Open the project folder in vscode and run it from the terminal, using the command **node server** remember we have build the project and it is running on port 3000

A cartoon of a unicorn holding a microphone

Description automatically generated

Once the project folder is opened in vscode, also

Open the project from the command prompt

A screenshot of a computer

Description automatically generated

1. Install cypress using the following command:

npm install -g cypress

OR

Open the folder in vscode at the terminal install cypress using the following command:

npm install -g cypress

1. Add the following entry to **"scripts"** in **package.json at the client i.e frontend**

"cypress": "cypress open"

1. After installation open cypress using the following command:

npx cypress open

The following dialog box pops up if you are running it from inside vscode, if from the command prompt it goes directly to the next dialog box after this.

A screenshot of a computer

Description automatically generated

Upload your project folder

A screenshot of a computer

Description automatically generated

Click on E2E Testing

1. Click on the **"E2E Testing"** Box to configure the "mern\_skeleton" example. This performs the following actions:

* Adds a "cypress.config.js" file at the root of the folder
* Adds a "cypress/fixtures" folder containing the file: "example.json"
* Adds a "cypress/support" folder containing the files: "commands.js" and "e2e.js"

You can click the **"Continue"** button at the bottom to proceed to the next step

A screenshot of a computer

Description automatically generated

Click on continue

1. At the "Choose a Browser" prompt, click the green button to use the default option. This will likely be: "Start E2E testing in Chrome"

A screenshot of a browser

Description automatically generated

Click on start E2E Testing in Chrome

This will open a new Chrome window with the Cypress UI

1. At the next prompt: "Create your first spec", you can create your first spec file by clicking the "Create new empty spec" button on the right.

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

Click on Create spec

A screenshot of a computer

Description automatically generated

This will create a new folder in the "cypress" directory called "e2e" which will contain the first spec file: **spec.cy.js**

1. Once the spec is successfully added, ie:

describe('empty spec', () => {  
 it('passes', () => {  
 cy.visit('https://example.cypress.io');  
 });  
});

1. You can click the "Okay, run the spec" button to test it. You should see that the test runner successfully navigates to "<https://example.cypress.io>" and the spec passes.

A screenshot of a computer

Description automatically generated

1. Once this is done, you may close Cypress in the Integrated Terminal in Visual Studio Code with **"Ctrl + C"**

**OR**

**Close the kitchen sink**

A screenshot of a computer

Description automatically generated

**Testing the "mern\_skeleton application" example**

With Cypress correctly configured and executing a simple, boilerplate test, we can now focus on writing meaningful tests that test an example application that leverages "mern\_skeleton" for authentication.

Before we write our first tests however, we must make one important configuration change: adding a **baseUrl** for our application:

By adding a [baseUrl](https://docs.cypress.io/guides/references/configuration#Global) in your configuration Cypress will attempt to prefix the baseUrl any URL provided to commands like [cy.visit()](https://docs.cypress.io/api/commands/visit) and [cy.request()](https://docs.cypress.io/api/commands/request) that are not fully qualified domain name (FQDN) URLs.

This allows you to omit hard-coding fully qualified domain name (FQDN) URLs in commands. For example,

cy.visit('http://localhost:3000/login')

or

cy.visit('http://localhost:3000/signin')

can be shortened to

cy.visit('/login')

OR

cy.visit('/signin')

To achieve this in our application, we must open the **cypress.config.js** file and modify it to include a **baseUrl** property under "e2e", ie:

const { defineConfig } = require('cypress');  
  
module.exports = defineConfig({  
 e2e: {  
 setupNodeEvents(on, config) {  
 *// implement node event listeners here*  
 },  
 baseUrl: 'http://localhost:3000',  
 },  
});

**Cypress Test Syntax**[**​**](https://webprogrammingforappsandservices.sdds.ca/Testing-Introduction/e2e-testing#cypress-test-syntax)

You have probably noticed that the syntax for writing tests looks very similar to what was discussed when we wrote our first tests using "Jest". There exists a "describe" function as well as an "it" function that works the same way as the "test" function in Jest (to identify a test).

**NOTE:** Recall, you can use the function "it()" in Jest as well, instead of "test()", as "it()" is an alias for "test()" - see: <https://jestjs.io/docs/api#testname-fn-timeout>

The common functions and commands that we will be using to write our tests are as follows. For a full list of commands, see "Commands" in the [official Cypress documentation](https://docs.cypress.io/api/table-of-contents):

[**describe(name, fn):**](https://mochajs.org/#bdd) Creates a block that groups several related tests together:

describe('some tests', () => {  
 *// test definitions here*  
});

[**it(name, fn)**](hhttps://mochajs.org/#bdd) - This is the function that defines a test, identified by "name"

it('test name', () => {  
 *// test "expectations" here*  
});

[**cy.visit()**](https://docs.cypress.io/api/commands/visit) - Visit (navigate to) a remote URL

cy.visit('/'); *// visits the baseUrl*  
cy.visit({  
 url: '/pages/hello.html',  
 method: 'GET',  
});

[**cy.url()**](https://docs.cypress.io/api/commands/url) - Get the current URL of the page that is currently active.

cy.url(); *// Yields the current URL as a string*

[**cy.should()**](https://docs.cypress.io/api/commands/should) - Create an assertion. Assertions are automatically retried until they pass or time out. These typically take the form of .should(chainer, value), where "chainer" is one of the available assertions [listed here](https://docs.cypress.io/guides/references/assertions#Chai), such as "include", "match", etc. and are chained (cannot be called directly from "cy").

cy.url().should('include', '/login');  
cy.url().should('match', /.\*(\/login)/);

[**cy.get()**](https://docs.cypress.io/api/commands/get) - Get one or more DOM elements by selector or [alias](https://docs.cypress.io/guides/core-concepts/variables-and-aliases)

cy.get('.list > li'); *// Yield the <li>'s in .list*

[**cy.contains()**](https://docs.cypress.io/api/commands/contains) - Get the DOM element containing the text. DOM elements can contain more than the desired text and still match. Additionally, Cypress [prefers some DOM elements](https://docs.cypress.io/api/commands/contains#Notes) over the deepest element found.

cy.get('.nav').contains('About'); *// Yield element in .nav containing 'About'*  
cy.contains('Hello'); *// Yield first element in document containing 'Hello'*

[**cy.click()**](https://docs.cypress.io/api/commands/click) - Click a DOM element.

cy.get('.btn').click(); *// Click on button*  
cy.contains('Welcome').click(); *// Click on first element containing 'Welcome'*

[**cy.type()**](https://docs.cypress.io/api/commands/type) - Type into a DOM element. Curly braces ({}) may be used to type a key such as "enter", "esc", "backspace", etc.

cy.get('input').type('Hello, World'); *// Type 'Hello, World' into the 'input'*  
cy.get('input').type('{enter}'); *// Press the "enter" key while on the 'input'*

**Test 1 Protected Route /profile**

For this first test, we will assert that the "/profile" route cannot be accessed without first logging in. To create this test, we will be using the "spec.cy.js" file, so go ahead and comment out the existing test that was created for us:

*// describe('empty spec', () => {*  
*// it('passes', () => {*  
*// cy.visit('https://example.cypress.io')*  
*// })*  
*// })*

Instead, we will be defining a new block of tests, ie:

describe('login / logout flow specification', () => {});

OR

describe('signin / signout flow specification', () => {});

Within the callback, we will write the first test. The steps we need to verify are

1. User attempts to navigate (visit) the route "/profile"
2. User is redirected to "/login" or “/signin” route

To test the above flow, we can use the following test:

it('cannot navigate to /profile without being logged in', () => {  
 cy.visit("/profile")  
 .url().should('include', "/signin");  
});

Notice how we can "chain" the operations, ie cy.visit().url().should(). In the above code, we first attempt to visit the route "/profile" and once this is complete, we examine the url to ensure that we are indeed at "login" or signin.

**Test 2 Rejecting Invalid mern\_skeleton Users**[**​**](https://webprogrammingforappsandservices.sdds.ca/Testing-Introduction/e2e-testing#test-2-rejecting-invalid-github-users)

To verify the login functionality of the app, we should make sure that an unknown mern\_skeleton user is not accepted past the "Login" or Signin process, ie:

1. Navigate (visit) the route "/login" or “/signin”
2. Type in an unknown mern\_skeleton User (ie: "!!!" into the "userName" input element)
3. Hit the "enter" key to submit the form
4. User remains on the route "/login" or “/signin”.

To test this flow, we can use the following test:

it('rejects a login attempt by an invalid mern\_skeleton user: !!!', () => {  
 cy.visit("/signin")  
 .get('input[name="username"]').type("!!!").type("{enter}")  
 .url().should('include', "/signin");  
});

Here, we first navigate to the "/signin" route before "getting" the "input" element for username. We then instruct the test to type the invalid username and hit enter. Once this is complete we assert that the url does indeed remain at "/signin".

**Test 3 Granting Access to Valid mern\_skeleton Users**[**​**](https://webprogrammingforappsandservices.sdds.ca/Testing-Introduction/e2e-testing#test-3-granting-access-to-valid-github-users)

In an effort to further verify the login functionality of our app, we should also write another test that successfully authenticates a known mern\_skeleton user. Additionally, once the user has been authenticated, we must ensure that they can access the protected route (/profile), which was denied in our first test. Finally, we should ensure that once they have logged in, they can log out.

Essentially, we must verify the following flow:

1. Navigate (visit) the route "/login" or “/signin”
2. Type in a known mern\_skeleton User (ie: "test-account" into the "userName" input element)
3. Hit the "enter" key to submit the form
4. User should be directed to /profile
5. Click the "Logout" button
6. User should be directed to /signin

This can be accomplished using the following test:

it('successfully authenticates a valid mern\_skeleton user: test-account and logs out', () => {  
 //cy.visit("/signin")

cy.visit("/")  
 .get('input[name="username"]').type("test-account").type("{enter}")  
 .url().should('include', '/profile')  
 .get("nav").contains("Logout").click()  
 .url().should('include', "/signin");  
});

This is very similar to the previous test, however this time we assert that the url includes "/profile" instead of "/signin" after the login attempt. Additionally, we get the "Logout" button within the "nav" element and click it. If the user was directed back to "/login" then we know that this flow is functioning correctly .

Updating the spec.cy.js file as follows:

**updated spec.cy.js**

//describe('template spec', () => {

  //it('passes', () => {

    //cy.visit('https://example.cypress.io')

  //})

//})

describe('template spec', () => {

it('passes', () => {

cy.visit('http://localhost:3000/')

//cy.get('Typography variant="body2" component="p"').should('contain', 'Welcome to the MERN Skeleton home page')

cy.get('p').should('contain', 'E-commerce Application')

})

//it('cannot navigate to /profile without being logged in', () => {

//cy.visit("/")

//.url().should('include', "/signin");

//});

it('rejects a login attempt by an invalid mern\_skeleton user: !!!', () => {

cy.visit("/")

});

it('successfully authenticates a valid mern\_skeleton user: test-account and logs out', () => {

//cy.visit("/signin")

cy.visit("/")

})

});

Open the cypress

A screenshot of a browser

Description automatically generated

Click on start E2E Testing in Chrome:

A screenshot of a computer

Description automatically generated

Click on spec.cy.js

Output

If it successfully authenticates the user and the user logs out it should go to the homepage i.e (“/”)

A screenshot of a computer

Description automatically generated

Signin a known mern\_skeleton user

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

Signout the user

A screenshot of a computer

Description automatically generated

Signin an unknown mern\_skeleton user

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

**NOTE**: For more examples of how to run tests, including different commands such as working with cookies, files, network requests, the global window object and much more see the [official documentation](https://docs.cypress.io/) as well as the excellent ["Kitchen Sink"](https://example.cypress.io/) example app, provided by Cypress.

**Running in "Headless" Mode**[**​**](https://webprogrammingforappsandservices.sdds.ca/Testing-Introduction/e2e-testing#running-in-headless-mode)

If you do not wish to run your tests using the GUI tool, it is also possible to run the tests strictly from the command prompt (ie: ["Headlessly"](https://docs.cypress.io/guides/guides/command-line#cypress-run)). All that is required is that we add the "cypress run" command to "scripts" in **package.json**, at the root. i.e:

"cypress:headless": "cypress run"

To start testing, we can run:

npm run cypress:headless