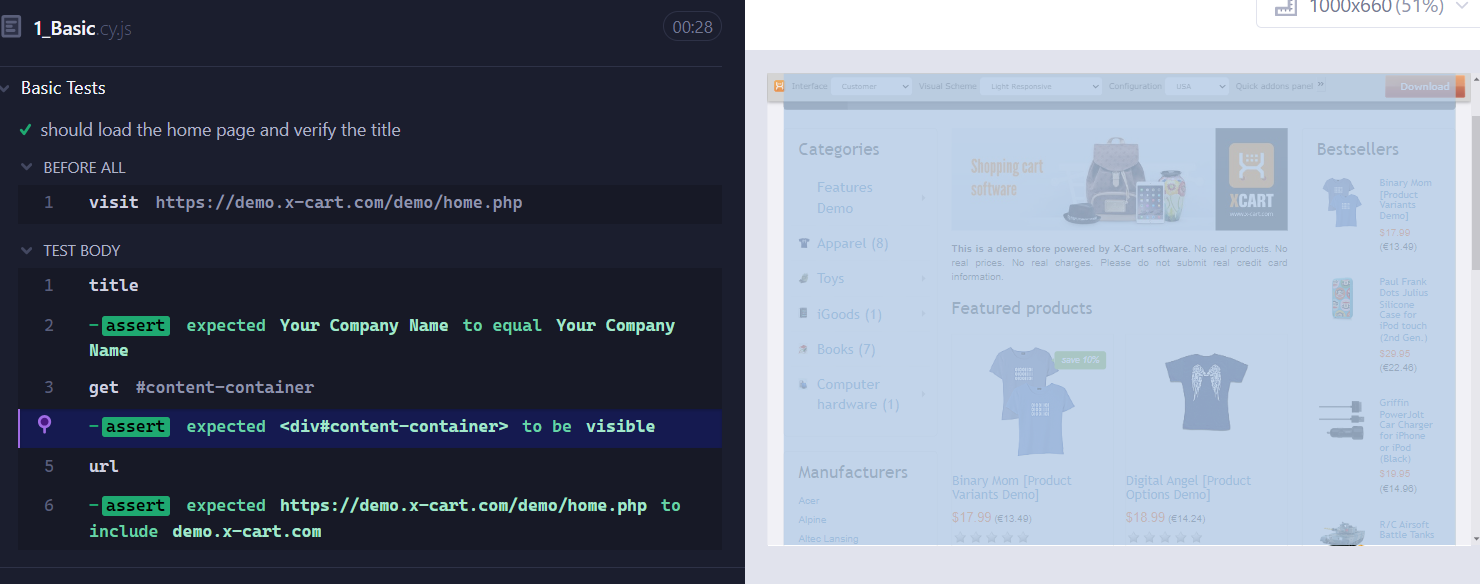
1. Visit the home page using cy.visit() to load the X-Cart demo.
2. Use cy.title() to confirm that the page title matches the expected value.
3. Check the URL with cy.url() to ensure the correct domain and path are loaded.
4. Ensure the main content container (the page's primary section) is visible using cy.get().

**Expected Results**:

* The home page should load successfully with the correct title ("Your Company Name") and the URL should contain "demo.x-cart.com".
* The content container should be visible, indicating that the page has fully loaded and is ready for interaction.

**Output Obtained**:

* The page loaded as expected, with the title, URL, and main content all correctly displayed. This test verified that the initial page load is functioning correctly, setting the stage for further testing of functionality.



**Functional Tests**

Functional tests go beyond simple page load and focus on core user interactions and application workflows. These tests mimic a typical user journey through the application and ensure that the system handles tasks like logging in, searching for products, adding items to the cart, and proceeding to checkout without any issues.

**User Login, Product Search, Add to Cart, and Checkout**

**Objective**: Test the entire user journey, including logging in, searching for products, adding items to the cart, and completing the checkout process.

**Steps**:

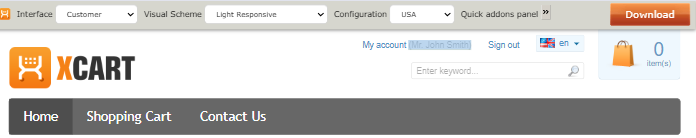
1. Navigate to the home page and click the Sign In button to trigger the login flow.
2. Use the cy.login() custom command to log in with valid credentials.
3. Search for a product (e.g., "phone") using the search bar.
4. Verify that relevant product results (e.g., "Apple" phones) are displayed.
5. Add one of the displayed products to the cart.
6. Confirm that the product has been successfully added to the cart by verifying the cart status.
7. Proceed to the checkout page and confirm that the user can finalize the purchase by clicking the appropriate buttons.
8. Validate that a success message appears after completing the order.

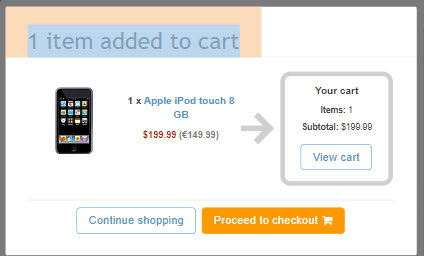
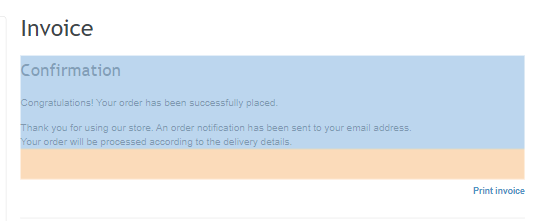
**Expected Results**:

* The user should be able to log in successfully and see their account name on the header.
* The product search should return relevant results that match the user's query.
* Adding items to the cart should update the cart status, and the checkout process should complete without errors.
* Upon successful order completion, the user should receive a confirmation message such as "Congratulations! Your order has been successfully placed."

**Output Obtained**:

* The test passed successfully. The user was able to log in, search for products, add them to the cart, and proceed through checkout to place an order. The confirmation message was displayed, confirming that the workflow functions as intended.





**Edge Case Tests**

Edge case tests are essential for evaluating how the system handles unexpected inputs or unusual user behavior. These tests help ensure that the application can gracefully handle errors, such as invalid form submissions or searches that return no results.

**Form Validation and Error Handling**

**Objective**: Test how the application handles invalid form inputs and searches for non-existent products.

**Steps**:

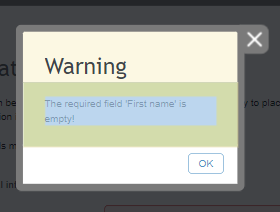
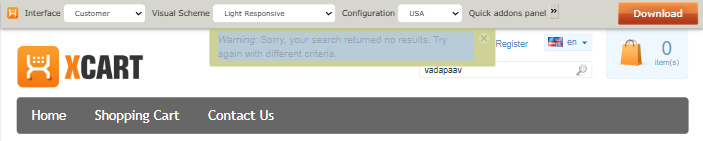
1. Attempt to register for a new account with empty first name and last name fields.
2. Submit the form with an invalid email address and check for the error message.
3. Perform a product search for a non-existent item **(e.g., "vadapaav")** and observe the system's response.

**Expected Results**:

* The form should also reject invalid email addresses, showing an appropriate validation message (e.g., "Email address is invalid! Please correct").
* The search for a non-existent product should return a warning message, informing the user that no results were found.

**Output Obtained**:

* The test successfully validated the error messages for missing fields and invalid email addresses. The search for a non-existent product returned a clear warning, "Warning: Sorry, your search returned no results." This confirms that the system is robust in handling invalid inputs.



**UI and Visual Tests**

**Objective:** Ensure that key UI components such as the header, footer, and images are visible on the page.

**Steps:**

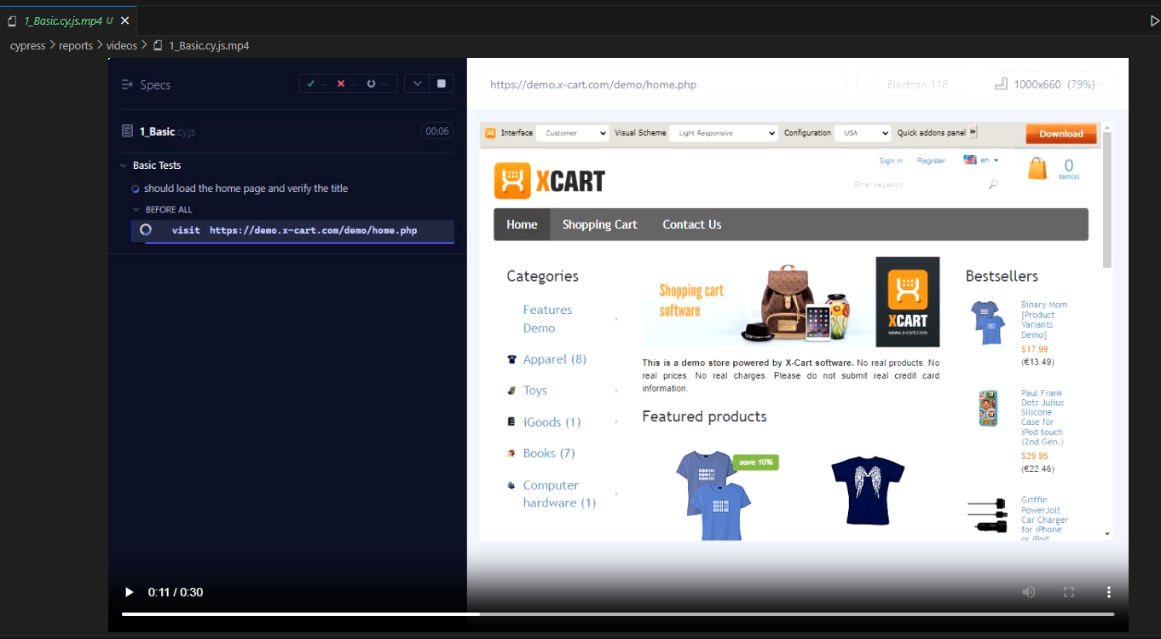
1. Visit the home page and check for the visibility of the header, footer, and product images.
2. Navigate through the site to verify consistency across different pages.
3. Click the sign-in button and open the login form.
4. Log in using the cy.login() custom command.
5. Verify that the user’s name is displayed in the header.

**Expected Results:**

The header, footer, and images should be visible and consistently rendered on all pages.

**Output Obtained:**

The test confirmed that the header, footer, and images were visible and consistent across all tested pages.

****

**Custom Commands**

**Login Command**

In this test suite, a custom command **cy.login()** is used to streamline the login process and simplify test cases that require user authentication. This custom command enhances test efficiency by reducing redundancy and making the login steps reusable across multiple tests.

**Objective**: Simplify the login process across various test cases.

**Steps**:

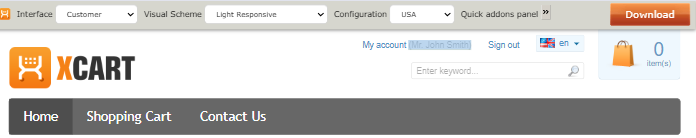
1. The custom command accepts email and password as arguments.
2. It automatically fills in the login form fields and submits the form, ensuring that the user is authenticated.

**Expected Results**:

* The custom command should enable easy and consistent login without repeating login steps in each test case.

**Output Obtained**:

* The custom login command worked as expected, and all test cases that relied on this command passed successfully.



**CI/CD Integration with Cypress Testing using GitHub Actions**

**Objective**

To establish a continuous integration and continuous deployment (CI/CD) pipeline using GitHub Actions, automating Cypress tests on each commit or pull request. The goal is to ensure the early detection of bugs and maintain code quality by running tests automatically.

**Description**

This integration allows Cypress tests to be executed automatically as part of the CI/CD pipeline within GitHub Actions. Each time a developer pushes new code or submits a pull request, Cypress tests are run. By automating test execution, this setup ensures timely feedback and helps identify issues before the code is merged into the main branch.

**Steps**

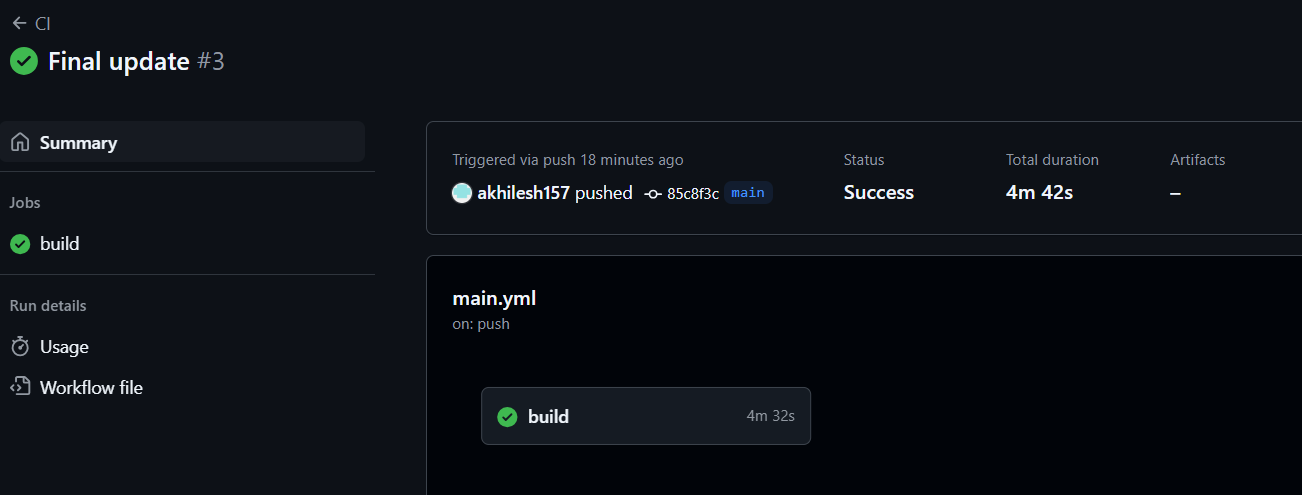
1. **Create a YAML file** in the .github/workflows directory of your repository.
2. **Define the workflow** by specifying the event triggers such as push or pull\_request. The pipeline is triggered whenever there is a new commit or pull request.
3. **Install dependencies** using Node.js and checkout the repository code.
4. **Run Cypress tests** using the cypress-io/github-action command. This runs all tests in headless mode by default, which is suitable for CI environments.
5. **Report test results** using GitHub Actions’ built-in reporting tools. GitHub Actions displays test results directly in the workflow interface, allowing developers to view the pass/fail status and error logs.

**Expected Results**

* **Automated Cypress test execution**: The pipeline should trigger Cypress tests on each push or pull request.
* **Test results reporting**: Test results (pass/fail) and any error messages should be visible in the GitHub Actions interface.
* **Failure handling**: If any Cypress tests fail, the pipeline should report the failure, causing the workflow to halt until the issues are resolved.

**Output Obtained**

* The CI/CD pipeline runs automatically on each commit or pull request.
* Cypress tests are executed without manual intervention, and the results are reported in real-time.
* If tests pass, the workflow succeeds; if any tests fail, the workflow is marked as failed, providing immediate feedback to developers.





**Generating Reports**

**Objective**

To generate detailed and insightful test reports using Cypress and Mochawesome, enabling developers and testers to analyze test results, debug issues, and review the overall health of the test suite.

**Description**

Mochawesome is a test reporter for Mocha, which Cypress uses as its underlying test framework. By configuring Cypress to work with Mochawesome, detailed reports can be generated in both **HTML** and **JSON** formats. These reports offer a comprehensive summary of test execution, including detailed information about test results, errors, screenshots, and videos.

**Steps**

1. Install Mochawesome reporter packages

*npm install --save-dev mochawesome mochawesome-report-generator cypress-mochawesome-reporter*

1. Configure Cypress to use Mochawesome

Update the Cypress configuration file (cypress.json or cypress.config.js) to specify Mochawesome as the reporter.

{ "reporter": "cypress-mochawesome-reporter",

"reporterOptions": {

"reportDir": "cypress/reports",

"overwrite": false,

"html": true }}

1. Run the tests to generate the reports:

When Cypress tests are executed, Mochawesome will generate detailed HTML and JSON reports summarizing the test execution results.

**Review the reports**:

Analyze the generated reports, which will include test suite and test case names, pass/fail status, error messages with stack traces, and optional screenshots/videos of failed tests.

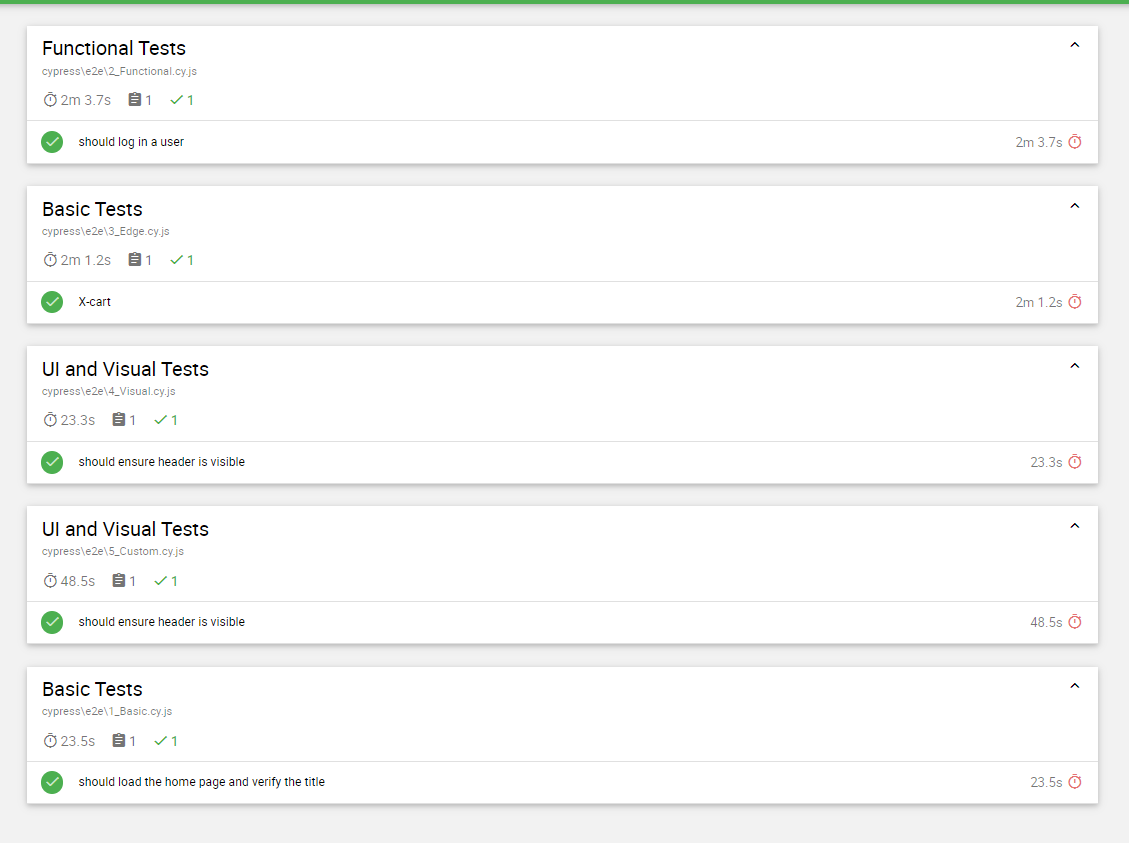
**Expected Results**

The reports should contain detailed information, including:

* Test suite and individual test case names.
* Pass, fail, or skip status for each test case.
* Error messages, stack traces for debugging, and screenshots of failures.
* Test execution time and duration for performance analysis.

**Output Obtained**

* Detailed Mochawesome reports were generated after running the Cypress tests, providing an easy-to-read HTML summary of test results. The reports included the pass/fail status of each test, error messages for failed tests, and attached screenshots where necessary.
* The JSON report can be used for further automation or integration with other tools.



**Challenges Faced**

1. **Environment Setup**: Configuring GitHub Actions to run Cypress tests with all dependencies, especially setting up the correct Node version and caching.
2. **Test Flakiness**: Some tests were unreliable in the CI/CD pipeline due to network instability or race conditions in asynchronous UI operations.
3. **Report Integration**: Ensuring that the Cypress-Mochawesome reporter generated reports correctly in the CI/CD environment, as path issues sometimes led to missing reports.

**Overcome methods**

**Environment Setup**: Used pre-built GitHub Actions (like actions/setup-node and cypress-io/github-action) for smoother dependency management.

1. **Test Stability**: Added retries and better wait commands for asynchronous operations to reduce test flakiness.
2. **Report Configuration**: Adjusted the Cypress configuration to ensure correct report paths, and verified report generation using local tests before CI integration.

**Future Aspects**

1. **Parallelization**: Running tests in parallel to reduce the time taken for test execution in CI.
2. **Cross-browser Testing**: Expanding test coverage to include different browsers like Firefox and Edge within the CI pipeline.
3. **Advanced Reporting**: Integrating dashboards like Cypress Dashboard for real-time test monitoring and in-depth analysis.

**Conclusion**

The Cypress test suite developed for the X-Cart demo application offers a comprehensive evaluation of the system's functionality, user interactions, and visual elements. By covering both functional testing (such as user workflows) and non-functional aspects (such as UI consistency and error handling), this test suite ensures that the application is both reliable and user-friendly.

* **Basic Tests** confirm the application's core functionality, ensuring that pages load correctly with the appropriate content and structure.
* **Functional Tests** simulate key user journeys, such as logging in, searching for products, and checking out, ensuring that the system performs these critical tasks without issues.
* **Edge Case Tests** validate how the application handles unexpected inputs, such as invalid form data or failed product searches, ensuring that the system remains robust and user-friendly even under non-ideal conditions.
* **UI and Visual Tests** ensure that the application maintains a consistent and polished user interface, with all essential elements (headers, footers, images) visible and correctly displayed.

Bottom of Form