Front-end Testing Framework: A Practical Guide

Bc. Dalibor Králik

This book has been created as a part of master thesis at Masaryk University of Brno. I want to express my gratitude to my supervisor, Prof. RNDr. Tomáš Pitner, Ph.D., for providing valuable feedback and advices for writing this book.

Contents

Chapter 1

Introduction

Modern front-end development is fast, dynamic, and complex. With the rapid evolution of frameworks like React and Next.js, developers face the challenge of delivering robust, high-quality applications while keeping pace with changing user expectations. Yet, testing remains a critical, often under-prioritized element in this process—one that directly impacts application reliability, performance, and user satisfaction.

This guide presents a comprehensive, actionable framework for front-end testing that spans the entire spectrum of testing techniques. Based on the research and practical experiences documented in the master thesis "Development of a Front-End Testing Framework: Strategies, Tools, and Implementation" at Masaryk University Faculty of Informatics, this PDF provides clear, step-by-step instructions and best practices for:

- Unit Testing
- Integration Testing
- End-to-End (E2E) Testing
- Visual Regression Testing
- Performance Testing
- Security Testing
- Cross-Browser Testing

Each testing phase is supported by a pre-configured Next.js template, which integrates popular tools such as Vitest, React Testing Library, Cypress, Lighthouse, and axe-core. This template serves as a practical starting point for developers, allowing them to implement and extend the testing framework in their own projects.

By following the guidelines in this document, front-end teams can:

- Catch defects early and reduce technical debt.
- Ensure consistency and reliability across various user scenarios.
- Improve overall code quality while streamlining development workflows.

Ultimately, this guide aims to bridge the gap between academic research and practical application, empowering developers to adopt a robust, multi-layered testing strategy that meets the demands of modern web applications.

Chapter 2

The Next.js Template for Front-End Testing

The Next.js template presented in this guide is a comprehensive, ready-to-use starting point for implementing a robust front-end testing framework. Developed as part of the master thesis "Development of a Front-End Testing Framework: Strategies, Tools, and Implementation" at Masaryk University Faculty of Informatics, this template integrates multiple testing environments into a single Next.js application. By leveraging modern tools and best practices, the template enables teams to implement unit tests, integration tests, end-to-end (E2E) tests, visual regression tests, performance tests, accessibility tests, security checks, internationalization tests, and cross-browser tests—all within one cohesive project structure.

2.1 Key Features

- Multi-Modal Testing Support:

  The template is pre-configured with environments for:

  - Unit Testing using Vitest and React Testing Library.
  - Integration Testing to verify interactions between components and services.
  - End-to-End Testing with Cypress for simulating real user scenarios.
  - Visual Regression Testing via Cypress with cypress-visual-regression.
  - Accessibility Testing with Cypress-axe and axe-core.
  - Performance Testing using Lighthouse CLI.
  - Security Testing with ESLint (including eslint-plugin-security) and external tools like OWASP ZAP.
  - Internationalization (i18n) Testing using next-intl and corresponding test wrappers.
  - Cross-Browser Testing with Cypress running tests across Chrome, Firefox, Edge, and Electron.

- Modern Tech Stack:

  The template is built with:

- Next.js 14 for the latest server-side rendering and app routing capabilities.
- Tailwind CSS 4 for utility-first styling.
- Typescript 5 for type safety and maintainability.
- Axios for data fetching.
- ESLint 9 to enforce code quality and security best practices.

2.2 How to use this Template

1. Access the template: Next.js Template is uploaded on the Github and also on Gitlab. On both platform there are same versions of the Next.js Template.
   - Github: Go to this Github repository (https://github.com/MrDalo/frontend-testing-framework) and on the top right corner click on the green button *"Use this template"* and then click on *"Create a new repository"*. It will create a new repository with the clean commit history.
   - Gitlab: Go to this Gitlab repository (https://gitlab.com/MrDalo/frontend-testing-framework) and on the top right corner click on the button *"Fork"* and create new forked repository.
2. Clone your repository locally.
3. Open README.md and follow the section *"How to Use This Template"*

2.3 Benefits of the Template

- Rapid Onboarding: Developers can quickly start writing tests using the pre-configured environment, reducing initial setup time.
- Consistent Best Practices: By following the structured test types and best practices outlined in this guide, teams can ensure consistent, reliable testing across the project.
- Scalability: The template's modular structure allows teams to easily extend or modify testing strategies as project requirements evolve.
- CI/CD Integration: The included GitHub Actions pipeline and Gitlab pipeline example shows how to integrate this template into a continuous integration workflow, ensuring that tests are automatically run on every commit or pull request.

Chapter 3

Testing Phases

This chapter provides a detailed guide on the different phases of testing in front-end development. For each testing phase, we outline what to test, the recommended tools and frameworks, and practical guidelines for writing and structuring tests. By following these recommendations, development teams can ensure thorough coverage and reliable outcomes.

3.1 Unit Testing

Unit testing forms the foundational layer of the testing pyramid. In this phase, the smallest units of code—such as functions, components, and custom hooks—are validated in isolation to ensure they work correctly under a variety of conditions.

Background

Unit tests verify individual code units without relying on external systems or dependencies. They provide rapid feedback during development and catch defects early in the development cycle. By ensuring that each unit behaves as expected, unit tests contribute to a more reliable and maintainable codebase.

What to Test

- Utility functions and helper methods.
- Individual React components.
- State management actions (e.g., Zustand store actions, selectors) and TanStack Query hooks.
- Custom hooks.

Tools:

Vitest, React Testing Library.

Common Pitfalls

- **Over-Mocking:** Excessive or improper use of mocks can hide integration issues and create tests that pass without reflecting real-world behavior.
- **Poor Test Isolation:** If tests are not isolated, a failure in one unit can cascade, making it difficult to identify the source of the problem.
- **Neglecting Edge Cases:** Focusing only on the "happy path" may leave critical error conditions untested.
- **Ambiguous Test Names:** Vague or generic test names reduce clarity and make debugging harder.

Test Structure Guidelines

- Use descriptive test names that clearly state what is being verified (e.g., *renders with negative count*, *should return 500 for AxiosError without response*)
- Follow the Arrange-Act-Assert pattern for clarity [1]
  - **Arrange:** Set up the test environment and prepare the inputs.
  - **Act:** Execute the code under test.
  - **Assert:** Verify that the output matches expectations.
- Isolate dependencies using mocks (e.g., `vi.mock()`) to focus tests on the unit under test.
- Handle edge cases and error conditions.
- Use setup and teardown methods (`beforeEach`/`afterEach`) to prepare test environments.

Code Examples

Below is a simple example of a unit test for a utility function using Vitest and React Testing Library:

```
/* utility.ts */
export const add = (a, b): string => a + b;

/* utility.unit.test.js */
import { add } from './utility';

describe('add function', () => {
  test('should return the sum of two positive numbers', () => {
    // Arrange
    const a = 2, b = 3;
    // Act
    const result = add(a, b);
    // Assert
```

```
    expect(result).toBe(5);
  });
});
```

Listing 3.1: Example Unit Test for a Utility Function

3.2 Integration Testing

Integration testing ensures that individual units work together as intended. This phase bridges the gap between isolated unit tests and full end-to-end tests by simulating real interactions among components and external services.

Background

Integration tests focus on verifying the interaction between multiple modules or components rather than testing them in isolation. These tests help identify issues that arise when units are combined, such as data flow problems or mismatched interfaces. By simulating realistic scenarios, integration testing provides a deeper level of assurance that the application behaves correctly when its parts are assembled.

What to Test

- **Component Interactions:** Verify that parent and child components communicate correctly (e.g., proper prop passing and state updates).
- **Data Flow:** Ensure that asynchronous data fetching (e.g., via React Query) integrates seamlessly with UI components.
- **Complex State Management:** Test scenarios where multiple components share and update state (e.g., using Zustand or Redux).
- **Authentication Flows:** Confirm that authentication and authorization mechanisms (e.g., NextAuth) operate as expected.
- **Server Actions and API Integrations:** Validate that external API calls and server actions are correctly integrated, often by using mocks to simulate external systems.

Tools

- Vitest
- React Testing Library

Common Pitfalls

- **Insufficient Isolation:** Overlapping test scenarios can make it difficult to pinpoint the source of failures.
- **Complex Data Setup:** Using overly complicated test data may lead to brittle tests that are hard to maintain.
- **Over-Mocking:** Excessively mocking external dependencies may hide real integration issues.

Test Structure Guidelines

- **Realistic Data Flow:** Use sample data that closely mirrors production scenarios to ensure accurate test outcomes.
- **Modular Organization:** Group tests by functionality so that similar tests are kept together.
- **Arrange-Act-Assert:** Structure each test into clear phases:
  - **Arrange:** Set up the components and state.
  - **Act:** Execute the interactions or data fetching.
  - **Assert:** Verify that the combined behavior meets expectations.
- **Setup/Teardown:** Use `beforeEach` and `afterEach` to establish a clean test environment.

Code Example

Below is an example integration test that verifies proper interaction between a parent component and its child component using Vitest and React Testing Library:

```
/* ParentComponent.integration.test.js */
import React from 'react';
import { render, screen } from '@testing-library/react';
import ParentComponent from './ParentComponent';

describe('ParentComponent integration', () => {
  test('passes the correct props to ChildComponent', () => {
    // Arrange: Render the ParentComponent
    render(<ParentComponent />);

    // Act: Find the element rendered by ChildComponent
    const childElement = screen.getByTestId('child-element');

    // Assert: Verify that ChildComponent received the expected prop value
    expect(childElement).toHaveTextContent('Expected Prop Value');
  });
});
```

Listing 3.2: Example Integration Test for Component Interaction

3.3 End-to-End (E2E) Testing

End-to-End testing simulates complete user workflows from the user interface down to the backend services. It is the final validation step that ensures all parts of the system work together as expected under real-world conditions.

Background

E2E tests focus on validating entire workflows rather than isolated components. By mimicking real user interactions—such as logging in, navigating between pages, and submitting forms—these tests reveal integration issues that unit or integration tests might miss. This holistic approach is essential for ensuring that the application delivers a seamless user experience.

What to Test

- **Critical User Workflows:** Key user actions such as login, registration, checkout, or data submission.
- **Navigation Flows:** Multi-page interactions and proper routing.
- **System Integration:** End-to-end interaction between the front-end and backend services.

Tools

- Cypress is the primary tool for E2E testing due to its robust API for simulating user interactions and its ability to run tests in a real browser environment.

Common Pitfalls

- **Flaky Tests:** Asynchronous operations and timing issues can lead to intermittent failures.
- **State Persistence:** Not resetting application state between tests may cause tests to interfere with each other.
- **Environmental Dependencies:** Relying on live production data or services without proper stubbing can result in unreliable tests.

Test Structure Guidelines

- **Arrange:** Prepare the application state, seed databases if necessary, and ensure the test environment is clean.
- **Act:** Use Cypress commands (e.g., `cy.visit()`, `cy.get()`, `cy.click()`) to simulate user interactions.

- **Assert:** Verify that the outcomes (e.g., page navigation, UI updates, API responses) meet expectations.
- **Documentation:** Include comments and clear step descriptions to make tests self-explanatory.

Code Example

Below is a sample E2E test that simulates a simple login workflow:

```
describe('Login Workflow', () => {
  beforeEach(() => {
    // Arrange: Reset state and navigate to the login page
    cy.visit('/login');
  });

  it('should log in successfully with valid credentials', () => {
    // Act: Fill in the login form and submit
    cy.get('input[name="username"]').type('testuser');
    cy.get('input[name="password"]').type('securepassword');
    cy.get('button[type="submit"]').click();

    // Assert: Verify that the dashboard is displayed
    cy.url().should('include', '/dashboard');
    cy.contains('Welcome, testuser').should('be.visible');
  });
});
```

Listing 3.3: Example E2E Test for a Login Workflow

Tool Tips

- **Simulate Real User Behavior:** Use realistic interaction patterns such as delays, scrolling, and conditional waits to mimic actual user actions.
- **Timeouts and Retries:** Configure custom timeouts and automatic retries to manage slow-loading pages or intermittent network issues.
- **Environment Setup:** Utilize seed data or dedicated test environments to ensure consistent test conditions.
- **CI/CD Integration:** Incorporate E2E tests in your continuous integration pipeline to catch regressions on every commit.

3.4 Visual Regression Testing

Visual regression testing aims to detect unintended changes in the user interface by comparing current screenshots against established baselines. This phase is critical for ensuring that UI updates do not introduce design inconsistencies or break responsive layouts.

Background

Visual regression testing provides an automated way to validate that the user interface remains consistent over time. By capturing screenshots during a baseline run and comparing them with subsequent tests, teams can quickly identify discrepancies caused by code changes, CSS modifications, or rendering differences across devices. This approach is especially useful in dynamic front-end applications where visual appearance is as important as functionality.

What to Test

- **UI Layout:** Verify that the structure and positioning of components remain consistent.
- **Component Appearance:** Ensure that styling and theming are rendered as expected.
- **Responsive Design:** Confirm that the interface adapts correctly across different screen sizes and devices.

Tools

- Cypress with the cypress-visual-regression plugin.

Common Pitfalls

- **False Positives:** Minor rendering differences (e.g., antialiasing) can trigger false alarms.
- **Inconsistent Environments:** Variations in browser settings or device resolutions may lead to inconsistent baseline captures.
- **Overly Sensitive Tolerances:** Setting pixel difference thresholds too low may cause tests to fail for insignificant changes.

Test Structure Guidelines

- **Establish a Baseline:** Capture baseline screenshots under controlled conditions. Update the baselines intentionally when UI changes are approved.
- **Compare with a Baseline images:** Compare subsequent screenshots against the baseline to detect discrepancies.
- **Define Tolerances:** Configure acceptable pixel difference thresholds to filter out negligible discrepancies.
- **Consistent Environment:** Ensure that tests run in a consistent environment, using the same browser version and settings for baseline and regression runs.

- **Automate Updates:** Integrate visual regression testing in your CI/CD pipeline so that any unexpected changes are flagged immediately.

Code Example

Below is an example Cypress test that captures and compares a screenshot for visual regression:

```
describe('Homepage Visual Regression', () => {
  beforeEach(() => {
    cy.visit('/'); // Navigate to the homepage
    cy.injectAxe(); // (Optional) Inject accessibility checks if needed
  });

  it('should match the baseline screenshot', () => {
    // Capture a screenshot and compare it with the baseline image.
    // If no baseline exists, it will be created.
    cy.compareSnapshot('homepage');
  });
});
```

Listing 3.4: Example Visual Regression Test

3.5 Performance Testing

Performance testing measures the responsiveness and stability of an application under varying levels of load. This phase is essential for ensuring that users experience fast and reliable performance even under high traffic conditions.

Background

Performance testing helps to uncover bottlenecks and inefficiencies in an application by simulating realistic usage scenarios. By measuring metrics such as page load times, time to interactive, and core web vitals (e.g., Largest Contentful Paint, First Input Delay, and Cumulative Layout Shift), teams can make data-driven decisions to optimize performance. Moreover, regularly running performance tests ensures that new code changes do not introduce regressions that degrade the user experience.

What to Test

- **Page Load Times:** Measure how long it takes for pages to load completely.
- **Time to Interactive:** Evaluate when the application becomes usable for the user.
- **Core Web Vitals:** Focus on key metrics such as Largest Contentful Paint (LCP), First Input Delay (FID), and Cumulative Layout Shift (CLS).
- **API Response Times:** Simulate stress to test the response times of backend services.

Tools

- Lighthouse CLI for performance audits.
- Optionally, WebPageTest or Sitespeed.io for more advanced scenarios.

Common Pitfalls

- **Single Iteration Testing:** Relying on one test run can lead to misleading results due to network or server variability.
- **Unrealistic Test Environments:** Testing in non-representative environments may produce results that do not reflect real user conditions.
- **Overly Strict Budgets:** Setting performance thresholds too low might cause false positives.

Best Practices

- **Run Multiple Iterations:** Execute tests several times and average the results for more reliable data.
- **Establish Performance Budgets:** Define acceptable thresholds for key performance metrics to flag regressions automatically.
- **Generate Detailed Reports:** Produce reports in both human-readable (HTML) and machine-readable (JSON) formats to aid in analysis and integration with CI/CD pipelines.
- **Consistent Environment:** Run tests under consistent conditions (same browser version, device simulation, network conditions) to ensure comparability.

Code Example

Below is an example command that uses Lighthouse CLI to run a performance test against a local instance of your application. The command generates both JSON and HTML reports and opens the report for visual inspection:

```
lighthouse http://localhost:3000 --chrome-flags='--no-sandbox␣--headless=new'
    --output json --output html --view --output-path ./src/tests/performance-
    tests/lighthouse-report
```

Listing 3.5: Example Lighthouse Command

3.6 Accessibility Testing

Accessibility testing ensures that the application is usable by people with disabilities, including those who rely on screen readers, keyboard navigation, and other assistive technologies. This testing phase validates that the user interface complies with established accessibility standards (e.g., WCAG) and is inclusive to all users.

Background

Accessibility testing goes beyond verifying that elements exist on the page. It involves ensuring that the semantic structure, ARIA attributes, and keyboard navigability allow users with disabilities to effectively interact with the application. Early detection of accessibility issues not only helps avoid legal risks but also improves the overall user experience.

What to Test

- **Semantic HTML and Landmarks:** Verify that the application uses proper HTML elements (e.g., `<main>`, `<nav>`, `<header>`) and that landmarks are present.
- **Keyboard Navigation:** Ensure that all interactive elements are accessible via keyboard, with logical focus order and visible focus indicators.
- **Color Contrast and ARIA Attributes:** Check that text has sufficient contrast against its background and that ARIA attributes are correctly implemented.
- **Alternative Texts:** Confirm that images and non-text content include appropriate `alt` attributes.
- **Screen Reader Compatibility:** Validate that the application's content is accessible to screen readers.

Tools

- Cypress with cypress-axe
- axe-core
- Lighthouse (for additional accessibility audits)

Common Pitfalls

- **Incomplete Landmark Coverage:** Missing or improperly structured landmarks can hinder navigation for assistive technologies.
- **Insufficient Focus Management:** Without clear focus indicators or logical focus order, keyboard users may become lost.
- **Ignoring Dynamic Content:** Applications that update content dynamically without notifying screen readers may cause confusion.
- **Over-Reliance on Automated Tools:** Automated tests may miss context-specific issues, so manual reviews remain important.

Test Structure Guidelines

- **Inject Axe-core:** Always inject the axe-core library after navigating to a page using `cy.injectAxe()`.
- **Run Accessibility Checks:** Use `cy.checkA11y()` to run tests on the entire page or on specific sections.
- **Customize Rules:** Adjust the axe-core configuration to disable known false positives or to enforce project-specific requirements.
- **Document Findings:** Clearly log and document any accessibility violations for further review and remediation.

Code Example

Below is an example Cypress test that uses cypress-axe to validate accessibility:

```
describe('Homepage Accessibility', () => {
  beforeEach(() => {
    // Navigate to the homepage and inject the axe-core library
    cy.visit('/');
    cy.injectAxe();
  });

  it('should have no accessibility violations', () => {
    // Run accessibility checks on the entire page
    cy.checkA11y(null, null, (violations) => {
      // Log detailed information for each violation
      cy.task('log', `${violations.length} accessibility violation(s) detected
          `);
      violations.forEach((violation) => {
        cy.log(`${violation.id}: ${violation.help}`);
      });
    });
  });
});
```

Listing 3.6: Example Accessibility Test

3.7 Security Testing

Security testing focuses on identifying and mitigating vulnerabilities in client-side code that could be exploited by malicious actors. This phase combines static and dynamic analysis to ensure that the application is resilient against common threats such as cross-site scripting (XSS), cross-site request forgery (CSRF), and insecure data handling.

Background

Security testing involves analyzing both the code and the running application. Static analysis tools scan your code for insecure patterns and potential vulnerabilities, while dynamic scanning tools simulate real-world attacks on your application. By integrating these approaches, teams can catch issues early in development and prevent costly security breaches in production.

What to Test

- **Vulnerabilities:** Identify potentially dangerous practices such as the use of `eval()`, unsafe regular expressions, and object injection.
- **Data Handling:** Verify that all user inputs are properly sanitized and that sensitive data is securely processed and stored.
- **HTTP Security Headers:** Ensure that critical security headers (e.g., Content Security Policy, X-Frame-Options) are correctly configured.

Tools

- ESLint with eslint-plugin-security for static code analysis.
- OWASP ZAP for dynamic vulnerability scanning.

Common Pitfalls

- **Overreliance on Automation:** Automated tools may miss context-specific issues, so it is important to complement them with manual code reviews.
- **Incomplete Coverage:** Focusing solely on client-side security can leave backend vulnerabilities unaddressed.
- **Neglecting Configuration:** Default or insecure configurations (such as missing security headers) are often overlooked.

Best Practices

- **Integrate Static Analysis:** Run ESLint with security plugins as part of your CI/CD pipeline to catch vulnerabilities during development.
- **Complement with Dynamic Scanning:** Use tools like OWASP ZAP to simulate real attacks and validate your security measures.
- **Regular Code Reviews:** Combine automated security checks with manual reviews for a more comprehensive assessment.
- **Keep Tools Updated:** Regularly update security tools, libraries, and configurations to address evolving threats.

3.8 Internationalization (i18n) Testing

Internationalization testing ensures that the application supports multiple languages and cultural formats. It verifies that locale switching works correctly and that translated strings, dates, currencies, and numbers are rendered appropriately for each locale.

Background

Internationalization (i18n) is critical for reaching a global audience. Beyond merely translating text, i18n requires adapting layouts, date and number formats, and even text direction (e.g., for right-to-left languages) to meet diverse cultural expectations. This phase helps ensure that all users experience a coherent and accessible interface regardless of their language or region.

What to Test

- **Locale Switching Logic:** Verify that the application can seamlessly switch between different languages.
- **Translation Accuracy:** Ensure that components display the correct translated strings.
- **Formatting Consistency:** Check that dates, currencies, and numbers adhere to locale-specific conventions.
- **Layout Adaptability:** Confirm that changes in text length or direction do not break the user interface.

Tools

- Vitest with React Testing Library for unit and integration tests.
- Cypress for end-to-end (E2E) tests.

Common Pitfalls

- **Missing Translation Keys:** Overlooked or absent translation keys may lead to fallback text or errors.
- **Layout Issues:** Translated strings may be longer or shorter than the original text, potentially disrupting the layout.
- **Hardcoded Strings:** Embedding literal text within components can hinder the flexibility required for i18n.
- **Inconsistent Formatting:** Dates, currencies, and numbers might not format correctly if locale data is not applied uniformly.

Best Practices

- **Write Isolated Tests:** Create unit tests that verify component rendering under different locales.
- **Implement E2E Tests:** Simulate complete language-switching workflows to ensure that the entire application adapts to the selected locale.
- **Maintain Translation Files:** Keep your translation files updated and under version control.
- **Simulate Real Environments:** Use environment variables or configuration settings in tests to mimic various locales.
- **CI/CD Integration:** Incorporate i18n tests into your CI/CD pipeline to ensure continuous verification as translations evolve.
- **Consistent Data:** Use standardized sample data across tests to avoid discrepancies in date or currency formatting.

Code Example

Below is an example unit test using Vitest and React Testing Library to verify that a component renders correctly for different locales:

```
import { render, screen } from '@testing-library/react';
import MyComponent from './MyComponent';
import { NextIntlWrapper } from '../NextIntlWrapper';

describe('MyComponent i18n', () => {
  test('renders correct text for English locale', () => {
    render(
      <NextIntlWrapper locale="en">
        <MyComponent />
      </NextIntlWrapper>
    );
    expect(screen.getByText('Hello')).toBeInTheDocument();
  });

  test('renders correct text for German locale', () => {
    render(
      <NextIntlWrapper locale="de">
        <MyComponent />
      </NextIntlWrapper>
    );
    expect(screen.getByText('Hallo')).toBeInTheDocument();
  });
});
```

Listing 3.7: Example i18n Test for a Component using custom NextIntlWrapper which is part of Next.js Template

3.9 Cross-Browser Testing

Cross-browser testing ensures that the application behaves consistently across different browsers and devices. This testing phase is critical to verify that UI elements, functionality, and user interactions remain stable regardless of the browser's rendering engine or platform.

Background

Modern web applications must support a diverse range of browsers and devices. Differences in browser engines (e.g., Chromium, Gecko, EdgeHTML) can lead to variations in how content is rendered and how interactions are handled. Cross-browser testing simulates these conditions to uncover issues such as layout inconsistencies, functional discrepancies, and performance variations. By validating the application across multiple browsers, teams can ensure a consistent and accessible user experience for all users.

What to Test

- **UI Consistency:** Verify that layouts, fonts, colors, and component placements are rendered consistently in browsers such as Chrome, Firefox, Edge, and Electron.
- **Functional Behavior:** Ensure that interactive elements (e.g., forms, buttons, navigation) function as expected across different browser environments.
- **Responsiveness:** Test that the application adapts appropriately to various screen sizes and resolutions.
- **Performance Variations:** Identify any differences in performance, such as page load times or responsiveness, that might arise due to browser-specific optimizations.

Tools

- Cypress is primarily used for cross-browser testing by leveraging the `--browser` flag or a CI matrix to run tests across multiple browsers.
- Additional tools such as BrowserStack or LambdaTest can be integrated for testing on real devices and a broader range of browsers.

Common Pitfalls

- **Inconsistent Defaults:** Variations in default browser settings (e.g., fonts, zoom levels) can lead to unexpected rendering differences.
- **Environmental Variability:** Testing in different environments without standardized configurations may result in inconsistent outcomes.
- **Overlooking Minor Differences:** Minor pixel-level discrepancies may not affect usability but can cause automated tests to fail if tolerances are not properly configured.

Test Structure Guidelines

- **Unified Test Suite:** Use the same E2E test suite across different browsers to ensure comprehensive coverage.
- **CI Matrix Configuration:** Leverage your CI/CD pipeline to run tests in parallel across multiple browsers.
- **Standardize Environments:** Configure consistent browser settings (e.g., viewport size, zoom level) across test runs.
- **Custom Tolerances:** Define acceptable pixel differences in visual regression tests to account for minor rendering variations.
- **Device Simulation:** Incorporate responsive design tests by simulating various device sizes and orientations.

Code Example

Below is an example of a command to run Cypress tests in Chrome using the `--browser` flag. This command can be integrated into your CI/CD pipeline or run locally:

```
npx cypress run --config-file cypress.config.ts --browser chrome --spec 'src/
    tests/e2e/**/*.e2e.cy.{ts,tsx}'
```

Listing 3.8: Example Command for Cross-Browser E2E Testing

3.10 Conclusion

Each phase of testing plays a critical role in ensuring the overall quality of modern front-end applications. By following these guidelines and structuring tests according to the phases outlined above, development teams can achieve comprehensive test coverage that mitigates risks, improves reliability, and ensures a superior user experience. This systematic approach enables teams to catch defects early, optimize performance, and maintain a high level of code quality, thereby reducing technical debt and improving project outcomes.

Chapter 4

Folder Structure & Commands

This chapter outlines an optimal folder structure for front-end projects with al ltesting environments—particularly for those built with React and Next.js. The structure described below is employed in the Next.js template developed for this guide, and it reflects industry best practices for organizing application and test code. A well-structured project facilitates collaboration, scalability, and maintainability, while enabling efficient testing across multiple levels.

4.1 Optimal folder structure

An ideal front-end project structure separates application code from tests and assets. The following layout is recommended:

```
/
├── cypress/
│   ├── support/
│   │   └── e2e.ts/
├── src/
│   ├── app/
│   │   ├── layout.tsx/
│   │   └── .../
│   ├── components/
│   │   ├── SomeComponent/
│   │   │   ├── SomeComponent.tsx/
│   │   │   └── __tests__/
│   │   │       ├── SomeComponent.unit.test.tsx/
│   │   │       ├── SomeComponent.integration.test.tsx/
│   │   │       └── SomeComponent.i18n.test.tsx/
│   ├── tests/
│   │   ├── e2e/
│   │   │   ├── login.e2e.cy.ts/
│   │   │   └── dashboard.e2e.cy.ts/
│   │   ├── visual-tests/
│   │   │   └── dashboard.visual.cy.ts/
│   │   ├── performance-tests/
│   │   │   ├── lighthouse-report.report.json/
│   │   │   └── lighthouse-report.report.html/
│   │   ├── accessibility-tests/
│   │   │   └── profile-page.accessibility.cy.tsx/
│   │   ├── i18n-tests/
│   │   │   └── profile-data.i18n.cy.tsx/
│   │   ├── cross-browser-tests/
│   │   │   └── profile-page.cross-browser.cy.ts/
├── vitest.config.ts/
├── vitest.setup.ts/
├── cypress.config.ts/
├── package.json/
├── README.md/
└── .../
```

- **cypress/**
  Contains configuration and support files for Cypress, which is used for end-to-end, visual regression, and accessibility testing.
  - `support/`: Global setup and custom commands (e.g., `e2e.ts`).
- **src/**
  Houses all the source code and tests for the application.

  - **app/**: Contains Next.js pages and route definitions, representing the core of the application including layouts, static pages, and dynamic routes.
  - **components/**: Contains reusable React components. Each component is organized in its own folder and typically includes:
    * The component implementation (e.g., `SomeComponent.tsx`).
    * A dedicated `__tests__/` folder with test files:
      · **Unit tests**: `SomeComponent.unit.test.tsx` validates isolated functionality.
      · **Integration tests**: `SomeComponent.integration.test.tsx` verifies interactions with other modules.
      · **Internationalization tests**: `SomeComponent.i18n.test.tsx` checks locale-specific rendering and translations.
  - **tests/**: Contains higher-level test suites covering the overall application.
    * **e2e/**: End-to-end tests written with Cypress (e.g., `login.e2e.cy.tsx`, `dashboard.e2e.cy.tsx`).
    * **visual-tests/**: Visual regression tests that compare UI snapshots against baseline images with Cypress and cypress-visual-regression plugin (e.g., `home-page.visual.cy.tsx`).
    * **performance-tests/**: Performance testing outputs (e.g., Lighthouse reports in JSON and HTML).
    * **accessibility-tests/**: Accessibility tests using tools like axe-core integrated with Cypress (e.g., `home-page.accessibility.cy.tsx`).
    * **i18n-tests/**: End-to-end tests verifying internationalization, ensuring that locale switching and translations work correctly (e.g., `home-page.i18n.cy.tsx`).
    * **cross-browser-tests/**: Tests aimed at verifying consistency across different browsers. You can place tests here (e.g., `home-page.cross-browser.cy.tsx`)

- **Configuration Files & Root Assets**: Files such as `vitest.config.ts`, `cypress.config.ts`, `eslint.config.mjs`, `vitest.setup.ts`, and `package.json` reside at the root, defining the testing configurations and scripts.

4.2 Command Reference

The Next.js template includes a comprehensive set of `npm` scripts to streamline test execution. Integrate them also to your projects. You can use scripts from the `package.json` from the Next.js template. Below is a summary of the key commands:

Vitest-Based Tests

- **Run all Vitest tests:** `npm run test`
- **Unit Tests:**

  - Run: `npm run test:unit`
  - Watch mode: `npm run test:unit:watch`
  - Coverage: `npm run test:unit:coverage`
- **Integration Tests:**
  - Run: `npm run test:integration`
  - Watch mode: `npm run test:integration:watch`
  - Coverage: `npm run test:integration:coverage`
- **Internationalization (i18n) Tests:**
  - Run: `npm run test:i18n`
  - Watch mode: `npm run test:i18n:watch`
  - Coverage: `npm run test:i18n:coverage`

Cypress-Based Tests

- **General Cypress Run:** `npm run test:cypress`
- **Cypress GUI:** `npm run test:cypress:open`
- **End-to-End (E2E) Tests:**
  - Run: `npm run test:e2e` or use browser-specific commands such as `npm run test:e2e:chrome`, `npm run test:e2e:firefox`, etc.
  - Open GUI: `npm run test:e2e:open`
- **Visual Regression Tests:**
  - Capture Baseline: `npm run test:visual:initial`
  - Run Comparison: `npm run test:visual`
  - Open GUI: `npm run test:visual:open`
- **Accessibility Tests:**
  - Run: `npm run test:accessibility`
  - Open GUI: `npm run test:accessibility:open`

Performance Testing

- **Run Performance Tests (Lighthouse):** `npm run test:performance`

Cross-Browser Testing

- **Run Tests in Multiple Browsers:** For example, `npm run test:e2e:all-browsers` sequentially executes tests across Chrome, Firefox, Edge, and Electron.

Run All Tests

- **Complete Pipeline:** `npm run test:all` sequentially executes Vitest tests, visual regression tests (initial capture and comparison), accessibility tests, cross-browser E2E tests, and performance tests.

4.3 Conclusion

An optimal folder structure and a well-defined set of commands not only promote best practices in front-end testing but also ensure that teams can quickly and efficiently assess application quality. The structure and commands presented in this guide—employed in the Next.js template—cover all critical aspects of testing: unit, integration, end-to-end, visual regression, performance, accessibility, security, internationalization, and cross-browser testing. By following this guide, developers can seamlessly integrate robust testing practices into their workflow, ensuring that applications are reliable, performant, and accessible to all users.

This structure promotes a clear separation of concerns, where each testing type is isolated in its own directory. It not only simplifies the process of locating test files but also allows developers to quickly identify which tests correspond to unit, integration, or end-to-end scenarios. Such an approach has been successfully adopted in the Next.js template and is highly recommended for modern front-end projects.

Chapter 5

Best Practices in Front-End Testing

Front-end testing plays a crucial role in delivering reliable, accessible, and high-performance web applications. By adopting a structured testing strategy, teams can mitigate risks, reduce technical debt, and enhance user satisfaction. The following best practices provide a roadmap for implementing a robust testing framework in modern React and Next.js projects.

Test Automation Pyramid

Implement at least unit testing, integration testing, and end-to-end (E2E) testing as defined by the **Test Automation Pyramid** introduced by Mike Cohn in the book Succeeding with Agile: Software Development Using Scrum [2]. A solid base of fast and isolated unit tests is complemented by integration tests that verify module interactions, and E2E tests that simulate real user workflows. This layered strategy not only catches defects early but also ensures comprehensive coverage while maintaining efficiency.

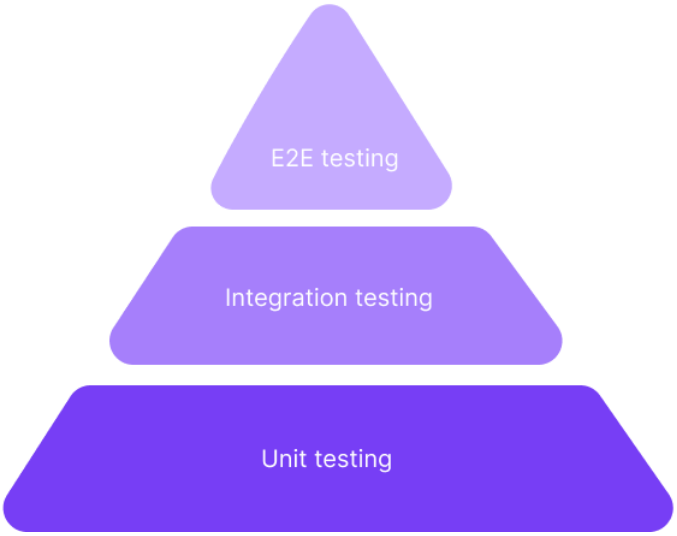

Figure 5.1: The Test Automation Pyramid

Follow the FIRST Principles

Unit tests should adhere to the FIRST principles:

- **Fast**: Tests should execute quickly to provide immediate feedback.
- **Isolated**: Each test should focus on a single unit of functionality.
- **Repeatable**: Tests must yield consistent results under the same conditions.
- **Self-validating**: Each test should clearly indicate a pass or fail outcome.
- **Timely**: Write tests early in the development process to catch issues before they escalate.

These principles, introduced by Jeff Langr and Tim Ottinger, help create a maintainable and reliable test suite. [3]

Start Testing Early and Continuously

Starting testing early in the development process allows teams to identify potential issues before they escalate, saving time and resources. Continuous testing throughout the development lifecycle ensures that new features or updates do not introduce regressions or defects. This approach aligns well with iterative and Agile methodologies.

Keep Tests Atomic

Tests should be small, isolated, and fast. Atomic tests enable developers to pinpoint failures precisely and debug issues more effectively. This isolation minimizes the impact of a single test failure on the overall suite.

Test Real User Scenarios

Prioritize tests that simulate realistic user interactions. End-to-end tests are particularly valuable for verifying complete workflows such as login, checkout, or data submission, ensuring that the application functions as expected under real-world conditions.

Ensure Cross-Browser Compatibility

Modern applications must work consistently across different browsers and devices. Cross-browser testing helps identify discrepancies in rendering or functionality caused by differences in browser engines. Automating these tests using tools like Cypress ensures that the user experience remains consistent regardless of the platform.

Mock External Dependencies

To maintain predictable and reliable tests, isolate your system under test by mocking external APIs and services. This approach prevents external factors from affecting test outcomes and allows you to simulate edge cases more effectively.

Leverage Continuous Integration (CI)

Integrate your testing processes into a CI/CD pipeline so that tests run automatically with every code change. This continuous feedback loop enables prompt issue resolution and ensures that only high-quality code is merged into the main branch.

Regularly Update Tools and Documentation

The front-end ecosystem evolves rapidly. Keep your testing tools, libraries, and dependencies up to date, and ensure that your testing documentation reflects current practices. Regular updates and clear documentation help maintain a robust testing environment.

Collaborate Effectively

Close collaboration between developers and testers is essential. A shared understanding of testing goals and strategies ensures that tests align with application requirements and user expectations. Effective communication helps resolve issues quickly and enhances the overall quality of the product. [4]

Balance Test Coverage with Practicality

While comprehensive test coverage is ideal, striving for 100% coverage can be impractical and may introduce diminishing returns. Focus on critical paths, high-risk areas, and frequently used features. Prioritize quality over quantity to ensure that testing efforts are efficient and effective.

Conclusion

By following these best practices, front-end teams can implement a robust testing framework that enhances code quality, improves reliability, and delivers a superior user experience. This structured approach to testing—spanning unit, integration, end-to-end, visual regression, performance, accessibility, security, internationalization, and cross-browser testing—empowers developers to build modern web applications with confidence.

Bibliography

[1] PANDA, Automation. *Arrange-Act-Assert: A Pattern for WritingGood Tests* [online]. Automation Panda, 2020 [visited on 2025-02-27]. Available from: `https://automationpanda.com/2020/07/07/arrange-act-assert-a-pattern-for-writing-good-tests/`

[2] COHN, Mike. *Succeeding with Agile: Software Development Using Scrum.* 1st. Addison-Wesley Professional, 2009. `isbn 978-0-321- 57936-2.`

[3] LANGR, J.; OTTINGER, T.; PFALZER, S.D. *Agile in a Flash: Speed-Learning Agile Software Development.* Pragmatic Bookshelf, 2011. Pragmatic Bookshelf Series. `isbn 978-1-934356-71-5.` Available also from: https://books.google.cz/books?id=odWicQAACAAJ.

[4] BOSE, Shreya. *15 Best Test Automation Practices to Follow in 2024* [online]. BrowserStack, 2024 [visited on 2025-01-05]. Available from: https://www.browserstack.com/guide/10-test-automation-best-practices.