Testing

Examples

- Next.js with Cypress
- Next.js with Playwright
- Next.js with Jest and React Testing Library
- Next.js with Vitest

Learn how to set up Next.js with commonly used testing tools: Cypress, Playwright, and Jest with React Testing Library.

Cypress

Cypress is a test runner used for **End-to-End (E2E)** and **Component Testing**.

Quickstart

You can use `create-next-app` with the with-cypress example to quickly get started.

Terminal

```
npx create-next-app@latest --example with-cypress with-cypress-app
```

Manual setup

To get started with Cypress, install the `cypress` package:

Terminal

```
npm install --save-dev cypress
```

Add Cypress to the `package.json` scripts field:

```
"scripts": {
  "dev": "next dev",
  "build": "next build",
  "start": "next start",
  "cypress": "cypress open",
}
```

Run Cypress for the first time to generate examples that use their recommended folder structure:

Terminal

```
npm run cypress
```

You can look through the generated examples and the Writing Your First Test ↗ section of the Cypress Documentation to help you get familiar with Cypress.

Should I use E2E or Component Tests?

The Cypress docs contain a guide ↗ on the difference between these two types of tests and when it is appropriate to use each.

Creating your first Cypress E2E test

Assuming the following two Next.js pages:

pages/index.js

```
import Link from 'next/link';

export default function Home() {
  return (
    <nav>
      <h1>Homepage</h1>
      <Link href="/about">About</Link>
    </nav>
  );
}
```

pages/about.js

```
export default function About() {
  return (
    <div>
      <h1>About Page</h1>
      <Link href="/">Homepage</Link>
    </div>
  );
}
```

Add a test to check your navigation is working correctly:

cypress/e2e/app.cy.js

```
describe('Navigation', () => {
  it('should navigate to the about page', () => {
    // Start from the index page
    cy.visit('http://localhost:3000/');

    // Find a link with an href attribute containing "about" and click it
    cy.get('a[href*="about"]').click();

    // The new url should include "/about"
    cy.url().should('include', '/about');

    // The new page should contain an h1 with "About page"
    cy.get('h1').contains('About Page');
  });
});
```

You can use `cy.visit("/")` instead of `cy.visit("http://localhost:3000/")` if you add `baseUrl: 'http://localhost:3000'` to the `cypress.config.js` configuration file.

Creating your first Cypress component test

Component tests build and mount a specific component without having to bundle your whole application or launch a server. This allows for more performant tests that still provide visual feedback and the same API used for Cypress E2E tests.

Note: Since component tests do not launch a Next.js server, capabilities like `<Image />` and `getServerSideProps` which rely on a server being available will not function out-of-the-box. See the Cypress Next.js docs ↗ for examples of getting these features working within component tests.

Assuming the same components from the previous section, add a test to validate a component is rendering the expected output:

pages/about.cy.js

```
import AboutPage from './about.js';

describe('<AboutPage />', () => {
  it('should render and display expected content', () => {
    // Mount the React component for the About page
    cy.mount(<AboutPage />);

    // The new page should contain an h1 with "About page"
    cy.get('h1').contains('About Page');

    // Validate that a link with the expected URL is present
    // *Following* the link is better suited to an E2E test
    cy.get('a[href="/"]').should('be.visible');
  });
});
```

Running your Cypress tests

E2E Tests

Since Cypress E2E tests are testing a real Next.js application they require the Next.js server to be running prior to starting Cypress. We recommend running your tests against your production code to more closely resemble how your application will behave.

Run `npm run build` and `npm run start`, then run `npm run cypress -- --e2e` in another terminal window to start Cypress and run your E2E testing suite.

Note: Alternatively, you can install the `start-server-and-test` package and add it to the `package.json` scripts field: `"test": "start-server-and-test start http://localhost:3000 cypress"` to start the Next.js production server in conjunction with Cypress. Remember to rebuild your application after new changes.

Component Tests

Run `npm run cypress -- --component` to start Cypress and execute your component testing suite.

Getting ready for Continuous Integration (CI)

You will have noticed that running Cypress so far has opened an interactive browser which is not ideal for CI environments. You can also run Cypress headlessly using the `cypress run` command:

package.json

```

"scripts": {
```

```
  //...
  "e2e": "start-server-and-test dev http://localhost:3000 \"cypress open --e2e\"",
  "e2e:headless": "start-server-and-test dev http://localhost:3000 \"cypress run --e2e\"",
  "component": "cypress open --component",
  "component:headless": "cypress run --component"
}
```

You can learn more about Cypress and Continuous Integration from these resources:

- Cypress Continuous Integration Docs ↗
- Cypress GitHub Actions Guide ↗
- Official Cypress GitHub Action ↗

Playwright

Playwright is a testing framework that lets you automate Chromium, Firefox, and WebKit with a single API. You can use it to write **End-to-End (E2E)** and **Integration** tests across all platforms.

Quickstart

The fastest way to get started is to use `create-next-app` with the with-playwright example ↗. This will create a Next.js project complete with Playwright all set up.

Terminal

```
npx create-next-app@latest --example with-playwright with-playwright-app
```

Manual setup

You can also use `npm init playwright` to add Playwright to an existing `NPM` project.

To manually get started with Playwright, install the `@playwright/test` package:

Terminal

```
npm install --save-dev @playwright/test
```

Add Playwright to the `package.json` scripts field:

```
"scripts": {
  "dev": "next dev",
  "build": "next build",
  "start": "next start",
  "test:e2e": "playwright test",
}
```

Creating your first Playwright end-to-end test

Assuming the following two Next.js pages:

pages/index.js

```
import Link from 'next/link';

export default function Home() {
  return (
    <nav>
      <Link href="/about">About</Link>
    </nav>
  );
}
```

pages/about.js

```
export default function About() {
  return (
    <div>
      <h1>About Page</h1>
    </div>
  );
}
```

Add a test to verify that your navigation is working correctly:

e2e/example.spec.ts

```
import { test, expect } from '@playwright/test';

test('should navigate to the about page', async ({ page }) => {
  // Start from the index page (the baseURL is set via the webServer in the playwright.co
  await page.goto('http://localhost:3000/');
  // Find an element with the text 'About Page' and click on it
  await page.click('text=About');
  // The new URL should be "/about" (baseURL is used there)
  await expect(page).toHaveURL('http://localhost:3000/about');
  // The new page should contain an h1 with "About Page"
```

```
11    await expect(page.locator('h1')).toContainText('About Page');
12  });
```

You can use `page.goto("/")` instead of `page.goto("http://localhost:3000/")`, if you add `"baseURL": "http://localhost:3000"` ↗ to the `playwright.config.ts` configuration file.

Running your Playwright tests

Since Playwright is testing a real Next.js application, it requires the Next.js server to be running prior to starting Playwright. It is recommended to run your tests against your production code to more closely resemble how your application will behave.

Run `npm run build` and `npm run start`, then run `npm run test:e2e` in another terminal window to run the Playwright tests.

> **Note**: Alternatively, you can use the `webServer` ↗ feature to let Playwright start the development server and wait until it's fully available.

Running Playwright on Continuous Integration (CI)

Playwright will by default run your tests in the headless mode ↗. To install all the Playwright dependencies, run `npx playwright install-deps`.

You can learn more about Playwright and Continuous Integration from these resources:

- Getting started with Playwright ↗
- Use a development server ↗
- Playwright on your CI provider ↗

Jest and React Testing Library

Jest and React Testing Library are frequently used together for **Unit Testing**. There are three ways you can start using Jest within your Next.js application:

1. Using one of our quickstart examples
2. With the Next.js Rust Compiler
3. With Babel

The following sections will go through how you can set up Jest with each of these options:

Quickstart

You can use `create-next-app` with the with-jest↗ example to quickly get started with Jest and React Testing Library:

```
Terminal
npx create-next-app@latest --example with-jest with-jest-app
```

Setting up Jest (with the Rust Compiler)

Since the release of Next.js 12↗, Next.js now has built-in configuration for Jest.

To set up Jest, install `jest`, `jest-environment-jsdom`, `@testing-library/react`, `@testing-library/jest-dom`:

```
Terminal
npm install --save-dev jest jest-environment-jsdom @testing-library/react @testing-library/je
```

Create a `jest.config.mjs` file in your project's root directory and add the following:

```
jest.config.mjs
import nextJest from 'next/jest.js';

const createJestConfig = nextJest({
  // Provide the path to your Next.js app to load next.config.js and .env files in your t
  dir: './',
});

// Add any custom config to be passed to Jest
/** @type {import('jest').Config} */
const config = {
  // Add more setup options before each test is run
  // setupFilesAfterEnv: ['<rootDir>/jest.setup.js'],

  testEnvironment: 'jest-environment-jsdom',
};

// createJestConfig is exported this way to ensure that next/jest can load the Next.js co
export default createJestConfig(config);
```

Under the hood, `next/jest` is automatically configuring Jest for you, including:

- Setting up `transform` using SWC
- Auto mocking stylesheets (`.css`, `.module.css`, and their scss variants), image imports and `next/font`
- Loading `.env` (and all variants) into `process.env`
- Ignoring `node_modules` from test resolving and transforms
- Ignoring `.next` from test resolving
- Loading `next.config.js` for flags that enable SWC transforms

Note: To test environment variables directly, load them manually in a separate setup script or in your `jest.config.js` file. For more information, please see Test Environment Variables.

Setting up Jest (with Babel)

If you opt out of the Rust Compiler, you will need to manually configure Jest and install `babel-jest` and `identity-obj-proxy` in addition to the packages above.

Here are the recommended options to configure Jest for Next.js:

jest.config.js

```
module.exports = {
  collectCoverage: true,
  // on node 14.x coverage provider v8 offers good speed and more or less good report
  coverageProvider: 'v8',
  collectCoverageFrom: [
    '**/*.{js,jsx,ts,tsx}',
    '!**/*.d.ts',
    '!**/node_modules/**',
    '!<rootDir>/out/**',
    '!<rootDir>/.next/**',
    '!<rootDir>/*.config.js',
    '!<rootDir>/coverage/**',
  ],
  moduleNameMapper: {
    // Handle CSS imports (with CSS modules)
    // https://jestjs.io/docs/webpack#mocking-css-modules
    '^.+\\.module\\.(css|sass|scss)$': 'identity-obj-proxy',

    // Handle CSS imports (without CSS modules)
    '^.+\\.(css|sass|scss)$': '<rootDir>/__mocks__/styleMock.js',

    // Handle image imports
    // https://jestjs.io/docs/webpack#handling-static-assets
```

```
    '^.+\\.(png|jpg|jpeg|gif|webp|avif|ico|bmp|svg)$/i': `<rootDir>/__mocks__/fileMock.js

    // Handle module aliases
    '^@/components/(.*)$': '<rootDir>/components/$1',
  },
  // Add more setup options before each test is run
  // setupFilesAfterEnv: ['<rootDir>/jest.setup.js'],
  testPathIgnorePatterns: ['<rootDir>/node_modules/', '<rootDir>/.next/'],
  testEnvironment: 'jsdom',
  transform: {
    // Use babel-jest to transpile tests with the next/babel preset
    // https://jestjs.io/docs/configuration#transform-objectstring-pathtotransformer--pat
    '^.+\\.(js|jsx|ts|tsx)$': ['babel-jest', { presets: ['next/babel'] }],
  },
  transformIgnorePatterns: [
    '/node_modules/',
    '^.+\\.module\\.(css|sass|scss)$',
  ],
};
```

You can learn more about each configuration option in the Jest docs ↗.

Handling stylesheets and image imports

Stylesheets and images aren't used in the tests but importing them may cause errors, so they will need to be mocked. Create the mock files referenced in the configuration above - `fileMock.js` and `styleMock.js` - inside a `__mocks__` directory:

__mocks__/fileMock.js

```
module.exports = {
  src: '/img.jpg',
  height: 24,
  width: 24,
  blurDataURL: '',
};
```

__mocks__/styleMock.js

```
module.exports = {};
```

For more information on handling static assets, please refer to the Jest Docs ↗.

Optional: Extend Jest with custom matchers

`@testing-library/jest-dom` includes a set of convenient custom matchers ↗ such as `.toBeInTheDocument()` making it easier to write tests. You can import the custom matchers for every test

by adding the following option to the Jest configuration file:

jest.config.js

```
setupFilesAfterEnv: ['<rootDir>/jest.setup.js'];
```

Then, inside `jest.setup.js`, add the following import:

jest.setup.js

```
import '@testing-library/jest-dom/extend-expect';
```

If you need to add more setup options before each test, it's common to add them to the `jest.setup.js` file above.

Optional: Absolute Imports and Module Path Aliases

If your project is using Module Path Aliases, you will need to configure Jest to resolve the imports by matching the paths option in the `jsconfig.json` file with the `moduleNameMapper` option in the `jest.config.js` file. For example:

tsconfig.json

```
{
  "compilerOptions": {
    "baseUrl": ".",
    "paths": {
      "@/components/*": ["components/*"]
    }
  }
}
```

jest.config.js

```
moduleNameMapper: {
  '^@/components/(.*)$': '<rootDir>/components/$1',
}
```

Creating your tests:

Add a test script to package.json

Add the Jest executable in watch mode to the `package.json` scripts:

```
"scripts": {
  "dev": "next dev",
  "build": "next build",
  "start": "next start",
  "test": "jest --watch"
}
```

`jest --watch` will re-run tests when a file is changed. For more Jest CLI options, please refer to the Jest Docs↗.

Create your first tests

Your project is now ready to run tests. Follow Jest's convention by adding tests to the `__tests__` folder in your project's root directory.

For example, we can add a test to check if the `<Home />` component successfully renders a heading:

__tests__/index.test.jsx

```
import { render, screen } from '@testing-library/react';
import Home from '../pages/index';
import '@testing-library/jest-dom';

describe('Home', () => {
  it('renders a heading', () => {
    render(<Home />);

    const heading = screen.getByRole('heading', {
      name: /welcome to next\.js!/i,
    });

    expect(heading).toBeInTheDocument();
  });
});
```

Optionally, add a snapshot test↗ to keep track of any unexpected changes to your `<Home />` component:

__tests__/snapshot.js

```
import { render } from '@testing-library/react';
import Home from '../pages/index';

it('renders homepage unchanged', () => {
  const { container } = render(<Home />);
  expect(container).toMatchSnapshot();
```

```
});
```

Note: Test files should not be included inside the Pages Router because any files inside the Pages Router are considered routes.

Running your test suite

Run `npm run test` to run your test suite. After your tests pass or fail, you will notice a list of interactive Jest commands that will be helpful as you add more tests.

For further reading, you may find these resources helpful:

- Jest Docs ↗
- React Testing Library Docs ↗
- Testing Playground ↗ - use good testing practices to match elements.

Community Packages and Examples

The Next.js community has created packages and articles you may find helpful:

- next-router-mock ↗ for Storybook.
- Test Preview Vercel Deploys with Cypress ↗ by Gleb Bahmutov.

For more information on what to read next, we recommend:

- pages/basic-features/environment-variables#test-environment-variables