End-to-End Testing

Cypress is one of the options when it comes to end-to-end (E2E) testing. Cypress is an all-in-one testing framework focused on E2E testing, meaning that you don't have to install 10 different things to get your test suite set up. You can write your first passing test in minutes without any configuration with the help of Cypress' API, which is easy to read and understand. It runs tests as fast as your browser can render content, which also makes test-driven development possible. You'll also profit from the time travel feature or the extensive debugging capabilities with Chrome DevTools. You can also use it with Gatsby, and this guide will explain how.

To run Gatsby's development server and Cypress at the same time, you'll use the little helper start-server-and-test. If you're already using @testing-library/react for unit testing, you might want to install @testing-library/cypress, too. This way you can use the same methods you used with `@testing-library/react` in your Cypress tests. Install the following packages to your `devDependencies`:

```
npm install --save-dev cypress start-server-and-test
```

We also want the URLs used by `cy.visit()` or `cy.request()` to be prefixed, hence you have to create the file `cypress.json` at the root of your project with the following content:

```
{
  "baseUrl": "http://localhost:8000/"
}
```

Last but not least you add additional scripts to your `package.json` to run Cypress:

```
{
  "scripts": {
    "develop": "gatsby develop",
    // highlight-start
    "cy:open": "cypress open",
    "test:e2e": "start-server-and-test develop http://localhost:8000 cy:open"
    // highlight-end
  }
}
```

Type `npm run test:e2e` in your command line and see Cypress running for the first time. A folder named `cypress` will be created at the root of your project and a new application window will pop up. Cypress' getting started guide is a good start to learn how to write tests!

Important note: If you are running Gatsby with the `--https` flag, whether using your own or automatically generated certificates, you will also need to tell `start-server-and-test` to disable HTTPS certificate checks (otherwise it will wait forever and never actually launch Cypress. You do this by passing in an environmental variable: `START_SERVER_AND_TEST_INSECURE=1`. start-server-and-test docs).

This means your `test:e2e` script would look like this:

```
"test:e2e": "START_SERVER_AND_TEST_INSECURE=1 start-server-and-test develop http://localhost
```

Continuous Integration

If you want to run Cypress in Continuous Integration (CI), you have to use `cypress run` instead of `cypress open`:

```
{
  "scripts": {
    "develop": "gatsby develop",
    "cy:open": "cypress open",
    "cy:run": "cypress run", // highlight-line
    "test:e2e": "start-server-and-test develop http://localhost:8000 cy:open",
    "test:e2e:ci": "start-server-and-test develop http://localhost:8000 cy:run" // highlight
  }
}
```

Please read the Cypress' official documentation on CI if you want to know how to set up Travis or GitLab with Cypress. You can also use a GitHub Action to implement basic CI directly in GitHub with the official Cypress GitHub action.

Writing tests

See an example repository with the completed tests.

A good use case for writing automated end-to-end tests is asserting **accessibility** with cypress-axe, a Cypress plugin that incorporates the axe accessibility testing API. While some manual testing is still required to ensure good web accessibility, automation can ease the burden on human testers.

To use cypress-axe, you have to install the `cypress-axe` and axe-core packages. You'll also use some commands from @testing-library/cypress to select elements — see best practices for selecting elements.

```
npm install --save-dev cypress-axe axe-core @testing-library/cypress
```

Then you add the `cypress-axe` and `@testing-library/cypress` commands in `cypress/support/index.js`:

```
import "./commands"
//highlight-start
import "cypress-axe"
import "@testing-library/cypress/add-commands"
//highlight-end
```

Cypress will look for tests inside the `cypress/integration` folder, but you can change the default test folder if you want. Because you're writing end-to-end tests, create a new folder called `cypress/e2e`, and use it as your `integrationFolder` in your `cypress.json`:

```
{
  "baseUrl": "http://localhost:8000/",
  "integrationFolder": "cypress/e2e" // highlight-line
}
```

Create a new file inside `cypress/e2e` folder and name it `accessibility.test.js`.

You'll use the `beforeEach` hook to run some commands before each test. This is what they do:

- First, Cypress loads the homepage with the cy.visit command and waits until the page loads (load event).
- Then, Cypress waits for the `main` element to appear with the cy.get command because Gatsby (in development) can trigger the load event before any content is displayed on the screen. If you don't wait, the tests may fail.
- After that, it initializes the `axe` accessibility testing API with the `injectAxe` command.

Finally, inside the first test, you'll use the `checkA11y` command from `cypress-axe` to check for accessibility violations:

```
/// <reference types="Cypress" />

describe("Accessibility tests", () => {
  beforeEach(() => {
    cy.visit("/").get("main").injectAxe()
  })
  it("Has no detectable accessibility violations on load", () => {
    cy.checkA11y()
  })
})
```

The `/// <reference types="Cypress" />` line at the top gives you autocompletion for the Cypress commands.

You can now type `npm run test:e2e` in your terminal to run the test. If you already have the development server open, you can type `npm run cy:open` instead.

One thing to keep in mind is that you can't always see the exact error message from the sidebar (command log). For that, you have to open the browser developer console and find the message in the output. You can see how an accessibility error looks in the cypress-axe GitHub page.

You don't have to use the `checkA11y` command only on page load. You can perform an action inside your application and check for accessibility errors again. This is useful if you open a modal or click a "dark mode" switch, for example.

The following test is for the gatsby-default-starter. Cypress visits the homepage and searches for the link that goes to page 2 with the `findByText` command from `@testing-library/cypress`. Then, Cypress clicks that link and `cypress-axe` checks for accessibility errors on the second page.

```
/// <reference types="Cypress" />

describe("Accessibility tests", () => {
  beforeEach(() => {
    cy.visit("/").get("main").injectAxe()
  })
  it("Has no detectable accessibility violations on load", () => {
    cy.checkA11y()
  })
  // highlight-start
  it("Navigates to page 2 and checks for accessibility violations", () => {
    cy.findByText(/go to page 2/i)
      .click()
      .checkA11y()
  })
  // highlight-end
})
```

You can also make assertions with the cy.should command.

You'll now write another test for the `gatsby-default-starter` homepage. In this test you get the footer link, focus on it with the cy.focus command, and make some assertions:

```
/// <reference types="Cypress" />

describe("Accessibility tests", () => {
  beforeEach(() => {
    cy.visit("/").get("main").injectAxe()
  })
  it("Has no detectable accessibility violations on load", () => {
```

```
    cy.checkA11y()
  })
  it("Navigates to page 2 and checks for accessibility violations", () => {
    cy.findByText(/go to page 2/i)
      .click()
      .checkA11y()
  })
  // highlight-start
  it("Focuses on the footer link and asserts its attributes", () => {
    cy.findAllByText("Gatsby").focus()

    cy.focused()
      .should("have.text", "Gatsby")
      .should("have.attr", "href", "https://www.gatsbyjs.com")
      .should("not.have.css", "outline-width", "0px")
  })
  // highlight-end
})
```

The autocomplete feature comes in handy with the `should` command because you can explore all the possible assertions.

Customize axe options

You can disable a specific accessibility rule, or run only a subset of the rules by changing the axe.run options — see in the link what other options you can change. To do that, you pass an options object as the second parameter to the `checkA11y` command:

```
const axeRunOptions = {
  rules: {
    "rule-id": { enabled: false },
  },
}

cy.checkA11y(null, axeRunOptions)
```

The `checkA11y` command runs the `axe.run` method under the hood. This method can slow down Cypress and make the window unresponsive if you have a lot of elements on the page. To overcome this, you can use the axe.run context and exclude some elements to improve speed. For example, code blocks from gatsby-remark-prismjs can slow down your tests. You can test if they are accessible in a single test, and exclude them in the rest of your tests.

```
const axeRunOptions = {
  rules: {
    "rule-id": { enabled: false },
  },
```

```
}
// highlight-start
const axeRunContext = {
  exclude: [[".gatsby-highlight"], ["#an-id"]],
}
// highlight-end

// highlight-next-line
cy.checkA11y(axeRunContext, axeRunOptions)
```

Other resources

You can read Cypress' getting started guide to revisit the concepts you explored here and learn about many more. If you want a more detailed guide, you can read Cypress' Core Concepts. If you want to learn more about a specific Cypress command, check the API Reference.