Automation with Cypress

Contents

[**1. Introduction to Cypress** 2](#_Toc190424696)

[**2. Setup and Installation** 2](#_Toc190424697)

[**3. Cypress Test Structure** 2](#_Toc190424698)

[**4. Writing Basic Tests** 3](#_Toc190424699)

[**5. Assertions in Cypress** 3](#_Toc190424700)

[**6. Handling UI Elements** 4](#_Toc190424701)

[**7. Data-Driven Testing** 5](#_Toc190424702)

[**8. Custom Commands in Cypress** 8](#_Toc190424703)

[**9. Parallel Test Execution in Cypress** 9](#_Toc190424704)

[**10. Report Generation in Cypress** 10](#_Toc190424705)

[**11. Continuous Integration (CI) in Cypress** 11](#_Toc190424706)

**1. Introduction to Cypress**

Cypress is a JavaScript-based end-to-end testing framework that simplifies web application testing. Unlike Selenium, Cypress runs directly in the browser, allowing for real-time debugging and faster test execution. It is designed specifically for modern web applications and provides powerful features such as automatic waiting, time travel debugging, and network request interception.

**Key Features of Cypress:**

* **Fast and Reliable**: Runs in the browser, eliminating the need for external drivers.
* **Time Travel**: Allows users to see snapshots of commands as they run.
* **Automatic Waiting**: No need to add explicit waits or sleeps.
* **Spies, Stubs, and Mocks**: Controls behavior of functions and network requests.
* **Real-time Reloads**: Automatically reloads the test runner on changes.
* **Screenshots and Videos**: Captures test executions for debugging.

**2. Setup and Installation**

Cypress is easy to install and set up, requiring only Node.js.

**Prerequisites:**

* **Node.js** (LTS version recommended)
* **Code Editor** (VS Code recommended)
* **NPM or Yarn** (for package management)

**Installation Steps:**

1. **Initialize a Node.js project** (if not already set up):
2. npm init -y
3. **Install Cypress** as a development dependency:
4. npm install cypress --save-dev
5. **Open Cypress** using the command:
6. npx cypress open

This will launch the Cypress Test Runner, where you can run tests and explore the UI.

**3. Cypress Test Structure**

Cypress follows a structured directory system for organizing test cases, fixtures, and plugins.

**Default Cypress Directory Structure:**

cypress/

├── fixtures/ # Store test data (JSON files)

├── integration/ # Test cases (spec files)

├── plugins/ # Custom plugin configurations

├── support/ # Custom commands and reusable functions

**Key Files:**

* **cypress.json** – Cypress configuration file.
* **cypress/integration/** – Directory where test cases are stored.
* **cypress/fixtures/** – Stores reusable test data (e.g., user credentials).
* **cypress/plugins/index.js** – Allows custom plugin configuration.
* **cypress/support/commands.js** – Stores custom commands.

**4. Writing Basic Tests**

Cypress tests are written in JavaScript and follow a simple, readable syntax.

**Example test (sample.spec.js):**

/// <reference types="cypress" />

describe('My First Cypress Test', () => {

it('Visits a web page and verifies the title', () => {

cy.visit('https://example.com'); // Navigates to the URL

cy.contains('Example Domain').should('be.visible'); // Assertion

cy.title().should('eq', 'Example Domain'); // Verifies the page title

});

});

**Explanation:**

* describe() – Groups test cases.
* it() – Defines a test case.
* cy.visit() – Opens a webpage.
* cy.contains() – Checks if an element with specific text exists.
* cy.title() – Validates the page title.
* should() – Performs assertions.

This test visits a webpage, verifies its content, and ensures the page title is correct.

**5. Assertions in Cypress**

Assertions in Cypress are essential for validating expected outcomes in test cases. Cypress provides built-in assertions that work seamlessly with its chainable commands, making it easier to verify UI elements, URLs, and API responses.

**Types of Assertions:**

* **Implicit Assertions**: Cypress automatically retries commands until assertions pass or timeout occurs.
* **Explicit Assertions**: Using expect() or assert() from Chai assertion library.

**Common Assertions:**

**Element-Based Assertions:**

cy.get('h1').should('have.text', 'Example Domain'); // Check text content

cy.get('.btn').should('be.visible'); // Verify visibility

cy.get('#username').should('have.value', 'testuser'); // Verify input field value

**URL Assertions:**

cy.url().should('include', 'example.com'); // Verify partial URL

cy.location('pathname').should('eq', '/home'); // Check specific path

**API Response Assertions:**

cy.request('/api/data').its('status').should('eq', 200); // Check response status

cy.request('/api/user').its('body').should('have.property', 'name', 'John Doe'); // Validate response data

**6. Handling UI Elements**

Cypress provides a variety of commands to interact with different form elements, allowing for end-to-end user interactions to be tested effectively.

**Clicking Elements:**

cy.get('.btn-primary').click(); // Click button

cy.get('a').contains('Learn More').click(); // Click link with specific text

cy.get('#submit').dblclick(); // Double-click a button

cy.get('#context-menu').rightclick(); // Right-click on an element

**Typing in Input Fields:**

cy.get('#username').type('testuser'); // Enter text in input field

cy.get('#search').type('Cypress{enter}'); // Simulate pressing Enter key

cy.get('#comments').type('This is a multiline text{shift}{enter}New line'); // Multiline input

**Selecting Dropdowns:**

cy.get('select').select('Option1'); // Select by visible text

cy.get('select').select('2'); // Select by index

cy.get('select').select('option-value'); // Select by value

**Checking and Unchecking Checkboxes:**

cy.get('#rememberMe').check(); // Check a checkbox

cy.get('#subscribe').uncheck(); // Uncheck a checkbox

cy.get('input[type="checkbox"]').check(['option1', 'option2']); // Check multiple checkboxes

**Handling Radio Buttons:**

cy.get('input[type="radio"][value="male"]').check(); // Select radio button

cy.get('input[name="gender"]').check('female'); // Select by name attribute

**Working with Sliders:**

cy.get('input[type="range"]').invoke('val', 75).trigger('change'); // Set slider value

**Handling Date Pickers:**

cy.get('#datePicker').type('2025-12-31'); // Direct input

cy.get('#calendar').click();

cy.get('.day').contains('15').click(); // Select specific date

**Uploading Files:**

cy.get('#fileUpload').attachFile('example.pdf'); // Upload file using cypress-file-upload plugin

**Drag and Drop:**

cy.get('#draggable').drag('#droppable'); // Drag an element to a drop zone

**Handling Modals and Alerts:**

cy.get('.modal').should('be.visible'); // Verify modal visibility

cy.get('.modal-close').click(); // Close modal

cy.on('window:alert', (text) => {

expect(text).to.equal('This is an alert');

});

cy.on('window:confirm', () => true); // Accept confirmation dialog

**Handling Auto-Suggestions:**

cy.get('#searchBox').type('Cyp');

cy.get('.suggestions-list').contains('Cypress Testing').click(); // Select suggestion

**Handling Tooltips:**

cy.get('#tooltip').trigger('mouseover'); // Hover to show tooltip

cy.get('.tooltip-text').should('be.visible'); // Validate tooltip text

Cypress provides comprehensive support for handling UI elements, making it an ideal tool for testing interactive web applications.

**7. Data-Driven Testing**

Data-driven testing in Cypress allows running tests with multiple sets of data dynamically, ensuring greater test coverage and reducing code duplication. Cypress supports data-driven testing using **fixtures**, **environment variables**, and **external file sources (CSV, JSON, or Excel).**

**Using Fixtures**

Fixtures store external test data in JSON format, making it easy to reference test cases.

**Example cypress/fixtures/userData.json:**

{

"users": [

{ "username": "testuser1", "password": "password123" },

{ "username": "testuser2", "password": "password456" }

]

}

**Using Fixtures in Tests:**

cy.fixture('userData').then((data) => {

data.users.forEach((user) => {

cy.get('#username').type(user.username);

cy.get('#password').type(user.password);

cy.get('#login-btn').click();

});

});

**Using Environment Variables**

Cypress allows storing reusable values in environment variables through the cypress.json file.

**Define Variables in cypress.json:**

{

"env": {

"baseUrl": "https://example.com",

"username": "testuser",

"password": "testpass"

}

}

**Access Environment Variables in Tests:**

cy.visit(Cypress.env('baseUrl'));

cy.get('#username').type(Cypress.env('username'));

cy.get('#password').type(Cypress.env('password'));

cy.get('#login-btn').click();

**Using External Data Files (CSV, JSON, Excel)**

**Reading CSV Data:**

cy.readFile('cypress/fixtures/testData.csv').then((data) => {

const rows = data.split('\n');

rows.forEach((row) => {

const [username, password] = row.split(',');

cy.get('#username').type(username);

cy.get('#password').type(password);

cy.get('#login-btn').click();

});

});

**Reading JSON Data:**

cy.fixture('userData.json').then((data) => {

data.users.forEach((user) => {

cy.get('#username').type(user.username);

cy.get('#password').type(user.password);

cy.get('#login-btn').click();

});

});

**Reading Excel Data:**

Using the xlsx package, Cypress can read Excel files:

1. Install dependencies:
2. npm install xlsx
3. Read Excel data in Cypress:
4. import \* as XLSX from 'xlsx';
5. cy.readFile('cypress/fixtures/testData.xlsx', 'binary').then((content) => {
6. const workbook = XLSX.read(content, { type: 'binary' });
7. const sheet = workbook.Sheets[workbook.SheetNames[0]];
8. const data = XLSX.utils.sheet\_to\_json(sheet);
9. data.forEach((row) => {
10. cy.get('#username').type(row.username);
11. cy.get('#password').type(row.password);
12. cy.get('#login-btn').click();
13. });
14. });

**Using Cypress Each for Iterative Tests:**

const testData = [

{ username: 'user1', password: 'pass1' },

{ username: 'user2', password: 'pass2' }

];

describe('Data-Driven Test', () => {

testData.forEach((data) => {

it(`Tests login for ${data.username}`, () => {

cy.visit('/login');

cy.get('#username').type(data.username);

cy.get('#password').type(data.password);

cy.get('#login-btn').click();

cy.get('.dashboard').should('be.visible');

});

});

});

**Benefits of Data-Driven Testing in Cypress:**

* **Reduces code duplication** by allowing multiple test cases with different data sets.
* **Improves test coverage** by validating various input scenarios.
* **Enhances maintainability** by storing data externally, making tests flexible and reusable.
* **Supports various data formats** such as JSON, CSV, and Excel, making it versatile for different testing needs.

Cypress makes data-driven testing efficient with its ability to integrate with different file formats, improving the reliability of automation tests.

**8. Custom Commands in Cypress**

Cypress allows creating custom commands to simplify repetitive test steps and improve code reusability. These commands are defined in the cypress/support/commands.js file and can be used across multiple test cases.

**Creating a Custom Command**

To define a custom command, use the Cypress.Commands.add method.

**Example: Custom Login Command**

Cypress.Commands.add('login', (username, password) => {

cy.get('#username').type(username);

cy.get('#password').type(password);

cy.get('#login-btn').click();

});

**Using the Custom Command in Tests**

describe('Login Test', () => {

it('logs in successfully', () => {

cy.visit('/login');

cy.login('testuser', 'password123');

cy.get('.dashboard').should('be.visible');

});

});

**Custom Commands for API Requests**

Cypress can also define custom commands for API interactions.

Cypress.Commands.add('fetchUserData', (userId) => {

cy.request(`/api/users/${userId}`).then((response) => {

expect(response.status).to.eq(200);

});

});

**Benefits of Custom Commands:**

* **Reduces code duplication** by reusing common actions.
* **Improves readability** of test cases.
* **Encapsulates complex logic** in reusable functions.

**9. Parallel Test Execution in Cypress**

Parallel execution helps speed up test execution by distributing tests across multiple machines or processes.

**Running Tests in Parallel**

Cypress supports parallel execution using the --parallel flag when running tests with Cypress Cloud.

npx cypress run --record --parallel

To enable parallel execution, configure your cypress.config.js file:

module.exports = {

projectId: 'your-project-id',

e2e: {

setupNodeEvents(on, config) {

// Configure parallelization settings

},

},

};

**Parallel Execution with CI/CD**

Most CI/CD tools, such as GitHub Actions, Jenkins, and CircleCI, support parallel test execution.

**Example GitHub Actions Workflow:**

name: Cypress Tests

on: [push]

jobs:

cypress-run:

runs-on: ubuntu-latest

strategy:

matrix:

containers: [1, 2, 3] # Runs tests in 3 parallel containers

steps:

- name: Checkout code

uses: actions/checkout@v2

- name: Install dependencies

run: npm install

- name: Run Cypress tests in parallel

run: npx cypress run --record --parallel

**Benefits of Parallel Execution:**

* **Reduces test execution time** by distributing tests across multiple workers.
* **Improves efficiency** in large test suites.
* **Enhances scalability** when integrated with cloud testing solutions.

Cypress makes it easy to implement parallel test execution, optimizing automation workflows for faster feedback cycles.

**10. Report Generation in Cypress**

Cypress provides various ways to generate test reports, making it easier to analyze test execution results and integrate with CI/CD pipelines. Reports can be generated using built-in Cypress reporters or third-party plugins like **Mochawesome**.

**10.1 Built-in Reporters**

Cypress supports different built-in reporters such as:

* spec (default) – Displays test execution output in the console.
* json – Generates test results in JSON format.
* junit – Produces reports compatible with JUnit.

To use a built-in reporter, modify the Cypress run command:

npx cypress run --reporter junit

**10.2 Generating Reports with Mochawesome**

**Mochawesome** is a popular reporting tool for Cypress that generates interactive HTML reports.

**Steps to Set Up Mochawesome:**

1. Install Mochawesome dependencies:
2. npm install --save-dev mochawesome mochawesome-merge mochawesome-report-generator
3. Configure Cypress to use Mochawesome in cypress.config.js:
4. module.exports = {
5. reporter: 'mochawesome',
6. reporterOptions: {
7. reportDir: 'cypress/reports',
8. overwrite: false,
9. html: true,
10. json: true
11. }
12. };
13. Run tests with Mochawesome:
14. npx cypress run --reporter mochawesome
15. To merge multiple JSON reports and generate a consolidated HTML report:
16. npx mochawesome-merge cypress/reports/\*.json > cypress/reports/merged-report.json
17. npx marge cypress/reports/merged-report.json -o cypress/reports

**10.3 Integrating Reports with CI/CD**

Cypress test reports can be integrated into CI/CD tools like **Jenkins, GitHub Actions, CircleCI, and GitLab CI/CD**.

**Example: Generating Reports in GitHub Actions**

name: Cypress Tests

on: [push]

jobs:

cypress-run:

runs-on: ubuntu-latest

steps:

- name: Checkout code

uses: actions/checkout@v2

- name: Install dependencies

run: npm install

- name: Run Cypress with Mochawesome

run: npx cypress run --reporter mochawesome

- name: Upload Test Report

uses: actions/upload-artifact@v2

with:

name: Cypress-Test-Report

path: cypress/reports/

**10.4 Viewing Reports**

* After running tests locally, reports will be available in the cypress/reports/ directory.
* For CI/CD, reports can be uploaded as artifacts and viewed post-execution.

**10.5 Benefits of Report Generation:**

* **Better Visibility** – Provides structured and readable reports.
* **Historical Tracking** – Stores results for trend analysis.
* **CI/CD Integration** – Enables automation pipelines to include test results.
* **Debugging** – Simplifies root cause analysis for test failures.

Cypress's reporting capabilities make it an excellent tool for both local and cloud-based test execution, ensuring robust test result analysis and tracking.

**11. Continuous Integration (CI) in Cypress**

Continuous Integration (CI) is the practice of automating the integration of code changes from multiple contributors into a shared repository. In the context of Cypress, CI involves running Cypress tests as part of your development pipeline whenever code is pushed to your version control system (such as GitHub). This ensures that tests are executed frequently, enabling faster feedback and helping catch issues early.

Let’s break it down:

**1. Continuous Integration (CI) in Cypress**

Cypress is a popular testing framework for end-to-end (E2E) testing, and CI tools like Jenkins and GitHub Actions are often used to automate the execution of tests on every commit or pull request.

In a typical CI pipeline for Cypress:

* **Cypress Tests** are executed on each change (e.g., commits, merges) to verify that the code doesn’t break the app.
* A **CI server** is used to automate running tests in the background without manual intervention.
* The **Cypress Dashboard** can be used to monitor test results, track trends, and get insights into failing tests.

**2. Integration of CI in Jenkins**

Jenkins is a widely used CI/CD tool that can be configured to run Cypress tests automatically whenever changes are pushed to the repository. Here's how you can integrate Cypress into Jenkins:

**Prerequisites:**

* Jenkins installed and configured.
* Node.js and npm installed on the Jenkins server.
* Cypress installed in the repository where your tests reside.

**Steps to Integrate Cypress with Jenkins:**

1. **Create a Jenkins Pipeline**:
   * Open Jenkins and create a new pipeline job (Freestyle Project or Pipeline).
   * In the configuration page, you will need to define the repository that contains your Cypress tests.
2. **Configure Node.js and Cypress**:
   * In the pipeline script, ensure that the environment has Node.js and npm installed. You can use Jenkins’ built-in tools or configure them with a tool like **NodeJS Plugin**.
   * Install Cypress in your repository (if you haven’t already) by adding it as a dependency:
   * npm install cypress --save-dev
3. **Set up the Build Script**: In the pipeline’s build section, define a script that installs dependencies, runs tests, and reports results. Example:
4. npm install # Install dependencies
5. npx cypress run --record --key <CYPRESS-DASHBOARD-KEY> # Run tests in headless mode and record results on Cypress Dashboard
6. **Add Test Reports**: Jenkins can display test results directly. After the tests run, you can configure the pipeline to capture test reports generated by Cypress:
7. npx cypress run --reporter mochawesome

Then, configure Jenkins to parse the **mochawesome** report:

* + Use the **Mochawesome Plugin** in Jenkins to visualize results.

1. **Triggering Builds**: Configure your pipeline to trigger the tests automatically:
   * **Webhook triggers** for GitHub pushes.
   * **Scheduled triggers** for regular tests.
2. **Monitor the Tests**: Jenkins will now run Cypress tests and display the results in the build history, making it easy to track which changes caused issues.

**Jenkins Pipeline Example:**

pipeline {

agent any

tools {

nodejs 'NodeJS'

}

stages {

stage('Install Dependencies') {

steps {

sh 'npm install'

}

}

stage('Run Cypress Tests') {

steps {

sh 'npx cypress run'

}

}

stage('Publish Test Results') {

steps {

publishHTML(target: [

reportDir: 'cypress/results',

reportFiles: 'index.html',

reportName: 'Cypress Test Report'

])

}

}

}

}

**3. Integration of CI in GitHub Actions**

GitHub Actions is a CI/CD tool that can automate your workflows directly from your GitHub repository. It allows you to set up workflows that are triggered by events like pushes, pull requests, or scheduled intervals.

**Steps to Integrate Cypress with GitHub Actions:**

1. **Create a Workflow Configuration File**: GitHub Actions workflows are defined in .github/workflows directory. Create a new file (e.g., cypress.yml).
2. **Define the Workflow**: The workflow will consist of jobs like setting up the environment, installing dependencies, and running Cypress tests.

Example workflow configuration:

name: Run Cypress Tests

on:

push:

branches:

- main

pull\_request:

branches:

- main

jobs:

cypress-run:

runs-on: ubuntu-latest

steps:

- name: Checkout code

uses: actions/checkout@v2

- name: Set up Node.js

uses: actions/setup-node@v2

with:

node-version: '14'

- name: Install dependencies

run: |

npm install

- name: Run Cypress Tests

run: |

npx cypress run --record --key ${{ secrets.CYPRESS\_KEY }}

- name: Upload Cypress Screenshots (optional)

uses: actions/upload-artifact@v2

with:

name: cypress-screenshots

path: cypress/screenshots/

1. **Secrets**: To avoid exposing sensitive data, store the Cypress Dashboard record key in GitHub secrets:
   * Go to the GitHub repository settings.
   * Under the **Secrets** section, add a secret for CYPRESS\_KEY.
2. **Triggering the Workflow**: The workflow is triggered on pushes to the main branch or on pull requests targeting main. You can adjust this to suit your needs.
3. **Monitoring and Reporting**: Once the workflow runs, GitHub Actions will display the results in the **Actions** tab. You can also configure Cypress to record the test results on the **Cypress Dashboard** for detailed reporting.

**Example Output on GitHub Actions:**

* View the Cypress results directly in GitHub Actions.
* View screenshots or video recordings of the failed tests (if configured).

**Summary of Differences:**

* **Jenkins** requires setting up a server, managing plugins, and configuring pipelines, which offers more customization but requires additional setup and maintenance.
* **GitHub Actions** is more integrated with GitHub repositories and is easier to set up for simple workflows. It's a good choice if you're already using GitHub and don't want to manage additional infrastructure.

**Conclusion:**

Both Jenkins and GitHub Actions can integrate Cypress for Continuous Integration. Jenkins provides more flexibility and is suited for complex enterprise environments, while GitHub Actions is simpler to set up and offers tight integration with GitHub repositories.