Installation

First, install cypress with npm: `npm install cypress --save-dev`

Add the following to `"scripts":` in 'package.json': `"cypress:open": "cypress open"`

The above script allows you to open the Cypress Dashboard with the following command:

```
npm run cypress:open
```

Now there will be a new ‘cypress’ folder in the root directory of the application.

For further information, the Cypress Docs have instructions for installation: https://docs.cypress.io/guides/getting-started/installing-cypress.html#System-requirements

If you want to be able to use `DOM Testing Library`'s query commands off the cy object, run `npm install --save-dev cypress @testing-library/cypress`

Then add this line to your project's `cypress/support/commands.js`:

```
import '@testing-library/cypress/add-commands';
```

Eslint configuration for Cypress

Run the following command to install eslint for Cypress: `npm install eslint-plugin-cypress --save-dev`

To make eslint friendly for certain chai expressions, run the following command:

```
npm install --save-dev eslint-plugin-chai-friendly
```

Now, create a new 'eslintrc.json' file in the cypress directory to configure the linting rules for cypress. The following configuration will set both the cypress and chai-friendly to the recommended configuration:

```
{

    "extends": [
        "plugin:cypress/recommended",
        "plugin:chai-friendly/recommended"
    ]

}
```

Here is the link to the official docs for the Cypress ESLint plugin:

https://github.com/cypress-io/eslint-plugin-cypress - Connect to preview

Configuration

A cypress.json file is created in your project when you add Cypress to it. Definitely leverage this. You can can change the base url:

```
{
    "baseUrl": "http://localhost:3000"
}
```

Imagine running on a local environment and then you end up using a provider. As long as you don't change the variable paths, this could save a bunch of time by only having to refactor in one place. Note that the baseUrl change does not apply to a statement such as `cy.url().should('eq', '/login');` Cypress will look for an exact match for the provided parameter. To work around this, you can use 'include' instead of 'eq', which in some cases causes an issue(if you want to test for a redirect to the root page, any other page beginning with a '/' will pass the test). Alternatively, put in the entire URL in the second parameter for which you desire an exact match.

Check out the cypress docs for more info:
https://docs.cypress.io/guides/references/configuration.html#Options

Writing tests

Test files are written inside the integration cypress subdirectory that was created during installation (folder structure: cypress>integration>…>anyNameHere_spec.js)

to create a test, create a test file 'myTest_spec.js'. If you want a test that always passes, try this:

```
describe('My First Test', () => {
  it('Does not do much!', () => {
    expect(true).to.equal(true)
  })
})
```

If you want a test that tells you something you don't already know, try something like this:

```
describe('My First Test', () => {
  it('Visits the login page', () => {
  // I want to start at the login page:
    cy.visit('http://localhost:3000/login');
//Now I want to enter valid credentials in the inputs and test submitting them:
    //This finds an input with name="Email", types into it, and then checks that the
// value is now equal to what was typed
    cy.get('input[name="Email"]').type('japp@tree.com').should('have.value',
'japp@tree.com');
    //Much the same as above, but now an input with name="Password"
    cy.get('input[name="Password"]').type('MyPassword@$2').should('have.value',
'MyPassword@$2');
    // query for 'submit', which is the submit button on the page, and then click it
    cy.contains('submit').click();
  //Now, I want to know if I was re-directed to the root page
    // cy.url() checks the current url. the 'eq' parameter checks for an exact match
to second parameter as opposed to 'include' which would allow any url that contains
the second parameter and any additional characters to pass.
    cy.url().should('eq', 'http://localhost:3000/');
     //and now, to verify I am now logged in, check if the the login button changed to
logout and if the email now displays in the header
    cy.contains('Logout');
    cy.contains('japp@tree.com');
  });
});
```

Note that the .visit() statement on line 4 doesn't have any assertions. However, even if that were the only code we had, it will pass as long as Cypress hears a 2xx response. Otherwise, the test would fail. The point is that unless you want to test for more details, no assertions are necessary, it is a sort of built-in test. The same can go for .contains(), .get(), etc.

Cypress automatically detects a page transition event and will wait until the new page is loaded until it runs the test, so there is no need to manually set a timeout.

Cypress will re-run your tests after every save you make on your test files.

Cypress has a pretty good tutorial to get started:
https://docs.cypress.io/guides/getting-started/testing-your-app.html#Step-3-Configure-Cypress

Here is another resource that teaches Cypress in conjunction with @testing-library:
https://mattermost.com/blog/modern-ways-of-end-to-end-testing-with-cypress-js/